



conference

proceedings

21st USENIX Security Symposium

***Bellevue, WA, USA
August 8–10, 2012***

Proceedings of the 21st USENIX Security Symposium

Bellevue, WA, USA August 8–10, 2012

Sponsored by



© 2012 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. Permission is granted to print, primarily for one person's exclusive use, a single copy of these Proceedings. USENIX acknowledges all trademarks herein.

ISBN 978-931971-95-9

USENIX Association

**Proceedings of the
21st USENIX Security Symposium**

**August 8–10, 2012
Bellevue, WA**

Conference Organizers

Program Chair

Tadayoshi Kohno, *University of Washington*

Program Committee

Ben Adida, *Mozilla*

Nikita Borisov, *University of Illinois at Urbana-Champaign*

David Brumley, *Carnegie Mellon University*

Kelly Caine, *Indiana University*

Srdjan Capkun, *ETH*

Sonia Chiasson, *Carleton University*

Mihai Christodorescu, *IBM T.J. Watson Research Center*

Anupam Datta, *Carnegie Mellon University*

William Enck, *North Carolina State University*

David Evans, *University of Virginia*

Kevin Fu, *University of Massachusetts Amherst*

Carrie Gates, *CA Technologies*

Roxana Geambasu, *Columbia University*

Ian Goldberg, *University of Waterloo*

Matthew Green, *Johns Hopkins University*

Urs Hengartner, *University of Waterloo*

Jaeyeon Jung, *Microsoft Research*

Sam King, *University of Illinois at Urbana-Champaign*

Engin Kirda, *Northeastern University*

Christian Kreibich, *International Computer Science Institute*

Kirill Levchenko, *University of California, San Diego*

David Lie, *University of Toronto*

Jonathan McCune, *Carnegie Mellon University*

David Molnar, *Microsoft Research*

Alex Moshchuk, *Microsoft Research*

Steven Murdoch, *University of Cambridge*

Cristina Nita-Rotaru, *Purdue University*

Niels Provos, *Google*

Vitaly Shmatikov, *University of Texas, Austin*

Diana Smetters, *Google*

Dan Wallach, *Rice University*

Invited Talks Committee

David Evans, *University of Virginia*

David Molnar, *Microsoft Research*

Bruce Potter, *Ponte Technologies*

Margo Seltzer, *Harvard School of Engineering and Applied Sciences and Oracle*

Poster Session Coordinator

Matt Bishop, *University of California, Davis*

Rump Session Chair

Matt Blaze, *University of Pennsylvania*

External Reviewers

Moheeb Abu Rajab

Ayo Akinyele

Ivan Alagenchev

Chaitrali Amrutkar

Dirk Balfanz

Jeremiah Blocki

Shuo Chen

Yikan Chen

Longze Chen

Shane Clark

Alan Dunn

Christina Garman

Chris Grier

Matt Hicks

David Huang

Suman Jana

Limin Jia

Haohui Mai

Prateek Mittal

Andres Molina-Markham

Shishir Nagaraja

James Newsome

Anh Nguyen

Matthew Pagano

Amir Rahmati

Mastooreh Salajegheh

Simon Sibomana

Arunesh Sinha

Sooel Son

Shuo Tang

Tianhao Tong

Michael Tschantz

Samee Zahur

Yuchen Zhou

21st USENIX Security Symposium
August 8–10, 2012
Bellevue, WA, USA

Message from the USENIX Security '12 Program Chair viii

Wednesday, August 8

Spam and Drugs

PharmaLeaks: Understanding the Business of Online Pharmaceutical Affiliate Programs. 1
Damon McCoy, *George Mason University*; Andreas Pitsillidis and Grant Jordan, *University of California, San Diego*; Nicholas Weaver and Christian Kreibich, *University of California, San Diego, and International Computer Science Institute*; Brian Krebs, *KrebsOnSecurity.com*; Geoffrey M. Voelker, Stefan Savage, and Kirill Levchenko, *University of California, San Diego*

B@bel: Leveraging Email Delivery for Spam Mitigation 17
Gianluca Stringhini and Manuel Egele, *University of California, Santa Barbara*; Apostolis Zarras and Thorsten Holz, *Ruhr-University Bochum*; Christopher Kruegel and Giovanni Vigna, *University of California, Santa Barbara*

Impact of Spam Exposure on User Engagement 33
Anirban Dasgupta, *Yahoo! Labs*; Kunal Punera, *RelateIQ Inc.*; Justin M. Rao, *Microsoft Research*; Xuanhui Wang, *Facebook*

CAPTCHAs and Password Strength

Security and Usability Challenges of Moving-Object CAPTCHAs: Decoding Codewords in Motion 49
Y. Xu, *University of North Carolina at Chapel Hill*; G. Reynaga and S. Chiasson, *Carleton University*; J.-M. Frahm and F. Monrose, *University of North Carolina at Chapel Hill*; P. van Oorschot, *Carleton University*

How Does Your Password Measure Up? The Effect of Strength Meters on Password Creation 65
Blase Ur, Patrick Gage Kelley, Saranga Komanduri, Joel Lee, Michael Maass, Michelle L. Mazurek, Timothy Passaro, Richard Shay, Timothy Vidas, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor, *Carnegie Mellon University*

I Forgot Your Password: Randomness Attacks Against PHP Applications. 81
George Argyros and Aggelos Kiayias, *University of Athens*

Browser Security

An Evaluation of the Google Chrome Extension Security Architecture. 97
Nicholas Carlini, Adrienne Porter Felt, and David Wagner, *University of California, Berkeley*

Establishing Browser Security Guarantees through Formal Shim Verification 113
Dongseok Jang, Zachary Tatlock, and Sorin Lerner, *University of California, San Diego*

The Brain

Neuroscience Meets Cryptography: Designing Crypto Primitives Secure Against Rubber Hose Attacks 129
Hristo Bojinov, *Stanford University*; Daniel Sanchez and Paul Reber, *Northwestern University*; Dan Boneh, *Stanford University*; Patrick Lincoln, *SRI*

On the Feasibility of Side-Channel Attacks with Brain-Computer Interfaces 143
Ivan Martinovic, *University of Oxford*; Doug Davies, Mario Frank, and Daniele Perito, *University of California, Berkeley*; Tomas Ros, *University of Geneva*; Dawn Song, *University of California, Berkeley*

Thursday, August 9

A Chance of Clouds

- Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud 159
Zhenyu Wu, Zhang Xu, and Haining Wang, *The College of William and Mary*
- Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services 175
Nuno Santos, *MPI-SWS*; Rodrigo Rodrigues, *CITI/Universidade Nova de Lisboa*; Krishna P. Gummadi, *MPI-SWS*; Stefan Saroiu, *Microsoft Research*
- STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. 189
Taesoo Kim, *MIT CSAIL*; Marcus Peinado and Gloria Mainar-Ruiz, *Microsoft Research*

Embedded Security

- Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices 205
Nadia Heninger, *UC San Diego*; Zakir Durumeric, Eric Wustrow, and J. Alex Halderman, *University of Michigan*
- TARDIS: Time and Remanence Decay in SRAM to Implement Secure Protocols on Embedded Devices without Clocks 221
Amir Rahmati and Mastooreh Salajegheh, *University of Massachusetts Amherst*; Dan Holcomb, *University of California, Berkeley*; Jacob Sorber, *Dartmouth College*; Wayne P. Bursleson and Kevin Fu, *University of Massachusetts Amherst*
- Gone in 360 Seconds: Hijacking with Hitag2 237
Roel Verdult and Flavio D. Garcia, *Radboud University Nijmegen*; Josep Balasch, *KU Leuven ESAT/COSIC and IBBT*

Secure Computation and PIR

- Taking Proof-Based Verified Computation a Few Steps Closer to Practicality 253
Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish, *The University of Texas at Austin*
- Optimally Robust Private Information Retrieval. 269
Casey Devet and Ian Goldberg, *University of Waterloo*; Nadia Heninger, *University of California, San Diego*
- Billion-Gate Secure Computation with Malicious Adversaries 285
Benjamin Kreuter, abhi shelat, and Chih-hao Shen, *University of Virginia*

Authentication and Secure Deletion

- Progressive Authentication: Deciding When to Authenticate on Mobile Phones 301
Oriana Riva, *Microsoft Research*; Chuan Qin, *University of South Carolina*; Karin Strauss and Dimitrios Lymberopoulos, *Microsoft Research*
- Origin-Bound Certificates: A Fresh Approach to Strong Client Authentication for the Web 317
Michael Dietz, *Rice University*; Alexei Czeskis, *University of Washington*; Dirk Balfanz, *Google Inc.*; Dan S. Wallach, *Rice University*
- Data Node Encrypted File System: Efficient Secure Deletion for Flash Memory 333
Joel Reardon, Srdjan Capkun, and David Basin, *ETH Zurich*

Privacy Enhancing Technologies and Network Traffic Analysis

- Throttling Tor Bandwidth Parasites. 349
Rob Jansen and Paul Syverson, *U.S. Naval Research Laboratory*; Nicholas Hopper, *University of Minnesota*
- Chimera: A Declarative Language for Streaming Network Traffic Analysis. 365
Kevin Borders, *National Security Agency*; Jonathan Springer, *Reservoir Labs*; Matthew Burnside, *National Security Agency*

Thursday, August 9 (continued)

New Attacks on Timing-based Network Flow Watermarks 381
Zi Lin and Nicholas Hopper, *University of Minnesota*

Friday, August 10

Web Security

On Breaking SAML: Be Whoever You Want to Be. 397
Juraj Somorovsky, *Ruhr-University Bochum*; Andreas Mayer, *Adolf Würth GmbH & Co. KG*; Jörg Schwenk, Marco Kampmann, and Meiko Jensen, *Ruhr-University Bochum*

Clickjacking: Attacks and Defenses. 413
Lin-Shung Huang, *Carnegie Mellon University*; Alex Moshchuk, Helen J. Wang, and Stuart Schechter, *Microsoft Research*; Collin Jackson, *Carnegie Mellon University*

Privilege Separation in HTML5 Applications 429
Devdatta Akhawe, Prateek Saxena, and Dawn Song, *University of California, Berkeley*

Software Security I

Fuzzing with Code Fragments 445
Christian Holler, *Mozilla Corporation*; Kim Herzig and Andreas Zeller, *Saarland University*

kGuard: Lightweight Kernel Protection against Return-to-User Attacks 459
Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis, *Columbia University*

Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization 475
Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum, *Vrije Universiteit Amsterdam*

Botnets and Web Security

From Throw-Away Traffic to Bots: Detecting the Rise of DGA-Based Malware 491
Manos Antonakakis, *Damballa Inc. and Georgia Institute of Technology*; Roberto Perdisci, *University of Georgia and Georgia Institute of Technology*; Yacin Nadji, *Georgia Institute of Technology*; Nikolaos Vasiloglou and Saeed Abu-Nimeh, *Damballa Inc.*; Wenke Lee and David Dagon, *Georgia Institute of Technology*

PUBCRAWL: Protecting Users and Businesses from CRAWLers 507
Gregoire Jacob, *University of California, Santa Barbara/Telecom SudParis*; Engin Kirda, *Northeastern University*; Christopher Kruegel and Giovanni Vigna, *University of California, Santa Barbara*

Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner. 523
Adam Doupe, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna, *University of California, Santa Barbara*

Mobile Devices

Aurasium: Practical Policy Enforcement for Android Applications 539
Rubin Xu, *Computer Laboratory, University of Cambridge*; Hassen Saïdi, *Computer Science Laboratory, SRI International*; Ross Anderson, *Computer Laboratory, University of Cambridge*

AdSplit: Separating Smartphone Advertising from Applications 553
Shashi Shekhar, Michael Dietz, and Dan S. Wallach, *Rice University*

DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis 569
Lok Kwong Yan, *Syracuse University and Air Force Research Laboratory*; Heng Yin, *Syracuse University*

(Friday, August 10, continues on p. vi)

Friday, August 10 (continued)

Software Security II

- STING: Finding Name Resolution Vulnerabilities in Programs 585
Hayawardh Vijayakumar, Joshua Schiffman, and Trent Jaeger, *The Pennsylvania State University*
- Tracking Rootkit Footprints with a Practical Memory Analysis System 601
Weidong Cui and Marcus Peinado, *Microsoft Research*; Zhilei Xu, *Massachusetts Institute of Technology*;
Ellick Chan, *University of Illinois at Urbana-Champaign*
- TACHYON: Tandem Execution for Efficient Live Patch Testing 617
Matthew Maurer and David Brumley, *Carnegie Mellon University*

Being Social

- Privacy-Preserving Social Plugins 631
Georgios Kontaxis, Michalis Polychronakis, and Angelos D. Keromytis, *Columbia University*; Evangelos P.
Markatos, *FORTH-ICS*
- Social Networking with Frienteegrity: Privacy and Integrity with an Untrusted Provider 647
Ariel J. Feldman, Aaron Blankstein, Michael J. Freedman, and Edward W. Felten, *Princeton University*
- Efficient and Scalable Software Detection in Online Social Networks 663
Md Sazzadur Rahman, Ting-Kai Huang, Harsha V. Madhyastha, and Michalis Faloutsos, *University of
California, Riverside*

Message from the USENIX Security '12 Program Chair

It is my great pleasure to welcome you to the 21st USENIX Security Symposium. We have an outstanding event in store for you, and for that, I thank all of you—the authors, the invited speakers, the program committee members and other organizers, the external reviewers, the sponsors, the USENIX staff, and the attendees. The USENIX Security Symposium would not be the premier venue that it is if it were not for your involvement.

This year USENIX Security received 222 submissions. As in previous years, the program committee used a multi-round reviewing process. The authors of submissions were not revealed to the reviewers, and every paper was reviewed by at least two reviewers. Papers that received a positive score in the first round were reviewed by at least two additional reviewers. The program committee met to discuss the submissions on April 19 and 20 at the Microsoft Research campus in Redmond, Washington. Jaeyeon Jung at Microsoft Research devoted a huge amount of time to ensure that all aspects of the meeting ran smoothly; I am deeply grateful to her for all of her hard work as host. I would also like to thank Microsoft Research and USENIX for funding the meals during the PC meeting.

After very careful and extensive deliberations, the program committee decided to accept or conditionally accept 43 papers—a record for USENIX Security. The quality of these papers is very high—a testimony to the strength of our community!

The entire program committee invested a tremendous effort in reviewing and discussing these papers. Please join me in thanking the program committee and all the external reviewers, listed on page ii, for their countless hours of work. I would also like to thank Will Enck, Kevin Fu, and Sam King for serving as deputy chairs and handling the submissions for which I had a conflict.

We also have a wonderful selection of invited talks for you. I would like to thank the invited talks committee—David Evans, Casey Henderson, David Molnar, Bruce Potter, and Margo Seltzer—for all of the hard work they invested toward ensuring an exciting, interesting, educational, and invigorating invited talks track. The Poster and Rump Sessions have also been hits at previous USENIX Security Symposiums, and I think you will find them to be “can’t-miss” events at this year’s USENIX Security too. I would like to thank Matt Bishop for serving as this year’s Poster Session Chair, and Matt Blaze for serving as this year’s Rump Session Chair.

I am also deeply grateful to the entire staff at USENIX. They have worked incredibly hard to help make USENIX Security one of the top conferences in the field. Please join me in thanking them. Please also join me in thanking Joseph Schwartz for capturing USENIX Security on video.

Finally, I would like to thank all of the authors who submitted their research papers, posters, and Rump Session talks.

Welcome to Bellevue, Washington, for the 21st USENIX Security Symposium! I hope you enjoy the conference.

Tadayoshi Kohno, University of Washington
USENIX Security '12 Program Chair

PharmaLeaks: Understanding the Business of Online Pharmaceutical Affiliate Programs

Damon McCoy[◇] Andreas Pitsillidis^{*} Grant Jordan^{*} Nicholas Weaver^{*†} Christian Kreibich^{*†}
Brian Krebs[‡] Geoffrey M. Voelker^{*} Stefan Savage^{*} Kirill Levchenko^{*}

[◇]*Department of Computer Science
George Mason University*

^{*}*Department of Computer Science and Engineering
University of California, San Diego*

[†]*International Computer Science Institute
Berkeley, CA*

[‡]*KrebsOnSecurity.com*

Abstract

Online sales of counterfeit or unauthorized products drive a robust underground advertising industry that includes email spam, “black hat” search engine optimization, forum abuse and so on. Virtually everyone has encountered enticements to purchase drugs, prescription-free, from an online “Canadian Pharmacy.” However, even though such sites are clearly economically motivated, the shape of the underlying business enterprise is not well understood precisely because it is “underground.” In this paper we exploit a rare opportunity to view three such organizations—the GlavMed, SpamIt and RX-Promotion pharmaceutical affiliate programs—from the inside. Using “ground truth” data sets including four years of raw transaction logs covering over \$185 million in sales, we provide an in-depth empirical analysis of worldwide consumer demand, the key role of independent third-party advertisers, and a detailed cost accounting of the overall business model.

1 Introduction

Much like the legitimate Internet economy, advertising is a major driver for the “underground” criminal economy as well. For all their variety, spam, search-engine abuse, forum spam and social spam—as well as the botnets, fast-flux networks and other technical infrastructure that enable these activities—are all simply low-cost advertising platforms that monetize latent consumer demand. Consequently, an emerging research agenda has developed around understanding the economic structure of these businesses, both to understand the scope and drivers for the problem [8, 9, 13], as well as to help prioritize interventions [14, 15]. Unfortunately, while clever inference and estimation techniques can illuminate a few of the key questions, much remains unclear. This is because, as a rule, there is little “ground truth” data in the field for either validating such results or to provide finer-grained analytics that can be obtained via inference.

This paper provides a rare counter-point to this rule. Under a variety of serendipitous circumstances (largely

driven by competition between criminal organizations), a broad corpus of ground truth data has become available. In particular, in this paper we analyze the content and implications of low-level databases and transactional metadata describing years of activity at the GlavMed, SpamIt and RX-Promotion pharmaceutical affiliate programs. By examining hundreds of thousands of orders, comprising a settled revenue totaling over US\$185M, we are able to provide comprehensive documentation on three key aspects of underground advertising activity:

Customers. We provide detailed analysis on the consumer demand for Internet-advertised counterfeit pharmaceuticals, covering customer demographics, product selection (including an examination of drug abuse as a driver), reorder rates and market saturation.

Advertisers. We quantitatively detail the role of third-party affiliate advertisers (both email/forum spammers and SEO-based advertisers), the dynamics of their labor market, their ability to drive revenue and the distribution of their commission income. This analysis includes the operators of many of the best-known botnets including MegaD, Grum, Rustock and Storm, and we document individual advertisers generating over \$10M in sales.

Sponsors. We derive an empirical revenue and cost model, including both direct costs (sales commissions, supply, payment processing) and indirect costs (hosting, domain registration, program advertisements). We also provide insight and validation about the most significant overheads for the operators of such programs.

This is an unusual research paper. We introduce no new artifact, we develop no new inference technique, we deploy no new measurement infrastructure. We do none of these things because we don’t need to; we have the actual data sets that we would otherwise try to measure, infer or estimate. Thus, while there are significant methodological challenges that we must overcome (mainly around the forensic reverse engineering of database schemas and their semantics), ultimately the contribution of this paper is in its results. However, we believe these are both unique and significant, with implications for best addressing this variety of Internet abuse.

2 Background

Abusive Internet advertising has existed virtually as long as the Internet itself. In addition to well-defined advertising channels such as sponsored search [11, 12], rogue advertisers make use of a broad range of vectors to attract customer traffic including email spam [1, 6, 14, 17], search engine manipulation [7, 13, 23], forums and blog spam [19, 24] as well as online social networks [4, 22]. Due to pressure against these tactics, few legitimate merchants will engage such advertisers and thus rogue advertising and rogue products tend to go hand in hand. For example, in one recent report on email spam, Symantec estimated that 80% of all such messages shilled for “prescription-free” pharmaceuticals [21].

However, the structure of this activity has changed significantly over the last decade. In particular, market specialization has largely eliminated the independent “soup-to-nuts” advertiser who previously handled the entirety of the sale process [16]. Instead the rise of the affiliate program, or “partnerka”, model has separated the role of the advertiser, paid on commission to attract customer traffic, from the sponsor who in turn handles Web site design, payment processing, customer service and fulfillment [18]. This evolution is not unique to abusive advertising; indeed, large legitimate merchants such as Amazon also sponsor affiliate programs as a means of advertising. However, it has been deeply internalized within the underground ecosystem including the pay-per-install [3], FakeAV [20], pornography [25], pharmaceuticals [2], herbal supplements [14], replica [14] and counterfeit software markets [9], among others.

Counterfeit pharmaceuticals represent a typical example. Here a range of sponsoring affiliate programs provide drugstore storefronts, drug fulfillment (typically via drop shipping from India), payment processing, customer service and so on. Independent advertisers, or *affiliates*, in turn promote the program (e.g., by using botnets to send spam email or manipulating search engine results) and are paid a commission on each sale that results from a click on one of their ads. Commissions range from 30%–40% of gross revenue, typically paid via a quasi-anonymous online money transfer service such as WebMoney or Liberty Reserve.

This business model has two key advantages for the advertiser: focus and mobility. Without needing to attend to issues such as Web site design, payment processing, customer service, fulfillment and so on, the advertiser is free to focus single-mindedly on the task of attracting customer traffic to these sites. Indeed, this functional specialization has supported the creation of ever more sophisticated botnets for email delivery or “black hat” search engine optimization, and many of the largest botnets are directly involved in advertising the programs in this paper (Rustock, MegaD, Grum, Cut-

wail, Storm, Waledac and others). The second advantage of this model, mobility, is that the loosely coupled nature of their relationship with affiliate programs allows an advertiser to switch programs at will (or even support multiple programs at once). This low “switching cost” provides bargaining power for the effective advertiser (indeed, we witness high-sales advertisers able to use this threat to drive higher commissions). More importantly, it reduces an advertiser’s exposure to business continuity risk. If a particular affiliate program should shut down, advertisers can still monetize their investments (e.g., in a botnet) by advertising for a different sponsor.

However, the benefits of this separation are strong for the sponsoring affiliate program as well. By outsourcing advertising they free themselves from direct exposure to the criminal risks associated with large-scale advertising enterprises (e.g., mass compromise of computers and online accounts). Second, because advertisers are paid on a commission basis, they also outsource “innovation risk”. Program sponsors need not predict the best way to attract customer traffic at a given point in time. Instead hundreds of advertisers innovate independently; if many of them fail, so be it. Since advertisers are only paid commissions on successful sales, a sponsor will only end up paying for effective advertising strategies and need not distinguish among strategies *a priori*.

Against this background, online pharmaceutical sales is one of the oldest and largest affiliate program markets. This market supports tens of affiliate programs and, as we will see, thousands of independent advertisers (*affiliates*) and hundreds of thousands of customers. However, while the mechanics of this business model are well-described in recent work [2, 14, 18], the dynamics of the actors and the underlying constants that define the cost structure (and hence the vulnerabilities in the business) are not well understood at all. Indeed, even simple questions such as “How big is sales turnover?” are imperfectly understood. For example, Kanich *et al.* used one method to estimate that the combined turnover across seven leading pharmacy programs (constituting two-thirds of affiliate brands advertised in spam) is roughly 86,000 orders per month [9]. However, Leontiadis *et al.* use a different technique to arrive at a much larger estimate suggesting over 640,000 orders per month [13].

In this paper, we answer this and many other such questions *precisely* by focusing in depth on three pharmaceutical affiliate programs: GlavMed, SpamIt and RX-Promotion. These organizations have been in business for five years or more. Together, they represent many tens of storefront “brands” (including the ubiquitous “Canadian Pharmacy”) and, according to the data from our prior measurement studies, these programs have been advertised in over a third of all spam email messages [14].

3 Authenticity and Ethics

Our use of “found data” creates two new concerns that we address here: authenticity and ethics.

First, it is useful to provide some rough context concerning the circumstances leading to the release of these data sets. As explained in the previous section, GlavMed and RX-Promotion are both long-operating pharmaceutical affiliate programs based in Russia. However, for a variety of reasons, enmity developed between owners in each program, revealed anecdotally through “sniping” on underground forums, claims of denial-of-service attacks and ultimately to the hacking of each other’s infrastructure sites. Perhaps inspired by the “online leak” meme, popularized recently by Wikileaks and others, elements of these two organizations (or parties sympathetic to their positions) gained access to information about each other’s operations and then made portions of this data available: sometimes publishing very broadly on underground forums and file-sharing sites, and other times distributing to a variety of journalists, e-crime researchers, law enforcement agencies as well as a broad range of underground actors.

Through these channels we obtained access to three transactional data sets: the complete dump, covering four years, of the GlavMed and SpamIt back-end database (comprising transactions, payments and so on) and a year of more restricted transactional data for the RX-Promotion program. We also received two metadata corpuses: detailed archived chat logs from the program operator for sites operated by GlavMed and SpamIt, as well as financial data concerning the revenue and cost structure for the RX-Promotion program. For further context and back-story about this data, we refer readers to the “Pharma Wars” series by Brian Krebs [10].

3.1 Authenticity

Given that we did not gather the information ourselves *and* the adversarial nature by which the data became available, an obvious question is how to evaluate its accuracy and authenticity: how do we know that our sources did not fake the data?

While we cannot establish clear provenance beyond all possible doubt, we observe a range of strong supporting evidence. First, we observe that the data sets are large and detailed (over 2M sales records, with over 140 linked tables, coupled with several GB of related metadata). These attributes do not entirely discount the possibility that they could be grossly fraudulent, but it suggests that the costs of creating such a forgery would be significant.

Second, we consider questions of internal and cross-consistency. The transactional data sets have complex schemas (covering orders, potentially many payment

transactions per order, commissions to advertisers, subsequent payouts, and so on) and we find direct concordances between the different elements (e.g., if we sum the settled sales for a particular affiliate it typically relates directly to the size of the payout to that affiliate). We also find concordances *between* the transactional data and the metadata. For example, we found multiple chat logs directing a GlavMed/SpamIt employee to make a payment to a particular affiliate that is then matched by an identical payout record in the associated transactional database. Similarly, the monthly revenue for shipped products for RX-Promotion is consistent with the settled revenue from its payment processor in the same period. Finally, during the period covered by all three transactional data sets we had placed multiple product orders from each of the associated programs [9, 14]. We find each and every one of our orders in the appropriate database with the correct data.

While this evidence cannot comprehensively prove the absence of fraud,¹ given the strong concordances and the absence of any evidence supporting the forgery hypothesis, we believe the greater likelihood is that these data sets are authentic and accurate. We proceed with this assumption going forward.

3.2 Ethics

The other fundamental issue concerns the ethics of using data that was, in all likelihood, gathered via illegal means. Here there are two kinds of questions. The first is a high-level question concerning whether the nature of how the data was originally gathered should *prima facie* proscribe all subsequent uses of it. This question is not new and it manifests in a range of fields. For example, should a political scientist be proscribed from analyzing the contents of the Pentagon papers (or the more contemporary Wikileaks data) in reasoning about U.S. foreign policy? Similarly, should researchers avoid using widely publicized stolen password data (e.g., from the Anonymous/Lulzsec leaks) when studying the strength of user-selected passwords? We justify our own choice to take such steps by reasoning about harm.

We observe that this data is already broadly available and the knowledge of its existence, its association with the GlavMed, SpamIt and RX-Promotion organizations, and some of the over-arching contents (e.g., total revenue, etc.) have already been widely and publicly documented. Consequently, we cannot create any new harm simply through association with these entities or repeating these findings.

To manage any remaining harms we institute a number

¹For example, while we believe comprehensive forgery would have been cost prohibitive given the size and richness of these data sets, a forger might have selectively altered only certain records and updated dependent schemas to be consistent.

Program	Period	Affiliates	Customers	Billed orders	Revenue
GlavMed	Jan 2007 – Apr 2010	1,759	584,199	699,516	\$81M
SpamIt	Jun 2007 – Apr 2010	484	535,365	704,169	\$92M
RX-Promotion	Oct 2009 – Dec 2010	415	59,769 – 69,446	71,294	\$12M

Table 1: Summary of the affiliate program data used in the analysis. Orders are rounded to the nearest thousand, revenue to the nearest million U.S. Dollars. Affiliates and customers are listed *after* de-duplication and billed orders and revenue reflect only those orders whose payment transactions completed (both processes are described in Section 4.1).

of controls in our work focused on the individual stakeholders. First and foremost, and in accordance with our institution’s human subjects review process, we protect customer confidentiality since, of all parties described in the data, they are most vulnerable. To this end, we committed to modify the raw data sets to anonymize personally identifiable customer data such as their name, address and the PAN component of their credit card information (though in a way that we are able to associate multiple orders from the same customer). For the remaining stakeholders, program employees, affiliates, suppliers and payment processors, we use a similar standard in publishing our work. In each of these cases the persons or organizations operate using handles or code names that are not clearly identifiable (e.g., “brainstorm” or “gl”) without the use of additional data sources. In some cases (e.g., payment processors, suppliers) we have become aware of the likely true names of these organizations (typically through reading the metadata) but we restrict ourselves to using these non-identifiable code names since the true names do not enhance our analysis. We do not name program employees and we typically discuss affiliates in aggregate, with an exception being the top affiliates whom we distinguish in this paper using only their online handles.

4 Derived Data

Using “found data” also introduces a range of methodological challenges, ranging from reverse engineering schemas to resolving ambiguities in the data. In this section we describe the data sets (summarized in Table 1) and explain how we derived the additional contextual relations used in our analysis.

4.1 GlavMed and SpamIt

The first two data sets are PostgreSQL database dumps of the operational databases for the GlavMed and SpamIt programs, including all schemas, data, and trigger functions, but no other code external to the database. The GlavMed database begins November 2005 and ends early May 2010, of which we use the period spanning all of 2007–2009 and the first four months of 2010.²

²Since our goal is accuracy and not completeness, we purposely exclude the first 14 months of the data set because it is both “poisoned”

GlavMed and SpamIt are sister programs run by the same organization and, indeed, both use the same database schema. In fact, it appears that SpamIt was “forked” from the GlavMed database on June 19, 2007: all records before that date are identical in both databases, while records after that date are distinct. Leaked chat logs of the program operators suggest that this split was related to the owner’s contemporaneous acquisition of Spamdott.biz, a popular closed spammer forum of that period. In part through this forum, the SpamIt program nominally catered to a select group of affiliates relying on email and other forms of spam, while GlavMed remained open to a broader range of advertisers who primarily advertised via search engine optimization techniques.³

A detailed description of the data and its associated schema, consisting of over 140 tables in each database, is outside the scope of this paper. However, we perform most of our analysis using five tables: `shop_sales` describing each order, `shop_transactions` recording attempts to bill (or refund) the order via a payment service provider, `shop_customers` recording customer information, `shop_affiliates` recording information about each affiliate, and `shop_affiliates_income_2` recording affiliate commissions for each sale. We also relied on instant message chat logs of the operators of GlavMed and SpamIt to aid our understanding and validate our hypotheses about the meaning and use of various tables.

However, the GlavMed and SpamIt databases are fundamentally operational in nature, and not naturally designed for the kind of broad analysis that are the goal of this paper. Thus, we now describe the additional data processing required to produce necessary relations (e.g., such as identifying unique customers).

4.1.1 Customers

In an ideal world, each customer record would represent a unique customer and include accurate demographic information for our analysis (age, sex, and either country or U.S. ZIP code). The reality, hardly unique to our data set, is less obliging: In addition to many test accounts

with transactions for other kinds of products, including \$500k in counterfeit software sales, and makes inconsistent use of the database schemas that become standard in the later portion of the date range.

³This distinction is not absolute, however; domains advertised by GlavMed affiliates have appeared in email spam.

used by the store operators, a large number of customer records are generated by irate users venting their frustration with the deluge of spam advertising the program.⁴ Thus, for the purpose of this study, we consider only customers who have successfully placed an order (more specifically, those whose credit card or other payment mechanism was successfully billed, as described later), which reduces the number of customer records by 21% in the GlavMed data set (from 875,457 to 690,590) and 39% in the SpamIt data set (from 1,145,521 to 693,319), the latter clearly attracting more abuse.

De-duplication. An additional problem is that, unless the customer uses a previously assigned customer number to explicitly log in, each repeat order would result in a new customer record. To identify repeat customers, we de-duplicate the remaining customer records by coalescing those whose name, billing address and email address are identical, reducing the number of unique customers to 584,199 in GlavMed and 535,365 in SpamIt. For address matching, we used the common Visa/MasterCard Address Verification System (AVS) predicate, which relies on street number and ZIP code only. Both names and email address matches were case insensitive, and we allowed first and last names to be transposed.

Demographics. Our analysis relies on customer demographic data consisting of the customer's country or U.S. ZIP code, as well as their self-reported age and sex. The country and ZIP code are necessary for proper order fulfillment, and therefore are generally reliable. However, customers optionally provide age and sex data when ordering, so it is not always present and it is subject to misreporting. Only 41% of GlavMed orders and 38% of SpamIt orders included this information, and we cannot validate it since customers could easily dissemble. Indeed, we found that a larger than expected number of users reported birth dates of January 1, February 2, and so on (these being some of the easiest dates to report via the interface). However, these anomalies are a small minority and we proceed under the assumption that the data is generally correct (eliminating these cases does not substantively change the results reported in Section 5.1.3).

4.1.2 Affiliates

As with customers, affiliate records also require de-duplication. However, here the duplication is not a mere artifact of the interface, but is frequently an intentional action. Affiliates frequently register under multiple identities, either to modulate their perceived earnings (affiliate programs commonly provide “top” lists showing the affiliates with the highest earned commissions) or to gain

⁴This frustration was well captured by the many regular expressions in the operators' customer blacklist, e.g., `(.*)SP(A+)M(.*)` and `(.*)F(U+)CK(.*)`.

access to additional referral commissions that are provided on sales generated by new affiliates referred into the program.⁵ To address these issues, we de-duplicate affiliates as follows. For all affiliates with over \$200 in revenue we link those who share an email address, ICQ number⁶ or “identified commission payments”. We considered a commission payment to be identified if it represents over 75% of an affiliate's revenue and includes unique payment account information (such as a WebMoney, Fethard Finance, or ePassporte account or an identified GlavMed payment card). The notion of identified payments was necessary to avoid incorrectly associating affiliates who use the commission payments system to pay third parties (e.g., by asking for small payouts to a third-party WebMoney purse).

4.1.3 Transaction Outcomes

In the GlavMed and SpamIt data sets, each customer sales record in turn drives the creation of one or more transaction records which reflect an attempt to transfer money to or from a customer (as identified by a credit card or Automated Clearing House (ACH) identifier) via a third-party payment service provider. When a transaction is successful the `response_status` field in this record is zero (we validated these semantics by examining both raw payment processing error messages and associated SQL triggers in the databases).

However, for a host of reasons transactions are frequently declined. Indeed, over 25% of all transaction attempts decline in both the GlavMed and SpamIt data sets. In these cases, new transactions may be generated, possibly using different payment service providers. In some cases, large order amounts are billed into two smaller transactions. Overall, 91% of sales are able to complete a payment transaction.

Finally, a transaction may be refunded, either partially or fully. An additional complexity arises from currency conversion because customer payments are internally valued in U.S. Dollars, but can arrive in Euros, Pounds and several other currencies. When refunds arrive in native currency, we locate the original transaction and calculate the dollar refund value on a pro-rated basis against the original value in the native currency. All revenue numbers reported in the analysis refer to the total amount billed, before any refunds against the transaction. Refunds are shown separately in Table 3.

Note that having this ground truth data allows us to calibrate biases in previous methods for estimating revenue. In particular, we revisit our “purchase pair” tech-

⁵As an incentive to attract affiliates, program sponsors will typically offer their affiliates a 5% commission on the future sales of any new affiliate they bring into the program.

⁶ICQ is one of the oldest widely-deployed IM chat systems, and is very popular in Russia and CIS states.

nique that infers order turnover via customer order number advancement and then conservatively estimates the average order size to gauge overall revenue [9]. Across four years, we find that a significant number of order numbers never appear in the database due to either filtering for customer fraud or shopping cart abandonment (between 13–28% for SpamIt and 7–17% for GlavMed). The lower number of absent orders for GlavMed is likely because the search engine vector used by its affiliates generates less antipathy among consumers. In both cases, 8–12% of the orders that do appear in the database are ultimately declined and do not ship. Consequently, true turnover is between 8% (low of GlavMed) and 35% (high of SpamIt) less than predicted by the “purchase pair” technique. However, since the average successful order size is between \$115 (GlavMed) and \$135 (SpamIt), revenue estimates based on an average sale of \$100 are roughly in-line with true revenue (within 6% overall for GlavMed and 13% overall for SpamIt).

4.2 RX-Promotion

Our third data set concerning transactions from the RX-Promotion program is far more limited. It only covers a single year of data from January to December of 2010, consisting of a single extracted view summarizing each sale during the period *made by U.S. customers*. In addition, roughly one week of data is missing (around the last week of April 2010). Consequently, this transactional data will strictly understate the turnover from RX-Promotion.⁷

Each sales record includes information about the customer (name only), the status of the order, its contents, the total price as well the amount paid to the supplier, shipper and the affiliate who generated the sale. Our analysis includes only orders with the status value “shipped”, which make up 77% of all sales records (“declined” was the next largest category at 14%).

Since the RX-Promotion data set does not include crisp customer identifiers, we use two approximations for identifying multiple orders belonging to the same customer. The conservative approximation of 69,446 customers only links sales records together if a customer explicitly logs into the site using a previously assigned customer ID. However, we note that this measure strictly overestimates the number of customers since many users prefer to place subsequent orders by entering in their information again. Alternatively, one can group customers that share the same first and last name (normalized for

⁷Based on our measurements of both the GlavMed and SpamIt data sets, our own previous study of the Eva Pharmacy program [9], and inference from the RX-Promotion metadata, we are confident that U.S. customers represent between 75% and 85% of total turnover. In addition, the missing week of data from April should cause our data to underestimate annual orders by an additional 2%.

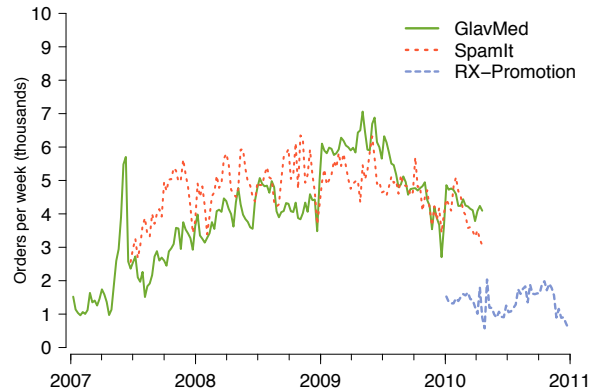


Figure 1: Weekly sales volume for each of the programs.

capitalization), resulting in 59,769 customers. This approach will accurately capture multiple orders from the same user, but at the expense of potentially aliasing users who happen to share the same first and last names. Thus, the true number of unique customers is likely between the two estimates, but to avoid aliasing issues we use the larger conservative estimate in our analyses.

Finally, we also make use of seven months of overlapping metadata that includes detailed spreadsheets accounting for month-by-month costs and cash flow. This data does not have any of the previous limitations and captures the financial performance of the program precisely and in its entirety.

5 Analysis

Using these data sets, we now provide a detailed assessment of the affiliate program business model. From the standpoint of the program sponsor, we consider four key aspects of the business enterprise in turn: customers, affiliate advertisers, costs and payment processing.

5.1 Customers

Neither online pharmacies nor their advertisers generate capital on their own. These activities thrive only because they exploit latent customer demand for the products on offer. It is this customer purchasing that drives the entire ecosystem and thus this is where we begin: how many purchases, for what, by whom and, perhaps, why?

Overall, as shown in Table 1, 584,199 unique customers placed orders via GlavMed during the measurement period and 535,365 placed orders via SpamIt; of these approximately 130K appear in both. RX-Promotion is a smaller program and covers a shorter time period, with somewhere between 59,769 and 69,446 distinct customers placing orders. In turn these customers generated almost 1.5M orders, varying from week to week as shown in Figure 1. Note that the spike in May 2007 for GlavMed is an artifact corresponding to the short period after GlavMed had purchased SpamIt, but before they

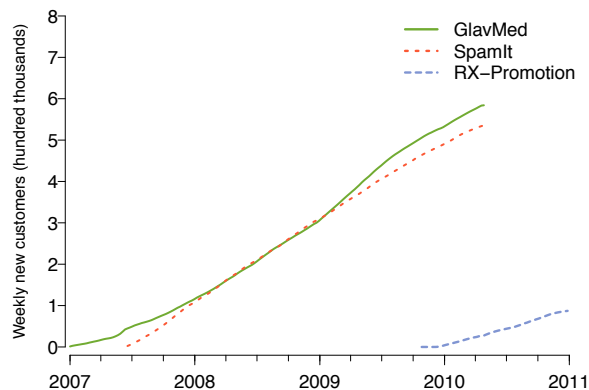


Figure 2: Cumulative number of new customers.

had forked the databases in June 2007 (Section 4.1). After the fork, GlavMed has very steady growth in orders until mid-2009, even surpassing SpamIt, and then starts to decline. Orders to SpamIt plateau for 2008–2009, similarly declining in mid-2009.⁸ RX-Promotion order volumes are considerably more dynamic, for reasons we will explain later, with totals varying between 1–2 thousand per week across the year of data.

5.1.1 First-time Customers

However, these million plus customers and their purchases do not necessarily constitute the entirety of this market, but only the *portion* that has been serviced to date by these particular programs. This raises the question: How saturated is the market for counterfeit pharmaceuticals? To evaluate this, Figure 2 shows the cumulative number of unique customers seen in each program per week over the measurement period. Thus, changes in slope indicate changes in the rate of new customer acquisition. From these trends it is clear that the affiliate programs are attracting new customers at a steady rate over time, and that the market *does not appear to be saturating at all*. In particular, sister programs GlavMed and SpamIt attract new customers at nearly the same rate (3,367/week and 3,569/week on average) while RX-Promotion, a smaller program, attracts customers at a slower, but still constant rate (1,429/week on average). The stability of this growth over time provides some explanation for why spammers continue to blast email indiscriminately to all Internet users over time: they are still mining a rich vein of latent customer demand.

⁸This decline undoubtedly has many roots including increasing pressure that mounted on SpamIt due to its high visibility (e.g., the principal owner of SpamIt was identified by Russian Newsweek as the World’s Biggest Spammer), shutdowns of large botnets operating as affiliates (e.g., the MegaD botnet, which we observed spamming for sites associated with SpamIt affiliate “docent”, ceased operating in November of 2009), and inter-program competition (e.g., starting in 2010, we see a roughly 15% reduction in the number of active affiliates in the SpamIt program and we witness one large affiliate, “anonymouse”, leaving SpamIt and moving to RX-Promotion during this period).

5.1.2 Repeat Customers

New customers, however, are not the whole story. The graphs in Figure 3 show total program revenue per week broken down into two components: revenues from first-time customers and revenue from repeat orders from existing customers. What we see is that repeat orders are an important part of the business, constituting 27% and 38% of average program revenue for GlavMed and SpamIt, respectively. For RX-Promotion revenue from repeat orders is between 9% and 23% of overall revenue.

Overall, revenue from repeat customers steadily increases over the years for GlavMed and SpamIt, and holds steady even when orders and overall revenue decline in mid-2009. The situation is more dynamic for RX-Promotion with a pronounced dip in program revenue in the middle of 2010 that impacts new and repeat customers both. This dip corresponds to the period when RX-Promotion lost its payment processing services for scheduled drugs.⁹ Indeed, if we only consider the period after August 2nd, repeat order revenue averages between 12% and 32%.

This data highlights a counterpoint to the conventional wisdom that online pharmacies are pure scams: simply taking credit cards and either never providing goods or providing goods of no quality. Were this hypothesis true, we would not expect to see repeat purchases—clear signs of customer satisfaction—in such numbers. Anecdotally, we have placed several hundred such orders ourselves and, while we cannot speak to the quality of the products we received, we have almost always received a product in return for our payment [9, 14].

5.1.3 Product Demand

Beyond measuring overall demand, we are particularly interested in determining what makes up this demand: which drugs are being purchased, and does this provide clues about *why* this market is preferred.

In an effort to reach all customer niches, each of the programs carries thousands of products. To reason about this multitude of drugs, we classified the bulk of the products into broad categories based on our best assessment (necessarily subjective) of the drug’s use: erectile dysfunction, pain/inflammation, male enhancement (not ED), mental health, sleep, obesity and other.

Using this classification, customer demand for specific kinds of drugs in the different programs is striking. As with the previous time series graphs, Figure 4 shows weekly revenue for the three affiliate programs over time,

⁹Associated metadata suggests that RX-Promotion’s payment service provider (PSP) had arranged for merchant accounts at an Icelandic bank to be used for RX-Promotion controlled drug payments. However, on May 10th 2010, a complaint by Visa caused the bank to shut down these accounts and thus processing for controlled substances was curtailed until August 2nd when the PSP established new accounts for this purpose with Azeri banks.

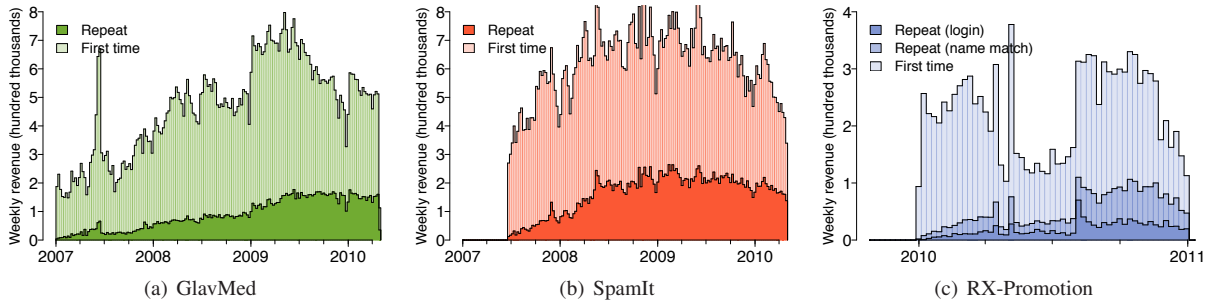


Figure 3: Weekly order revenue shown by customer class.

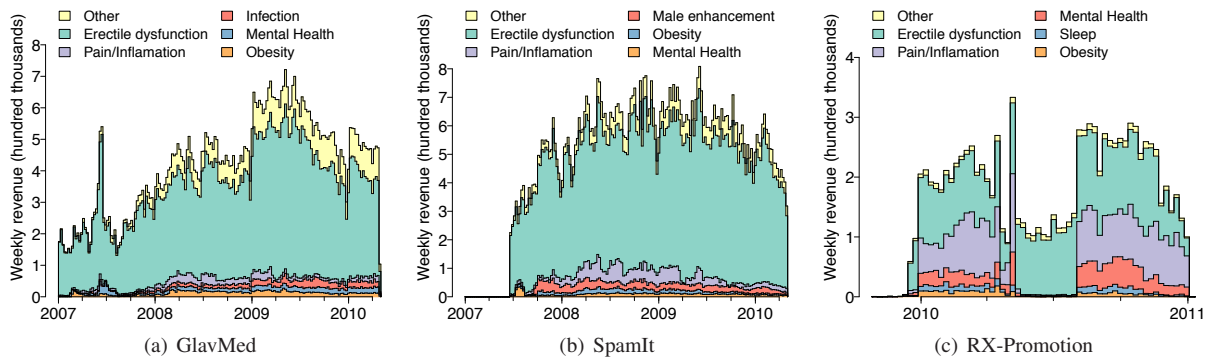


Figure 4: Weekly order revenue shown by drug type.

but here each of the top five revenue-earning drug categories is colored distinctly. For GlavMed and SpamIt, the jokes about spam are spot on: “erectile dysfunction” (ED) purchases dominate their revenue. Customers do purchase other notable drugs, but they represent a small fraction of revenue over time for these programs.

In contrast, revenue from pain/inflammation orders matches revenue from ED in RX-Promotion. RX-Promotion has a markedly different formulary from GlavMed and SpamIt, prominently offering products that GlavMed and SpamIt do not sell. Specifically, these include scheduled drugs for pain (Oxycodone, Hydrocodone, Vicodin, etc.), mental health (Adderal, Ritalin, etc.), and sleep (Valium, etc.), all of which have high abuse potential.¹⁰

These examples suggest that there may in fact be a range of distinct reasons *why* different drugs are popular via this medium. Table 2 summarizes order volume and program revenue for different groups of drugs sold to customers by the three affiliate programs. Here we merge our original set of categories into three groups that correspond to different customer motivations for purchasing drugs. The first group includes erectile dysfunction (ED), male enhancement, and related products (including fakes such as “Herbal Viagra”). These drugs, some-

times called “lifestyle” drugs, do not address chronic or acute illness. While they are relatively easy to obtain under prescription, seekers may prefer the online channel for reasons of embarrassment or price.¹¹ The second group includes drugs that have the potential to be seriously abused, and includes addictive drugs such as opiates, depressants, stimulants, etc. For many of these drugs, customers run substantial legal risk in purchasing them without prescription, and presumably run this risk because of a strong desire or need. The third group includes drugs for treating chronic or acute illnesses. Since these drugs carry no strong abuse risk, nor represent a clear cause for social discomfort, we presume that their purchase is motivated by economics: lower direct drug costs (which can be substantial) and the absence of indirect costs (for a doctor’s visit). In each category, the table also lists the top categories or specific products.

Reflecting Figure 4, the ED group dominates items ordered and revenue to the program, particularly for GlavMed and SpamIt. For RX-Promotion, though, drugs with the potential for abuse are high-revenue orders. Although they comprise just 14% of orders for

¹⁰The Controlled Substances Act in the U.S. defines five drug “schedules”, or classifications, according to various criteria such as potential for abuse. Scheduled drugs require prescriptions and have heavy financial and/or criminal penalties for illegal sale.

¹¹The per-item drug price offered by such programs is frequently less than 20% of that offered by legitimate retailers. For example, the median price for 10 tablets of 100mg Sildenafil Citrate was \$42.57 on GlavMed and \$23.40 at RX-Promotion. By contrast, according to data at drugs.com, legitimate brand Viagra in the same amount sells for \$193.99. Note that these prices do not account for shipping, which can add \$15 to \$30 per order.

Product	GlavMed		SpamIt		RX-Promotion	
	Orders	Revenue	Orders	Revenue	Orders	Revenue
<i>ED and Related</i>	580K (73%)	\$55M (75%)	670K (79%)	\$70M (82%)	58K (72%)	\$5.3M (51%)
Viagra	300K (38%)	\$28M (38%)	290K (34%)	\$31M (36%)	33K (41%)	\$2.7M (27%)
Cialis	180K (23%)	\$19M (26%)	190K (22%)	\$23M (27%)	18K (22%)	\$1.9M (19%)
Combo Packs	49K (6.1%)	\$3.9M (5.4%)	110K (14%)	\$8.4M (9.8%)	5100 (6.4%)	\$350K (3.4%)
Levitra	32K (4.1%)	\$3.2M (4.4%)	35K (4.2%)	\$3.9M (4.5%)	1200 (1.5%)	\$150K (1.5%)
<i>Abuse Potential</i>	48K (6.1%)	\$4.5M (6.1%)	64K (7.6%)	\$6.2M (7.3%)	11K (14%)	\$3.3M (32%)
<i>Painkillers</i>	29K (3.7%)	\$2.4M (3.3%)	53K (6.3%)	\$4.7M (5.5%)	10K (13%)	\$3.0M (29%)
<i>Opiates</i>	—	—	—	—	8000 (10%)	\$2.7M (26%)
Soma/Ultram/Tramadol	20K (2.5%)	\$1.8M (2.4%)	46K (5.5%)	\$4.1M (4.8%)	1000 (1.3%)	\$150K (1.5%)
<i>Chronic Conditions</i>	120K (15%)	\$9.5M (13%)	64K (7.6%)	\$5.2M (6.1%)	8500 (11%)	\$1.3M (13%)
<i>Mental Health</i>	23K (2.9%)	\$2.1M (2.9%)	16K (1.9%)	\$1.4M (1.7%)	6000 (7.4%)	\$1.1M (11%)
<i>Antibiotics</i>	25K (3.2%)	\$2.1M (2.9%)	16K (1.9%)	\$1.4M (1.6%)	1300 (1.6%)	\$97K (0.9%)
<i>Heart and Related</i>	12K (1.5%)	\$770K (1.1%)	9700 (1.2%)	\$630K (0.7%)	390 (0.5%)	\$35K (0.3%)
<i>Uncategorized</i>	48K (6.0%)	\$4.0M (5.5%)	47K (5.6%)	\$3.9M (4.6%)	2400 (3.0%)	\$430K (4.2%)

Table 2: Product popularity in each of the three programs. Product groupings and categories are in italics; individual brands are without italics. Opiates are a further subcategory of Painkillers, and include Oxycodone, Hydrocodone, Vicodin, and Percocet. Note, this table *only* describes revenue from drugs and does not capture shipping charges, which are orthogonal to drug popularity.

RX-Promotion, they account for nearly a third of program revenue, with the Schedule-II opiates—only available at RX-Promotion—accounting for a quarter of revenue. Indeed, during the period when RX-Promotion had working credit card processing for controlled meds, sales of Schedule II, III and IV drugs produced 48% of all revenue! The fact that such drugs are over-represented in repeat orders as well (roughly 50% more prevalent in both RX-Promotion and, for drugs like Soma and Tramadol, in SpamIt) reinforces the hypothesis that abuse may be a substantial driver for this component of demand.

5.1.4 Demographics

Although ED drugs account for the majority of business for affiliate programs, focusing on the remaining products reveals remarkably pronounced age and sex trends among customers.

Focusing on customers reporting age and sex information, Figure 5 shows the percentage of all items ordered as a function of age, sex, and detailed product category for GlavMed and SpamIt (excluding ED products, as they would overwhelm the graph). The left half of each graph shows results for women, and the right half shows results for men. The y-axis is the self-reported age of customers, and the x-axis is the percent of all items these customers ordered. For each age the graphs show stacked horizontal bars, with segments for the top ten *non-ED* product categories.

Both age and sex purchasing patterns emerge from this visualization. For example, male GlavMed customers in Figure 5(a) purchase male pattern baldness products (peaking between ages 20–30) and male enhancement products (peak 45–50), while women predominantly purchase obesity (peak 40–45) and reproduc-

tive health products (peak 25–30).¹² Mental health and pain/inflammation products are roughly equally popular for men and women, with an older age bias for men.

In contrast to GlavMed, just a few categories predominate for SpamIt in Figure 5(b): pain/inflammation, infection, and mental health for both men and women, male enhancement for men. Other categories more popular in GlavMed, such as acne and male pattern baldness, are smaller. One explanation is that the differences in product popularity correlates with the vector used to advertise the different affiliate programs. Since GlavMed is more likely to be involved in search engine optimization (SEO) oriented advertising, they have an opportunity to target narrower markets (e.g., by manipulating search results for keywords correlated with specific product categories). By contrast, spam is an indiscriminate advertising medium and customers clicking on spam-advertised links are predominantly taken to storefronts advertising ED products. Thus, for these customers to buy other products would require additional initiative to search within the site.

5.1.5 Geography

While both affiliate programs are located in Russia, most of their customers are not. Based on customer shipping addresses, we can determine that, across GlavMed and SpamIt programs, customers from the United States dominate at 75% of orders, with Canada, Australia, and populous countries in Western Europe following in single digits. Emphatically, Western money fuels these af-

¹²Interestingly, male customers also purchase the estrogen drug Clomid, which we have come to understand may be explained by body builders who commonly abuse the drug to counter some of the side-effects of steroids.

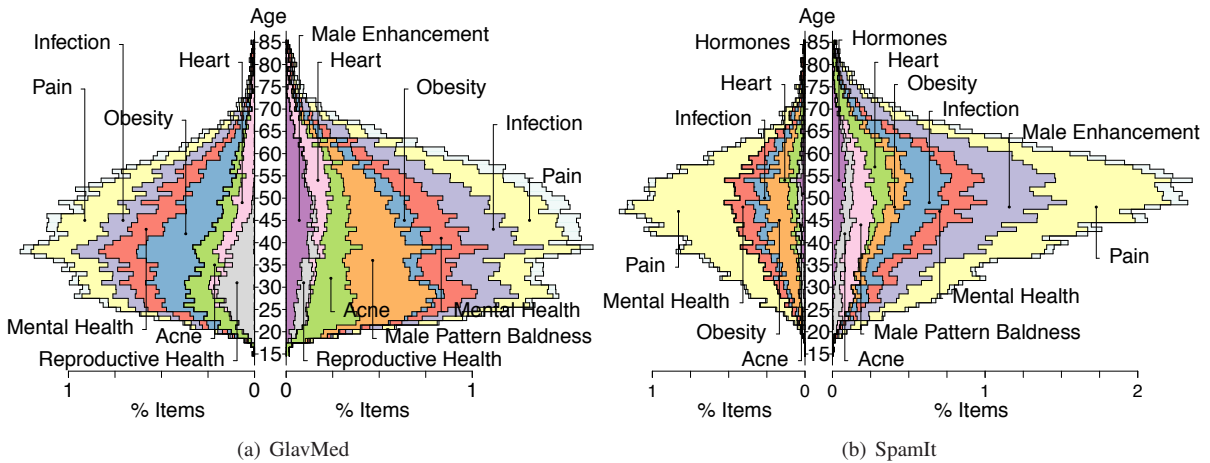


Figure 5: Items purchased separated into product category and customer age. The left half of each graph show orders from women, and the right half shows orders from men. Customers restricted to those who self-report age and sex.

affiliate programs with the U.S., Europe, Canada and Australia constituting 97% of all orders, consistent with the breakdown previously observed in [9].¹³

5.2 Affiliates

While customer purchasing drives the online pharmaceutical ecosystem, affiliates are the ones who attract and deliver the customers—and their money—to the online pharmacies. Affiliates operate by commission, receiving a significant fraction (typically 30–40%) of each customer purchase that reflects the substantial risk they assume in their aggressive advertising activities. Next we analyze the role affiliates play in making online pharmaceutical programs successful as a business.

As discussed in Section 4.1.1, we merge separate accounts in the GlavMed and SpamIt databases that belong to the same affiliate. After account merging, during the 2007–2010 measurement period 1,037 affiliates were active in GlavMed and 305 in SpamIt. Lacking detailed account profile information in RX-Promotion, we consider each account a separate affiliate. With this assumption, during the smaller one-year period for RX-Promotion 415 affiliates were active.

5.2.1 Program Revenue

GlavMed and RX-Promotion are open affiliate programs, and as such they actively advertise and recruit new affiliates to join their programs (with the public advertising focused on SEO-based advertising vectors). SpamIt, on

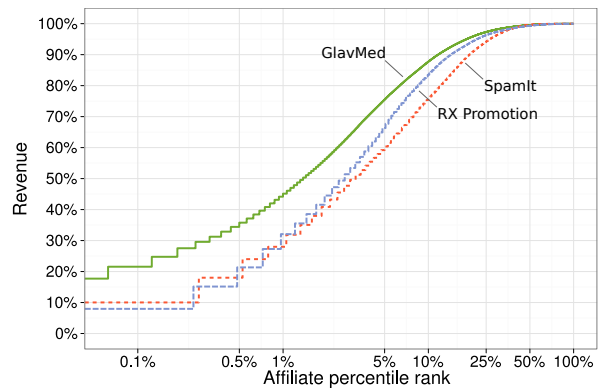


Figure 6: Distribution of affiliate contributions to total program revenue for each program.

the other hand, is a closed program—focused specifically on email spam—where affiliates join by invitation (Section 4.1). These models influence the kinds of affiliates in a program, the impact they have on generating revenue for a program, as well as the commissions they earn.

Although the programs contain hundreds to thousands of affiliates, most affiliates contribute little to the overall revenue of the programs. Figure 6 shows the CDFs of affiliate contributions to total program revenue for the three affiliate programs. The x -axis is the percent of affiliates, sorted from highest to lowest revenue they generate for the program, and the y -axis is the percent of total program revenue. The graph shows that just 10% of the highest-revenue affiliates account for 75–90% of total program revenue across the three affiliate programs; for GlavMed and RX-Promotion in particular, the remaining 90% of affiliates bring in just 10–15% of total revenue.

In the end, the most important affiliates for a program are just a small fraction of all affiliates. From a business perspective, programs can focus their attention and en-

¹³This previous study also identified substantive differences in the make-up of drugs purchased in the U.S. vs. other Western countries (with U.S. customers driving a disproportionate fraction of demand for non-ED meds). While we still observe this pattern in the SpamIt data (with the fraction of non-ED meds in U.S. customer orders being 3.8× larger than for Europe and Canada), it is absent in GlavMed customers, suggesting that the advertising vector plays a key role in this effect.

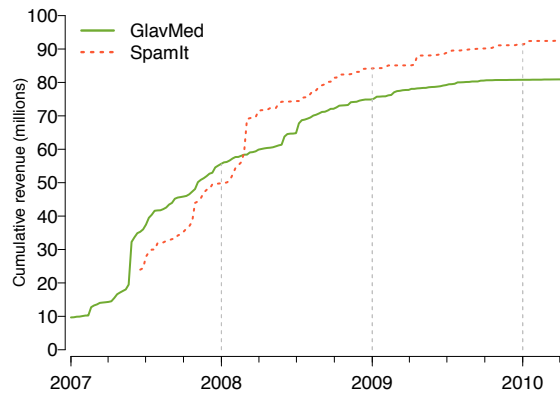


Figure 7: Cumulative contribution of new affiliates over time to the three-year total program revenue. Each week adds the contribution to total program revenue made by the new affiliates that appear that week.

ergy on the top performing affiliates. Alternatively, from an intervention perspective, undermining the activities of just a handful of affiliates would have a considerable affect on a program’s bottom line: undermining the top 3–10 affiliates would impact 25–40% of program revenue.

Moreover, there is evidence that these high-revenue affiliates are not simply lucky, but represent the best-established and experienced advertisers. Figure 7 shows that it is the oldest affiliates who contribute most to weekly program revenue on an ongoing basis. For both programs, the curves show the cumulative contribution to total program revenue over time for new affiliates. For the new affiliates that appear each week, we increment a running sum with the total revenue those affiliates generate for the program—revenue generated from the moment they join until the end of the measurement period. For instance, the affiliates that generate revenue in the first week account for nearly 10% of all revenue for the entire three years of business. The dashed lines show the contributions to total revenue by affiliates that have joined on year intervals, emphasizing that the older affiliates are important for generating revenue over time. Affiliates that joined before 2008 contributed 69% GlavMed and 54% of SpamIt total program revenue as of April 2010. In contrast, affiliates that joined in 2009 and 2010 contributed less than 10% of that total.

5.2.2 Affiliate Commissions

Since only a small fraction of affiliates account for much of the business, many affiliates earn small commissions. Indeed, the median annualized affiliate commissions for GlavMed, SpamIt, and RX-Promotion are just \$292, \$3,320, and \$428, respectively. This skew dovetails with suggestions that spam-based advertising may be a labor “lemon market” [5]. On the other hand, the most successful affiliates are able to derive substantial income through their advertising. Indeed, the top five affiliates were able

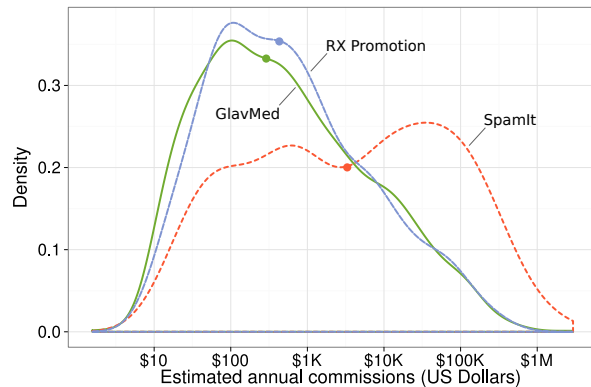


Figure 8: Distribution of affiliate commissions in each program.

to earn over \$1M for themselves in a twelve-month period (and a dozen exceeded \$500K).¹⁴ Virtually all of these earnings result from sales commissions with only a minor share deriving from referral commissions (i.e., referral commissions are not a major source of income).

Figure 8 reveals a more nuanced picture of affiliate commissions. For each program, the graph shows a PDF of annualized commissions across all affiliates: the *x*-axis is the annualized commission earned by an affiliate, and the *y*-axis is the fraction of all affiliates that earned a given commission. We calculate the commission for an affiliate using the total customer sales linked to the affiliate multiplied by the commission rate of the affiliate, plus any referral commissions. Sales commission rates range from 15–45%, with 30–40% being the most common (generally high-revenue affiliates receive the highest commission rates).¹⁵ The “dots” on the PDFs denote the median annualized commissions for that program.

For the open programs GlavMed and RX-Promotion, the majority of affiliates earn very low annualized commissions. The peaks of the PDFs range between \$20–\$200 a year for GlavMed, and \$20–\$2,000 a year for RX-Promotion. The closed program SpamIt, however, shows a bimodal distribution, with a mass of “poor” affiliates earning small commissions (mode around \$500) and another mass of “rich” affiliates earning large commissions (mode around \$30,000), but still with many affiliates earning over \$100,000 a year.

As another perspective, on an ongoing basis the active affiliates in SpamIt, a closed program, each generate three times more revenue than active affiliates in GlavMed and RX-Promotion, both open programs. Fig-

¹⁴Note that Figure 8 does not involve extrapolating, but is based on taking the best four consecutive quarter’s earnings for each affiliate and thus gains accuracy at the potential expense of right-censoring.

¹⁵Note that not all programs reward commissions uniformly over all drugs. For example, RX-Promotion typically discounts commissions by 10% on controlled drugs, so an affiliate receiving 40% on the sale of Viagra may only receive 30% on the sale of Oxycodone.

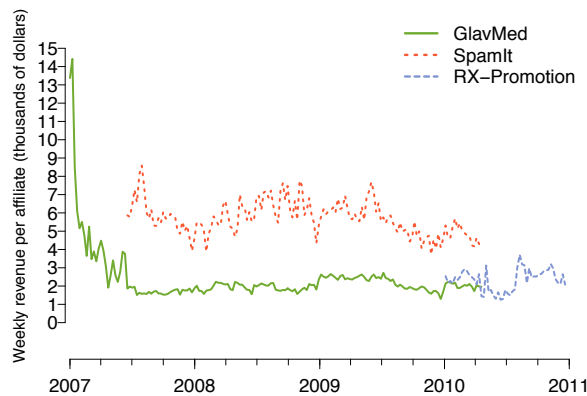


Figure 9: Average revenue per active affiliate each week.

Figure 9 shows the average weekly revenue generated by active affiliates. For each week, we total the revenue generated by the affiliates that were active in attracting customers that week, and divide by the number of active affiliates. This metric is surprisingly stable over time and strongly correlates with the nature of the affiliate program. In both GlavMed and RX-Promotion, the average weekly revenue per affiliate is around \$2,000. In SpamIt, though, the average weekly revenue per affiliate ranges between \$5,000–\$7,000. Open programs focus on increasing the total number of affiliates, but the vast majority have little impact on total revenue. Instead, by focusing on quality affiliates, the closed nature of the SpamIt program is much more effective at attracting productive affiliates and avoiding unproductive ones.

Focusing only on these most productive affiliates, we would intuitively expect them to also be the operators of the largest spamming botnets. However, even a cursory examination of the data shows that there is considerable more complexity at work. For example, while the operators of the prodigious Rustock botnet (*cosma2k*, *bird*, and *adv1*) indeed receive large commission payments (over \$1.9M), botnet operators do not appear to dominate the top earners. Indeed, two of the largest botnet operators, *docent* (operator of MegaD) and *severa* (operator of Storm and Waledac) only received modest payments of \$308k and \$169k, respectively, for directly advertising SpamIt sites.¹⁶

There are a number of potential reasons for these results. First, we are only privy to sales for these particular affiliate programs and thus, if a botnet devotes much of its resources to another program, those earnings are outside our analysis. Moreover, while some botnets are largely monopolized by their owners, in many other cases the botnets are rented to provide service for third

¹⁶We identify botnet operators through metadata, documented more fully in the many articles in the “PharmaWars” series [10], and corroborated based on which affiliates receive money for domains known to be advertised via particular botnets.

parties. For example, the second most profitable affiliate, *scorpp2*, earned close to \$3M while advertising domains that we witnessed emerging from a range of botnets including MegaD, Cutwail and Xarvester. Adding to the confusion, in a number of cases botnet code is sold between parties and, thus, what some researchers may identify as a single botnet may in fact reflect multiple distinct infrastructures. Finally, we also note spamming is not the only profitable advertising vector. Indeed, the largest overall earner, *webplanet*, appears to have earned \$4.6M using Web-based advertising instead. Fully unraveling the complexities of these relationships and why certain affiliates are more successful than others remains an open question.

5.3 Costs

Affiliate programs operate a complex business. As such, they have a range of costs and overheads to cover and only a fraction of their revenue translates to profit. Using a combination of transactional and metadata, we next reconstruct both direct and indirect costs for the programs. We also explore in more detail the cost structure of fulfillment (drug markup and shipping).

5.3.1 Direct Costs and Gross Margin

Direct costs are costs attributable to individual sales. While advertising is normally considered an indirect cost, affiliate programs pay for advertising as a direct cost of a sale, so we consider affiliate commissions to be a direct cost in this context. In addition, direct costs include the supplier costs for the products themselves, shipping them to customers, the fees charged by banks and credit card processors for processing customer credit card transactions, and customer refunds.

However, of these quantities only commissions are completely unambiguously encoded across all transactional data sets; RX-Promotion also includes a measure of the supplier cost and a field indicating the type of shipping (from which the shipping cost can be reverse engineered). The situation with GlavMed and SpamIt is more complex. Starting on August 8, 2008 both databases include fine-grained information about shipping and supply cost for each order. For periods before this, we are forced to extrapolate. Refunds can be calculated directly in the SpamIt and GlavMed data sets; for RX-Promotion, we infer refunds based on orders with a cancelled status. Finally, processing charges can vary among payment processors, currencies, card brands and over time. However, in examining a large number of recorded fees (found in the chatlogs) over the full period these fees range between 7–12% in practice, so as an approximation we use 10%.

Putting this data together, Table 3 itemizes the gross revenue and direct cost breakdown for GlavMed and

	GlavMed & SpamIt					RX-Promotion
	2007	2008	2009	2010	2010	
Gross revenue	\$27.3M	\$60.1M	\$67.7M	\$18.0M	\$12.8M	
Direct costs	\$17.2M (63.1%)	\$42.9M (71.4%)	\$45.6M (67.3%)	\$12.1M (67.1%)	\$9.9M (77.1%)	
Commissions	\$7.9M (28.9%)	\$23.0M (38.3%)	\$24.9M (36.8%)	\$6.6M (36.7%)	\$3.9M (30.2%)	
Suppliers (goods) ^a	\$1.9M (7%)	\$4.3M (7.2%)	\$4.2M (6.2%)	\$1.1M (6.1%)	\$1.0M (7.6%)	
Suppliers (shipping) ^b	\$3.1M (11.4%)	\$7.6M (12.6%)	\$7.8M (11.5%)	\$2.1M (11.7%)	\$1.5M (11.5%)	
Processing ^c	\$2.7M (10%)	\$6.0M (10%)	\$6.8M (10%)	\$1.8M (10%)	\$1.3M (10%)	
Refunds	\$1.6M (5.9%)	\$2.0M (3.3%)	\$1.9M (2.8%)	\$0.5M (2.6%)	\$1.0M (7.8%)	
Gross margin	\$10.1M (36.9%)	\$17.2M (28.6%)	\$22.1M (32.7%)	\$5.9M (32.9%)	\$2.9M (22.9%)	

^a Average supplier costs used to estimate missing supplier costs for 35% of goods.

^b Average shipping costs used to estimate missing shipping costs for 60% of orders.

^c Processor costs range between 7% and 11% of sales revenue.

Table 3: Gross revenue, direct costs and resulting gross margin for the GlavMed and SpamIt programs combined.

SpamIt (combined) and RX-Promotion on a yearly basis. Not surprisingly (given average affiliate commissions of 30–40%) direct costs consume the majority of revenue. Note that, due to holdback charges, the gross margin number likely overstates cash flow by around 10%, and may in fact overstate revenue as well (if holdback charges are not released). Payment processors comporting with “high risk” merchants such as these universally hold back a portion of net proceeds to handle future chargebacks and fines. From examining the logs, a 10% holdback of up to 180 days is common and, in reviewing discussions about holdbacks, the operators of GlavMed/SpamIt routinely operate under the assumption that this money may never be made available.

5.3.2 Indirect Costs and Net Revenue

Indirect costs are costs that are not generally attributable to individual sales. For online pharmacies, indirect costs are incurred for marketing (i.e., advertising the affiliate program on popular blogs and forums to attract new *affiliates*), for IT (i.e., registering domains for affiliates to use in URLs that link to storefront pages, as well as server and hosting costs), for administrative costs (i.e., staff salaries), customer service, bank fines and “lobbying”. By also calculating indirect costs, we can then estimate a program’s net profit—its proverbial “bottom line.”

However, indirect costs are difficult to extract from transaction data since they are necessarily *indirect*. Thus, for this analysis we focus in particular on RX-Promotion for which we have highly detailed metadata comprising the raw monthly balance sheets (in spreadsheet form) for seven months of revenue. The full spreadsheet is too large to reproduce here, but we have extracted the equivalent direct costs that we calculated from transactional data in Table 3, and aggregated indirect costs in key areas. We summarize the resulting balance sheet in Table 4, reflecting seven months of revenue between March and September in 2010.

The direct costs taken from the balance sheet data are highly similar to the transactional equivalents, dif-

	RX-Promotion March – September 2010
Gross revenue	\$7.8M
Direct costs	\$5.5M (70.8%)
Commissions	\$3M (38.1%)
Suppliers ^a	\$1.4M (17.6%)
Processing	\$1M (13.2%)
Other direct	\$148.3K (1.9%)
Indirect costs	\$1004K (12.8%)
Administrative	\$197K (2.5%)
Customer service	\$124K (1.6%)
Fines	\$107K (1.4%)
IT expenses	\$202K (2.6%)
Domains	\$114K (1.5%)
Servers, hosting	\$66K (0.8%)
Selling expenses	\$315K (4%)
Marketing	\$105K (1.3%)
Lobbying	\$157K (2%)
Other indirect	\$134K (1.7%)
Net revenue	\$1.3M (16.3%)

^a Costs of goods and shipping are combined.

Table 4: Balance sheet for RX-Promotion detailing indirect costs.

fering primarily due to differences in the make-up of commission tiers during this seven-month period and the greater precision available for payment processing overheads. Overall indirect costs represent almost 13% of gross, split among a range of different overheads. Note that the \$157k lobbying charge is concentrated in two large payments which may be related to conflict between RX-Promotion and GlavMed/SpamIt. Overall, the net revenue for this period—the profit returned to the affiliate program owners—is just 16.3% of gross revenue. This value is not uniform from month to month, however. For example, during the period when processing for controlled drugs was lost, RX-Promotion simultaneously lost revenue, incurred large fines, and had to pay greater average commissions (since the commissions for controlled drugs were discounted 10%) leading to a net

loss for at least one month. By contrast, during the very best month (September) net revenue exceeds 30%.

We do not have equivalent indirect cost data for GlavMed or SpamIt, but we are able to infer a subset of these overheads. The operators used a special affiliate (`affiliate_id` value 20) to manage the working capital of each. The Affiliate 20 account received referral commissions from all affiliates who did not have a referring affiliate designated explicitly. During the measurement period, Affiliate 20 earned \$2.7M. Operating expenditures, as well as some affiliate payouts, were deducted from this account.

Starting May 2009, the `comment` field of each payout began including a short description of the payment. A payment for a banner advertisement (recruiting affiliates), for example, would be listed as described as “banner GM - gofuckbiz.com”. Although free-form, the comment text typically used a small number of phrases. Using a manually generated list of regular expressions, we identified several indirect costs during the period from May 2009 to April 2010. These costs include marketing (\$153k, 0.2% of revenue), domain purchasing (\$511k, 0.8% of revenue) and servers/hosting (\$247k, 0.4% of revenue). Interestingly, it appears that marketing and servers/hosting are similar costs between the two programs (suggesting they are largely fixed costs) but domain purchasing appears to track revenue (presumably since greater advertising volume requires more domain turnover due to blacklisting).

Finally, we also have anecdotal data in the form of chat logs between the lead operator and the owner of GlavMed/SpamIt. These logs state that overall net revenue fluctuated between 10% and 20%, agreeing structurally with the RX-Promotion data.

Thus, we believe that 10–20% is likely to reflect a typical net revenue for successful pharmaceutical programs. While this is smaller on an earnings-per-sale basis than the commissions awarded to individual affiliates, it is a more profitable enterprise when the affiliate program is successful. For example, the largest SpamIt affiliate might make \$2M in a year, but in that same year the program itself is likely to clear over \$10M in profit.

5.3.3 Markup

After commissions, supply costs for the programs are one of the largest expenses. Using the categories from Figure 2, ED contains by far the most popular products purchased, and also has the highest markups of more than 15 to 20 *times* the supply cost. The average markup of Viagra in GlavMed and SpamIt, for instance, translates to a customer price 25 times cost. Markups in the Abuse and Chronic categories are considerably smaller, ranging between 5–8 times supply cost. Interestingly, the shipping cost is a loss leader for GlavMed/SpamIt since they

charge a flat fee per order (orders with more than one item result in supplier shipping costs higher than collected shipping fees) and offer free shipping for orders over \$200. In fact, for the orders for which we have fine-grained product and shipping cost data, the supplier costs of delivering the drugs (8.5M) actually exceeded the costs of the drugs delivered.

5.4 Payment Processing

Finally, affiliate programs must arrange for reliable processing of customer payments. In a sense, obtaining reliable payment processing services may be the most important function of the affiliate program, since it is the only mechanism by which all other efforts can be monetized. Previously, our group identified that a small number of banks were critical to virtually all online pharmaceutical sales [14]. However, the means by which those banks were accessed has never been well documented.

In fact, in the “high-risk” payment market, merchant processing is frequently handled by independent Payment Service Providers (PSPs) who manage the relationships with acquiring banks and provide Web-based payment gateway services to clients.¹⁷ While users of these services may have a contractual relationship with the bank, in other cases PSPs may “front” their own merchant accounts on behalf of their clients (a form of identity laundering called “factoring” and typically disallowed by card association rules). Merchants in turn can mitigate some of their own risk by working with multiple providers; this strategy not only provides redundancy, but each provider may place limits on transaction volumes (e.g., to fit within the underwriting risk limits on their overall merchant portfolio) and may have different services they are willing to offer (e.g., MC, Visa, Amex, eCheck, etc.) for different product categories (e.g., herbal vs. prescription vs. controlled drugs).

In the case of RX-Promotion the affiliate program enjoyed a partnership with a large ISO/PSP and thus this entity handled virtually all of their processing needs. GlavMed and SpamIt, by contrast, did not work with any single provider, but no less than twenty-one distinct providers over the lifetime of our data sets. However, these providers differ considerably in what services they are used for, the volume of transactions they are able to handle and how long-lived they are. In fact, almost half of these providers are never used to process significant transaction volumes (mostly likely due to risk controls).

Illustrating this point, Figure 10 graphs the transaction volume of GlavMed/SpamIt handled by different payment service providers over time. The y-axis identifies

¹⁷We use the term “payment service provider” here in a generic sense, and the organizations involved may be some combination of proper PSPs, account brokers, merchant servicers, ISO/MSPs with third-party servicers, etc.

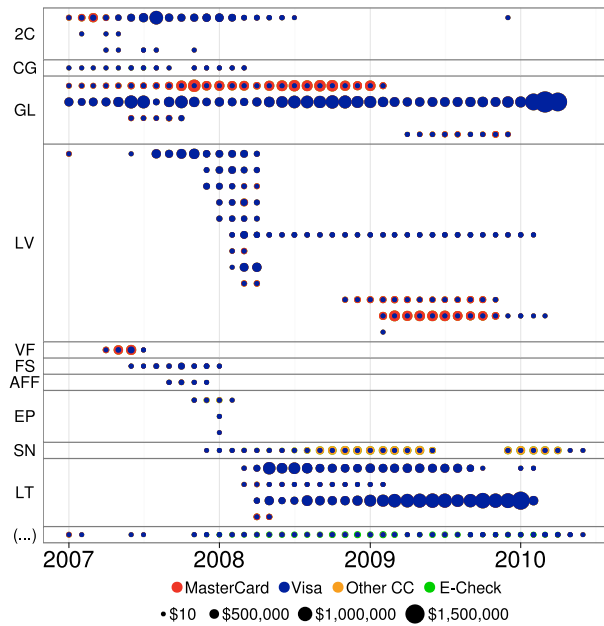


Figure 10: Payment transactions over time by payment service provider. The colored volume of each circle corresponds to the transaction volume in a month for a particular terminal (color indicating payment method), with terminals grouped by providers.

the top nine providers (using a designator taken directly from the database or an abbreviation thereof) while the remaining providers are aggregated together under the ellipsis. Each circle in the graph represents the number of transactions processed via a particular *terminal* in a month, with terminals belonging to a particular provider grouped together based on time of first use.¹⁸ In any given circle, the color red indicates MasterCard transactions, blue is for Visa, yellow for other credit cards (primarily Amex), and green for eCheck.

There are a number of striking observations one can draw from this figure. First is the clear dominance of Visa processing. Aggregating across both GlavMed and SpamIt, Visa transactions represent almost 67% of all revenue, followed by MasterCard with 23% and American Express with 6% (with the remainder concentrated in eCheck transactions through the ACH system). While part of this discrepancy is likely due to demand—Visa is the most popular payment card brand—this difference also reflects a supply issue as well. For reasons not entirely clear, it has traditionally been far easier for online pharmaceutical programs to obtain payment processing services for Visa than for MasterCard or Amex. Indeed,

¹⁸A terminal is effectively an interface point for sending payment transactions, corresponding to a particular merchant account. Note that while some terminals are for general purpose use, others service a particular function such as providing a compatible base currency (e.g., the terminal named “lt-euro-visa” provides European Visa transactions) or handling rebills (e.g., “gl-rebill-m”).

we find that during periods in which MasterCard processing was *available*, Visa/MasterCard revenue percentages stabilized at around 63%/30%, respectively, for both GlavMed and SpamIt.

Second, a relatively small number of payment service providers dominate the transaction volume (in particular GL, LT and LV). Together these three providers are responsible for 84% of all revenue for GlavMed and SpamIt. Many of the other providers are active for very short lifetimes, and with very low volumes, before they are either abandoned or, more typically, they are unwilling to continue business with the program operators.

Finally, there are also clear patterns indicative of problems with particular providers over time. For example, for each terminal a sudden drop in volume and rise in declines (not shown) is typically a precursor to that terminal being abandoned. Some of these cases clearly reflect changes in long-term business relationships: in March of 2008, for instance, there is a clear transition moving the largest volume of Visa processing between LV and LT; similarly, American Express processing moves from AFF to SN during the same period. In the last five months of 2010 it appears that GlavMed/SpamIt experienced significant setbacks in processing capability, with LT processing only minor volumes (forcing them to push a higher volume of transactions through GL). These findings provide additional support and context for our previous findings that the financial aspect of the counterfeit pharmaceutical ecosystem is among the most fragile components [14].

6 Conclusion

This paper provides an unprecedented view inside the economics of modern pharmaceutical affiliate programs: an enterprise that ultimately capitalizes a wide array of infrastructure services including botnets, malware, bullet-proof hosting and so on. Among the results of this work, we have shown that the customer market is large and far from fully tapped, with repeat orders playing a key role in mature programs. We have also seen that a small number of big affiliates can dominate the revenue equation and that disrupting these particular affiliates would have disproportionate damage on the whole program. Finally, even very large programs such as GlavMed/SpamIt depend on a handful of payment service providers to reliably monetize their activities, reinforcing the observation that financial services are a “weak point” in the value chain. Surprisingly, while affiliate programs can drive substantial sales, their costs are significant and ultimately net revenues are modest, typically under just 20% of sales. This finding again suggests that such organizations are fragile to economic disruptions of even a modest scale.

Acknowledgments

We would like to thank the various anonymous providers of our data sets, without which there would have been no paper. We have also benefited heavily from the many members of the cyber-investigations community who have provided us valuable insight as we have tried to map data onto meaning. Closer to home, we would like to thank Erin Kenneally for her ongoing legal guidance and ethical oversight, as well as the technical support of Brian Kantor and Cindy Moore who have managed our systems and storage needs.

This work was supported in part by National Science Foundation grants NSF-0433668, NSF-0433702, NSF-0831138 and CNS-0905631, by the Office of Naval Research MURI grant N000140911081, and by generous research, operational and/or in-kind support from Google, Microsoft, Yahoo, Cisco, HP and the UCSD Center for Networked Systems (CNS).

References

- [1] D. S. Anderson, C. Fleizach, S. Savage, and G. M. Voelker. Spamscatter: Characterizing Internet Scam Hosting Infrastructure. In *Proc. of 16th USENIX Security*, 2007.
- [2] Behind Online Pharma. From Mumbai to Riga to New York: Our Investigative Class Follows the Trail of Illegal Pharma. <http://behindonlinepharma.com>, 2009.
- [3] J. Caballero, C. Grier, C. Kreibich, and V. Paxson. Measuring Pay-per-Install: The Commoditization of Malware Distribution. In *Proc. of 20th USENIX Security*, 2011.
- [4] C. Grier, K. Thomas, V. Paxson, and M. Zhang. @spam: The Underground on 140 Characters or Less. In *Proc. of 17th ACM CCS*, 2010.
- [5] C. Herley and D. Florêncio. Nobody Sells Gold for the Price of Silver: Dishonesty, Uncertainty and the Underground Economy. In *Proc. of 8th WEIS*, 2009.
- [6] J. P. John, A. Moshchuk, S. D. Gribble, and A. Krishnamurthy. Studying Spamming Botnets Using Botlab. In *Proc. of 6th NSDI*, 2009.
- [7] J. P. John, F. Yu, Y. Xie, A. Krishnamurthy, and M. Abadi. deSEO: Combating Search-Result Poisoning. In *Proc. of 20th USENIX Security*, 2011.
- [8] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. Spamalytics: An Empirical Analysis of Spam Marketing Conversion. In *Proc. of 15th ACM CCS*, 2008.
- [9] C. Kanich, N. Weaver, D. McCoy, T. Halvorson, C. Kreibich, K. Levchenko, V. Paxson, G. M. Voelker, and S. Savage. Show Me the Money: Characterizing Spam-advertised Revenue. In *Proc. of 20th USENIX Security*, 2011.
- [10] B. Krebs. SpamIt, Glavmed Pharmacy Networks Exposed. Krebs on Security Blog, <http://www.krebssecurity.com/category/pharma-wars/>, 2011.
- [11] LegitScript and KnujOn. No Prescription Required: Bing.com Prescription Drug Ads. <http://www.legitscript.com/download/BingRxReport.pdf>, 2009.
- [12] LegitScript and KnujOn. Yahoo! Internet Pharmacy Advertisements. <http://www.legitscript.com/download/YahooRxAnalysis.pdf>, 2009.
- [13] N. Leontiadis, T. Moore, and N. Christin. Measuring and Analyzing Search-Redirection Attacks in the Illicit Online Prescription Drug Trade. In *Proc. 20th USENIX Security*, 2011.
- [14] K. Levchenko, N. Chachra, B. Enright, M. Felegyhazi, C. Grier, T. Halvorson, C. Kanich, C. Kreibich, H. Liu, D. McCoy, A. Pitsillidis, N. Weaver, V. Paxson, G. M. Voelker, and S. Savage. Click Trajectories: End-to-End Analysis of the Spam Value Chain. In *Proc. of 32nd IEEE Security and Privacy*, 2011.
- [15] H. Liu, K. Levchenko, M. Félegyházi, C. Kreibich, G. Maier, G. M. Voelker, and S. Savage. On the Effects of Registrar-level Intervention. In *Proc. of 4th USENIX LEET*, 2011.
- [16] B. S. McWilliams. *Spam Kings: The Real Story Behind the High-Rolling Hucksters Pushing Porn, Pills and @*#?% Enlargements*. O'Reilly Media, Sept. 2004.
- [17] A. Ramachandran and N. Feamster. Understanding the Network-Level Behavior of Spammers. In *Proceedings of ACM SIGCOMM*, Pisa, Italy, Sept. 2006.
- [18] D. Samosseiko. The Partnerka — What is it, and why should you care? In *Proc. of Virus Bulletin Conference*, 2009.
- [19] Y. Shin, M. Gupta, and S. Myers. The Nuts and Bolts of a Forum Spam Automator. In *Proc. of 4th USENIX LEET*, 2011.
- [20] B. Stone-Gross, R. Abman, R. Kemmerer, C. Kruegel, D. Steigerwald, and G. Vigna. The Underground Economy of Fake Antivirus Software. In *Proc. of 10th WEIS*, 2011.
- [21] Symantec. MessageLabs June 2010 Intelligence Report. http://www.symanteccloud.com/mlireport/MLI_2010_06_June_FINAL.pdf.
- [22] K. Thomas, C. Grier, V. Paxson, and D. Song. Suspended Accounts In Retrospect: An Analysis of Twitter Spam. In *Proc. of 11th IMC*, 2011.
- [23] D. Wang, S. Savage, and G. M. Voelker. Cloak and Dagger: Dynamics of Web Search Cloaking. In *Proc. of 18th CCS*, 2011.
- [24] Y.-M. Wang, M. Ma, Y. Niu, and H. Chen. Spam Double-Funnel: Connecting Web Spammers with Advertisers. In *Proc. of 16th WWW*, 2007.
- [25] G. Wondracek, T. Holz, C. Platzer, E. Kirda, and C. Kruegel. Is the Internet for Porn? An Insight into the Online Adult Industry. In *Proc. of 9th WEIS*, 2010.

B@bel: Leveraging Email Delivery for Spam Mitigation

Gianluca Stringhini[§], Manuel Egele[§], Apostolis Zarras[‡], Thorsten Holz[‡],
Christopher Kruegel[§], and Giovanni Vigna[§]

[§]University of California, Santa Barbara

{gianluca, maeg, chris, vigna}@cs.ucsb.edu
{apostolis.zarras, thorsten.holz}@rub.de

[‡]Ruhr-University Bochum

Abstract

Traditional spam detection systems either rely on *content analysis* to detect spam emails, or attempt to detect spammers before they send a message, (i.e., they rely on the *origin* of the message). In this paper, we introduce a third approach: we present a system for filtering spam that takes into account *how* messages are sent by spammers. More precisely, we focus on the email delivery mechanism, and analyze the communication at the SMTP protocol level.

We introduce two complementary techniques as concrete instances of our new approach. First, we leverage the insight that different mail clients (and bots) implement the SMTP protocol in slightly different ways. We automatically learn these *SMTP dialects* and use them to detect bots during an SMTP transaction. Empirical results demonstrate that this technique is successful in identifying (and rejecting) bots that attempt to send emails. Second, we observe that spammers also take into account *server feedback* (for example to detect and remove non-existent recipients from email address lists). We can take advantage of this observation by returning fake information, thereby poisoning the server feedback on which the spammers rely. The results of our experiments show that by sending misleading information to a spammer, it is possible to prevent recipients from receiving subsequent spam emails from that same spammer.

1 Introduction

Email spam, or *unsolicited bulk email*, is one of the major open security problems of the Internet. Accounting for more than 77% of the overall world-wide email traffic [21], spam is annoying for users who receive emails they did not request, and it is damaging for users who fall for scams and other attacks. Also, spam wastes resources on SMTP servers, which have to process a significant amount of unwanted emails [41].

A lucrative business has emerged around email spam, and recent studies estimate that large affiliate cam-

paigns generate between \$400K and \$1M revenue per month [20].

Nowadays, about 85% of world-wide spam traffic is sent by *botnets* [40]. Botnets are networks of compromised computers that act under the control of a single entity, known as the *botmaster*. During recent years, a wealth of research has been performed to mitigate both spam and botnets [18, 22, 29, 31, 33, 34, 50].

Existing spam detection systems fall into two main categories. The first category focuses on the *content* of an email. By identifying features of an email's content, one can classify it as spam or *ham* (i.e., a benign email message) [16, 27, 35]. The second category focuses on the *origin* of an email [17, 43]. By analyzing distinctive features about the sender of an email (e.g., the IP address or autonomous system from which the email is sent, or the geographical distance between the sender and the recipient), one can assess whether an email is likely spam, without looking at the email content.

While existing approaches reduce spam, they also suffer from limitations. For instance, running content analysis on every received email is not always feasible for high-volume servers [41]. In addition, such content analysis systems can be evaded [25, 28]. Similarly, origin-based techniques have coverage problems in practice. Previous work showed how IP blacklisting, a popular origin-based technique [3], misses a large fraction of the IP addresses that are actually sending spam [32, 37].

In this paper, we propose a novel, third approach to fight spam. Instead of looking at the content of messages (what) or their origins (who), we analyze *the way* in which emails are sent (how). More precisely, we focus on the email delivery mechanism. That is, we look at the communication between the sender of an email and the receiving mail server at the SMTP protocol level. Our approach can be used in addition to traditional spam defense mechanisms. We introduce two complementary techniques as concrete instances of our new approach: *SMTP dialects* and *Server feedback manipulation*.

SMTP dialects. This technique leverages the observation that different email clients (and bots) implement the SMTP protocol in slightly different ways. These deviations occur at various levels, and range from differences in the case of protocol keywords, to differences in the syntax of individual messages, to the way in which messages are parsed. We refer to deviations from the strict SMTP specification (as defined in the corresponding RFCs) as *SMTP dialects*. As with human language dialects, the listener (the server) typically understands what the speaker (a legitimate email client or a bot) is saying. This is because SMTP servers, similar to many other Internet services, follow *Postel's law*, which states: "Be liberal in what you accept, and conservative in what you send."

We introduce a model that represents SMTP dialects as state machines, and we present an algorithm that learns dialects for different email clients (and their respective email engines). Our algorithm uses both passive observation and active probing to efficiently generate models that can distinguish between different email engines. Unlike previous work on service and protocol fingerprinting, our models are stateful. This is important, because it is almost never enough to inspect a single message to be able to identify a specific dialect.

Leveraging our models, we implement a decision procedure that can, based on the observation of an SMTP transaction, determine the sender's dialect. This is useful, as it allows an email server to terminate the connection with a client when this client is recognized as a spambot. The connection can be dropped before any content is transmitted, which saves computational resources at the server. Moreover, the identification of a sender's dialect allows analysts to group bots of the same family, or track the evolution of spam engines within a single malware family.

Server feedback manipulation. The SMTP protocol is used by a client to send a message to the server. During this transaction, the client receives from the server information related to the delivery process. One important piece of information is whether the intended recipient exists or not. The performance of a spam campaign can improve significantly when a botmaster takes into account server feedback. In particular, it is beneficial for spammers to remove non-existent recipient addresses from their email lists. This prevents a spammer from sending useless messages during subsequent campaigns. Indeed, previous research has shown that certain bots report the error codes received from email servers back to their command and control nodes [22, 38].

To exploit the way in which botnets currently leverage server feedback, it is possible to manipulate the responses from the mail server to a bot. In particular, when

a mail server identifies the sender as a bot, instead of dropping the connection, the server could simply reply that the recipient address does not exist. To identify a bot, one can either use traditional origin-based approaches or leverage the SMTP dialects proposed in this paper. When the server feedback is poisoned in this fashion, spammers have to decide between two options. One possibility is to continue to consider server feedback and, as a result, remove valid email addresses from their email list. This reduces the spam emails that these users will receive in the future. Alternatively, spammers can decide to distrust and discard any server feedback. This reduces the effectiveness of future campaigns since emails will be sent to non-existent users.

Our experimental results demonstrate that our techniques are successful in identifying (and rejecting) bots that attempt to send unwanted emails. Moreover, we show that we can successfully poison spam campaigns and prevent recipients from receiving subsequent emails from certain spammers. However, we recognize that spam is an adversarial activity and an arms race. Thus, a successful deployment of our approach might prompt spammers to adapt. We discuss possible paths for spammers to evolve, and we argue that such evolution comes at a cost in terms of performance and flexibility.

To summarize, the paper makes the following main contributions:

- We introduce a novel approach to detect and mitigate spam emails. This approach focuses on the email delivery mechanism — the SMTP communication between the email client and the email server. It is complementary to traditional techniques that operate either on the message origin or on the message content.
- We introduce the concept of SMTP dialects as one concrete instance of our approach. Dialects capture small variations in the ways in which clients implement the SMTP protocol. This allows us to distinguish between legitimate email clients and spambots. We designed and implemented a technique to automatically learn the SMTP dialects of both legitimate email clients and spambots.
- We implemented our approach in a tool, called `B@bel`. Our experimental results demonstrate that `B@bel` is able to correctly identify spambots in a real-world scenario.
- We study how the feedback provided by email servers to bots is used by their botmasters. As a second instance of our approach, we show how providing incorrect feedback to bots can have a negative impact on the spamming effectiveness of a botnet.

2 Background: The SMTP Protocol

The Simple Mail Transfer Protocol (*SMTP*), as defined in RFC 821 [1], is a text-based protocol that is used to send email messages originating from *Mail User Agents* (MUAs — e.g., Outlook, Thunderbird, or Mutt), through intermediate *Mail Transfer Agents* (MTAs — e.g., Sendmail, Postfix, or Exchange) to the recipients' mailboxes. The protocol is defined as an alternating dialogue where the sender and the receiver take turns transmitting their messages. Messages sent by the sender are called *commands*, and they instruct the receiver to perform an action on behalf of the sender. The SMTP RFC defines 14 commands. Each command consists of four case-insensitive, alphabetic-character command codes (e.g., MAIL) and additional, optional arguments (e.g., FROM:<me@example.com>). One or more space characters separate command codes and argument fields. All commands are terminated by a line terminator, which we denote as <CR><LF>. An exception is the DATA command, which instructs the receiver to accept the subsequent lines as the email's content, until the sender transmits a dot character as the only character on a line (i.e., <CR><LF>.<CR><LF>).

SMTP *replies* are sent by the receiver to inform the sender about the progress of the email transfer process. Replies consist of a three-digit status code, followed by a space separator, followed by a short textual description. For example, the reply 250 OK indicates to the sender that the last command was executed successfully. Commonly, replies are one line long and terminated by <CR><LF>¹. The RFC defines 21 different reply codes. These codes inform the sender about the specific state that the receiver has advanced to in its protocol state machine and, thus, allows the sender to synchronize its state with the state of the receiver. A plethora of additional RFCs have been introduced to extend and modify the original SMTP protocol. For example, RFC 1869 introduced *SMTP Service Extensions*. These extensions define how an SMTP receiver can inform a client about the extensions it supports. More precisely, if a client wants to indicate that it supports SMTP Service Extensions, it will greet the server with EHLO instead of the regular HELO command. The server then replies with a list of available service extensions as a multi-line reply. For example, a server capable of handling encryption can announce this capability by responding with a 250-STARTTLS reply to the client's EHLO command.

MTAs, mail clients, and spambots implement different sets of these extensions. As we will discuss in de-

¹The protocol allows the server to answer with multi-line replies. In a multi-line reply, all lines but the last must begin with the status code followed by a dash character. The last line of a multi-line reply must be formatted like a regular one-line reply

```
Server: 220 debian
Client: HELO example.com
Server: 250 OK
Client: MAIL FROM:<me@example.com>
Server: 250 2.1.0 OK
Client: RCPT TO:<you@example.com>
Server: 250 2.1.5 OK
Client: DATA
```

Figure 1: A typical SMTP conversation

tail later, we leverage these differences to determine the SMTP dialect spoken in a specific SMTP conversation.

In this paper, we consider an *SMTP conversation* the sequence of commands and replies that leads to a DATA command, to a QUIT command, or to an abrupt termination of the connection. This means that we do not consider any reply or command that is sent after the client starts transmitting the actual content of an email. An example of an SMTP conversation is listed in Figure 1.

3 SMTP Dialects

The RFCs that define SMTP specify the protocol that a client has to speak to properly communicate with a server. However, different clients might implement the SMTP protocol in slightly different ways, for three main reasons:

1. The SMTP RFCs do not always provide a single possible format when specifying the commands a client must send. For example, command identifiers are case insensitive, which means that EHLO and ehlo are both valid command codes.
2. By using different SMTP extensions, clients might add different parameters to the commands they send.
3. Servers typically accept commands that do not comply with the strict SMTP definitions. Therefore, a client might implement the protocol in slightly wrong ways while still succeeding in sending email messages.

We call different implementations of the SMTP protocol *SMTP dialects*. A dialect \mathbf{D} is defined as a state machine

$$\mathbf{D} = \langle \Sigma, S, s_0, T, F_g, F_b \rangle,$$

where Σ is the input alphabet (composed of server replies), S is a set of states, s_0 is the initial state, and T is a set of transitions. Each state s in S is labeled with a client command, and each transition t in T is labeled with a server reply. $F_g \subseteq S$ is a set of good states, which represent successful SMTP conversations, while $F_b \subseteq S$ is a set of bad states, which represent failed SMTP conversations.

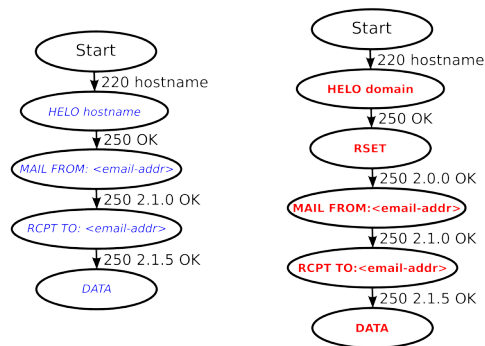


Figure 2: Simplified state machines for Outlook Express (left) and Bagle (right).

The state machine **D** captures the order in which commands are sent in relation to server replies by that particular dialect.

Since SMTP messages are not always constant, but contain variable fields (e.g., the recipient email address in an RCPT command), we abstract commands and replies as templates, and label states and transitions with such templates.

We do not require **D** to be deterministic. The reason for this is that some clients show a non-deterministic behavior in the messages they exchange with SMTP servers. For example, bots belonging to the *Lethic* malware family use EHLO and HELO interchangeably when responding to a server 220 reply. Figure 2 shows two example dialect state machines (Outlook Express and Bagle, a spambot).

3.1 Message Templates

As explained previously, we label states and transitions with message templates. We define the templates of the messages that belong to a dialect as regular expressions. Each message is composed of a sequence of tokens. We define a token as any sequence of characters separated by delimiters. We define spaces, colons, and equality symbols as delimiters. We leverage domain knowledge to develop a number of regular expressions for the variable elements in an SMTP conversation. In particular, we define regular expressions for email addresses, fully qualified domain names, domain names, IP addresses, numbers, and hostnames (see Figure 3 for details). Every token that does not match any of these regular expressions is treated as a keyword.

An example of a message template is

```
MAIL From: <email-addr>
```

where `email-addr` is a regular expression that matches email addresses.

Given two dialects **D** and **D'**, we consider them different if their state machines are different. For example, the two dialects in Figure 2 differ in the sequence of commands that the two clients send: Bagle sends a RSET

```
Email address: <?[\w\.-]+@[\w\.-]+>?
IP address: \[?\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\]?
Fully qualified domain name: [\w-]+\.[\w-]+\.[\w-]+
Domain name: [\w-]+\.[\w-]+
Number: [0-9]{3}[0-9]+
Hostname: [\w-]{5}[\w-]+
```

Figure 3: Regular expressions used in message templates.

command after the HELO, while Outlook Express sends a MAIL command directly. Also, the format of the commands of the two dialects differs: Outlook Express puts a space between MAIL FROM: and the sender email address, while Bagle does not.

In Section 4, we show how we can learn the dialect spoken by an SMTP client. In Section 5, we show how these learned dialects can be matched against an SMTP conversation, which is crucial for performing spam mitigation, as we will show in Section 7.

4 Learning Dialects

To distinguish between different SMTP speakers, we require a mechanism that learns which dialect is spoken by a particular client. To do this, we need a set of SMTP conversations **C** generated by the client. Each conversation is a sequence of `<reply, command>` pairs, where `command` can be empty if the client did not send anything after receiving a reply from the server.

It is important to note that the state machine learned for the dialect is affected by the type of conversations in **C**. For example, if **C** only contains successful SMTP conversations, the portion of the dialect state machine that we can learn from it is very small. In the typical SMTP conversation listed in Figure 1, the client first connects to the SMTP server, then announces itself (i.e., sends a HELO command), states who the sender of the email is (i.e., sends a MAIL command), lists recipients (by using one or more RCPT commands), and starts sending the actual email content (by sending a DATA command). Observing this type of communication gives no information on what a client would do upon receiv-

ing a particular error, or a specific SMTP reply from the server. To mitigate this problem, we collect a diverse set of SMTP conversations. We do this by directing the client to an SMTP server under our control, and sending specific SMTP replies to it (see Section 4.2).

Even though sending specific replies allows us to explore more states than the ones we could explore otherwise, we still cannot be sure that the dialects we learn are complete. In Section 7, we show how the inferred state machines are usually good enough for discriminating between different SMTP dialects. However, in some cases, we might not be able to distinguish two different dialects because the learned state machines are identical.

4.1 Learning Algorithm

Analyzing the set C allows us to learn part of the dialect spoken by the client. Our learning algorithm processes one SMTP conversation from C at a time, and iteratively builds the dialect state machine.

4.1.1 Learning the Message Templates

For each message observed in a conversation Con in C , our algorithm generates a regular expression that matches it. The regular expression generation algorithm works in three steps:

Step 1: First, we split the message into tokens. As mentioned in Section 3.1, we consider the space, colon, and equality characters as delimiters.

Step 2: For each token, we check if it matches a known regular expression. More precisely, we check it against all the regular expressions defined in Figure 3, from the most specific to the least specific, until one matches (this means that we check the regular expressions in the following order: email address, IP address, fully qualified domain name, domain name, number, hostname).

If a token matches a regular expression, we substitute the token with the matched regular expression's identifier (e.g., `<email-addr>`). If none of the regular expressions are matched, we consider the token a keyword, and we include it verbatim in the template.

Step 3: We build the message template, by concatenating the template tokens (which can be keywords or regular expressions) and the delimiters, in the order in which we encountered them in the original message.

Consider, for example, the command:

```
MAIL FROM:<evil@example.com>
```

First, we break the command into tokens:

```
[MAIL, FROM, <evil@example.com>]
```

The only token that matches one of the known regular expressions is the email address. Therefore, we consider the other tokens as keywords. The final template for this command will therefore be:

```
MAIL FROM:<email-addr>
```

Notice that, by defining message format templates as we described, we can be more precise than the SMTP standard specification and detect the (often subtle) differences between two dialects even though both might comply with the SMTP RFC. For example, we would build two different message format templates (and, therefore, have two dialects) for two clients that use different case for the EHLO keyword (e.g., one uses EHLO as a keyword, while the other uses Ehlo).

4.1.2 Learning the State Machine

We incrementally build the dialect state machine by starting from an empty initial state s_0 and adding new transitions and states as we observe more SMTP conversations from C . For each conversation Con in C , the algorithm executes the following steps:

Step 1: We set the current state s to s_0 .

Step 2: We examine all tuples $\langle r_i, c_i \rangle$ in Con . An example of a tuple is $\langle 220 \text{ server, HELO evil.com} \rangle$.

Step 3: We apply the algorithm described in 4.1.1 to r_i and c_i , and build the corresponding templates t_r and t_c . In the example, t_r is `220 hostname` and t_c is `HELO domain`. Note that c_i could be empty, because the client might not have sent any command after a reply from the server. In this case t_c will be an empty string.

Step 4: If the state machine has a state s_j labeled with t_c , we check if there is a transition t labeled with t_r going from s to s_j . (i) If there is one, we set the current state s to s_j , and go to Step 6. (ii) If there is no such transition, we connect s and s_j with a transition labeled with t_r , set the current state s to s_j , and go to Step 6. (iii) If none of the previous conditions hold, we go to Step 5.

Step 5: If there is no state labeled with t_c , we create a new state s_n , label it with t_c , and connect s and s_n with a transition labeled t_r . We then set the current state s to s_n . Following the previous example, if we have no state labeled with `HELO domain`, we create a new state with that label, and connect it to the current state s (in this case the initial state) with a transition labeled with `220 hostname`. If there are no tuples left in Con , and t_c is empty, we set the current state as a failure state for the current dialect, and add it to F_b . We then move to the next conversation in C , and go back to Step 2². Otherwise, we go to Step 6.

Step 6: If s is labeled with `DATA`, we mark the state as a good final state for this dialect, and add it to F_g . Else, if s is labeled with `QUIT`, we mark s as a bad final state and add it to F_b . We then move to the next conversation in C , and we go back to Step 2.

²By doing this, we handle cases in which the client abruptly terminates the connection

4.2 Collecting SMTP Conversations

To be able to model as much of a dialect as possible, we need a comprehensive set of SMTP conversations generated by a client.

As previously discussed, the straightforward approach to collect SMTP conversations is to passively observe the messages exchanged between a client and a server. In practice, this is often enough to uniquely determine the dialect spoken by a client (see Section 7 for experimental results). However, there are cases in which passive observation is not enough to uniquely identify a dialect. In such cases, it would be beneficial to be able to send specifically-crafted replies to a client (e.g., malformed replies), and observe its responses.

To perform this exploration, we set up a testing environment in which we direct clients to a mail server we control, and we instrument the server to be able to craft specific responses to the commands the client sends.

The SMTP RFCs define how a client should respond to unexpected SMTP replies, such as errors and malformed messages. However, both legitimate clients and spam engines either exhibit small differences in the implementation of these guidelines, or they do not implement them correctly. The reason for this is that implementing a subset of the SMTP guidelines is enough to be able to perform a correct conversation with a server and successfully send an email, in most cases. Therefore, there is no need for a client to implement the full SMTP protocol. Of course, for legitimate clients, we expect the SMTP implementation to be mature, robust, and complete — that is, corner cases are handled correctly. In contrast, spambots have a very focused purpose when using SMTP: send emails as fast as possible. For spammers, taking into account every possible corner case of the SMTP protocol is unnecessary; even more problematic, it could impact the performance of the spam engine (see Section 7.4 for more details).

In summary, we want to achieve two goals when actively learning an SMTP dialect. First, we want to learn how a client reacts to replies that belong to the language defined in the SMTP RFCs, but are not exposed during passive observation. Second, we want to learn how a client reacts to messages that are invalid according to the SMTP RFCs.

We aim to systematically explore the message structure as well as the state machine of the dialect spoken by a client. To this end, the variations to the SMTP protocol we use for active probing are of two types: (i) variations to the protocol state machine, which modify the sequence or the number of the replies that are sent by the server; and (ii) variations to the replies, which modify the structure of the reply messages that are sent by the server.

In the following, we discuss how we generate variations of both the protocol state machine and the replies.

Protocol state machine variations. We use four types of protocol variation techniques:

Standard SMTP replies: These variations aim at exposing responses to replies that comply with the RFCs, but are not observable during a standard, successful SMTP conversation, like the one in Figure 1. An example is sending SMTP errors to the commands a client sends. Some dialects continue the conversation with the server even after receiving a critical error.

Additional SMTP replies: These variations add replies to the SMTP conversation. More precisely, this technique replies with more than one message to the commands the client sends. Some dialects ignore the additional replies, while others will only consider one of the replies.

Out-of-order SMTP replies: These variations are used to analyze how a client reacts when it receives a reply that should not be sent at that point in the protocol (i.e., a state transition that is not defined by the standard SMTP state machine). For example, some senders might start sending the email content as soon as they receive a 354 reply, even if they did not specify the sender and recipients of the email yet.

Missing replies: These variations aim at exposing the behavior of a dialect when the server never sends a reply to a command.

Message format variations. These variations represent changes in the format of the replies that the server sends back to a client. As described in Section 2, SMTP server replies to a client's command have the format `CODE TEXT<CR><LF>`, where `CODE` represents the actual response to the client's command, `TEXT` provides human-readable information to the user, and `<CR><LF>` is the line terminator. According to the SMTP specification, a client should read the data from the server until it receives a line terminator, parse the code to check the response, and pass the text of the reply to the user if necessary (e.g., in case an error occurred).

Given the specification, we craft reply variations in four distinct ways to systematically study how a client reacts to them:

Compliant replies: These reply variations comply with the SMTP standard, but are seldom observed in a common conversation. For example, this technique might vary the capitalization of the reply (uppercase/lowercase/mixed case). The SMTP specification states that reply text should be case-insensitive.

Incorrect replies: The SMTP specification states that reply codes should always start with one of the digits 2, 3, 4, or 5 (according to the class of the status code), and be three-digits long. These variations are replies that do not comply with the protocol (e.g., a message with a re-

ply code that is four digits long). A client is expected to respond with a QUIT command to these malformed replies, but certain dialects behave differently.

Truncated replies: As discussed previously, the SMTP specification dictates how a client is supposed to handle the replies it receives from the server. Of course, it is not guaranteed that clients will follow the specification and process the entire reply. The reason is that the only important information the client needs to analyze to determine the server’s response is the status code. Some dialects might only check for the status code, discarding the rest of the message. For these reasons, we generate variations as follows: For each reply, we first separate it into tokens as described in Section 3.1. Then, for each token, we generate N different variations, where N is the number of tokens in each reply. We obtain such variations by truncating the reply with a line terminator after each token.

Incorrectly-terminated replies: From a practical point of view, there is no need for a client to parse the full reply until it reaches the line terminator. To assess whether a dialect checks for the line terminator when receiving a reply, we terminate the replies with incorrect terminators. In particular, we use the sequences <CR>, <LF>, <CR><CR>, and <LF><LF> as line terminators. For each terminator, similar to what we did for truncated replies, we generate $4N$ different variations of each reply, by truncating the reply after every token.

We developed 228 variations to use for our active probing. More precisely, we extracted the set of replies that are contained in the Postfix³ source code. Then, we applied to them the variations described in this section, and we injected them into a reference SMTP conversation. To this end, we used the sequence of server replies from the conversation in Figure 1.

5 Matching Conversations to Dialects

After having learned the SMTP dialects for different clients, we obtain a different state machine for each client. Given a conversation between a client and a server, we want to assess which dialect the client is speaking. To do this, we merge all inferred dialect state machines together into a single *Decision State Machine* \mathbf{M}_D .

5.1 Building the Decision State Machine

We use the approach proposed by Wolf [46] to merge the dialect state machines into a single state machine. Given two dialects \mathbf{D}_1 and \mathbf{D}_2 , the approach works as follows:

Step 1: We build the Cartesian product $\mathbf{D}_1 \times \mathbf{D}_2$. That is, for each combination of states $\langle s_1, s_2 \rangle$, where s_1 is a

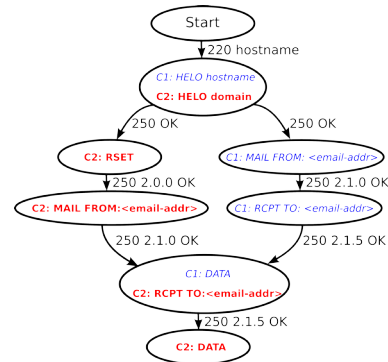


Figure 4: An example of decision state machine

state in \mathbf{D}_1 and s_2 is a state in \mathbf{D}_2 , we build a new state s_D in the decision state machine \mathbf{M}_D .

The label of s_D is a table with two columns. The first column contains the identifier of one of the dialects s_D was built from (e.g., \mathbf{D}_1), and the second column contains the label that dialect had in the original state (either s_1 or s_2). Note that we add one row for each of the two states that s_D was built from. For example, the second state of the state machine in Figure 4 is labeled with a table containing the two possible message templates that the clients C1 and C2 would send in that state (i.e., HELO hostname and HELO domain).

We then check all the incoming transitions to s_1 and s_2 in the original state machines \mathbf{D}_1 and \mathbf{D}_2 . For each combination of transitions $\langle t_1, t_2 \rangle$, where t_1 is an incoming transition for s_1 and t_2 is an incoming transition for s_2 , we check if t_1 and t_2 have the same label. If they do, we generate a new transition t_d , and add it to \mathbf{M}_D . The label of t_d is the label of t_1 and t_2 . The start state of t_d is the Cartesian product of the start states of t_1 and t_2 , respectively, while the end state is s_D . If the labels of s_1 and s_2 do not match, we discard t_d . For example, a transition t_1 labeled as 250 OK and a transition t_2 labeled as 553 Relaying Denied would not generate a transition in \mathbf{M}_D . At the end of this process, if s_D is not connected to any other state, it will be not part of the decision state machines \mathbf{M}_D , since that state would not be reachable if added to \mathbf{M}_D .

Step 2: We reduce the number of states in \mathbf{M}_D by merging together states that are *equivalent*. To evaluate if two states s_1 and s_2 are equivalent, we first extract the set of incoming transitions to s_1 and s_2 . We name these sets I_1 and I_2 . Then, we extract the set of outgoing transitions from s_1 and s_2 , and name these sets O_1 and O_2 . We consider s_1 and s_2 as equivalent if $|I_1| = |I_2|$ and $|O_1| = |O_2|$, and if the edges in the sets I_1 and I_2 , and in O_1 and O_2 have the exact same labels.

If s_1 and s_2 are equivalent, we remove them from \mathbf{M}_D , and we add a state s_d to \mathbf{M}_D . The label for s_d is a table composed of the combined rows of the label tables of s_1 and s_2 . We then adjust all the transitions in \mathbf{M}_D that

³A popular open-source Mail Transfer Agent: <http://www.postfix.org/>

had s_1 or s_2 as start states to start from s_d , and all the transitions that had s_1 or s_2 as end states to end at s_d .

We iteratively run this algorithm on all the dialects we learned, and we build the final decision state machine \mathbf{M}_D . As an example, Figure 4 shows the decision state machine built from the two dialects in Figure 2. Wolf shows how this algorithm produces nearly-minimal resulting state machines [46]. Empirical results indicate that this works well in practice and is enough for our purposes. Also, as for the dialect state machines, the decision state machine is non-deterministic. This is not a problem, since we analyze different states in parallel to make a decision as we explain in the next section.

5.2 Making a Decision

Given an SMTP conversation Con , we assign it to an SMTP dialect by traversing the decision state machine \mathbf{M}_D in the following way:

Step 1: We keep a list A of active states, and a list C_D of dialect candidates. At the beginning of the algorithm, A only contains the initial state of \mathbf{M}_D , while C_D contains all the learned dialects.

Step 2: Every time we see a server reply r in Con , we check each state s_a in A for outgoing transitions labeled with r . If such transition exists, we follow each of them and add the end states to a list A' . Then, we set A' as the new active state list A .

Step 3: Every time we see a client command c in Con , we check each state s_a in A . If s_a 's table has an entry that matches c , and the identifier for that entry is in the dialect candidate list C_D , we copy s_a to a list A' . We then remove from C_D all dialect candidates whose table entry in s_a did not match c . We set A' as the new active state list A .

The dialects that are still in C_D at the end of the process are the possible candidates the conversation belongs to. If C_D contains a single candidate, we can make a decision and assign the conversation to a unique dialect.

5.3 Applying the Decision

The decision approach explained in the previous section can be used in different ways, and for different purposes. In particular, we can use it to assess to which client a server is talking. Furthermore, we can use it for spam mitigation, and close connections whenever a conversation matches a dialect spoken by a bot.

Similarly to what we discussed in Section 4, the decision process can happen passively, or actively, by having a server decide which replies to send to the client. In the first case, we traverse the decision state machine for each reply, as described in Section 5.2, and end up with a dialect candidate set at the end of the conversation. Consider, for example, the decision state machine in Figure 4. By passively observing the SMTP conver-

sation, our approach is able to discard one of the two dialects from the candidate set as soon as the client sends the HELLO message. If the commands of the remaining candidate match the ones in the decision state machine for that client until we observe the DATA command, we can attribute the conversation to that dialect. Otherwise, the conversation does not belong to any learned dialect.

As discussed in Section 4, passive observation gives no guarantee to uniquely identify a dialect. In this context, a less problematic use case is to deploy this approach for spam detection: once the candidate set C_D contains only bots, we can close the connection and classify this conversation as related to spam. As we will show in Section 7, this approach works well in practice on a real-world data set. If passive observation is not enough to identify a dialect, one can use active probing.

Gain heuristic. To perform active detection, we need to identify “good” replies that we can send to achieve our purpose (dialect classification or spam mitigation). More specifically, we need to find out which replies can be used to expose the deviations in different implementations. To achieve this goal, we use the following heuristic: For each state c_i in which a dialect i reaches the end of a conversation (i.e., sends a DATA or QUIT command, or just closes the connection), we assign a *gain* value g_i to the dialect i in that state. The gain value represents how much it would help achieve our detection goal if we reached that state during our decision process. Then, we propagate the gain values backwards along the transitions of the decision state machine. For each state s , we set the gain for i in that state as the maximum of the gain values for i that have been propagated to that state. To correctly handle loops, we continue propagating the gain values until we reach a fixed point. We then calculate the gain for s as the minimum of the gains for any dialect j in s . We do this to ensure that our decision is safe in the worst-case scenario (i.e., for the client with the minimal gain for that state). We calculate the initial gain for a state in different ways, depending on the goal of our decision process.

When performing spam mitigation, we want to avoid a legitimate client from failing to send an email. For this reason, we strongly penalize failure states for legitimate clients, while we want to have high gains for states in which spambots would fail. For each state in which a dialect reaches a final state, we calculate the gain for that state as follows: First, we assign a score to each client with a final label for that state (i.e., a QUIT, a DATA, or a connection closed label). We want to give more importance to states that make bots fail, while we never want to visit states that make legitimate clients fail. Also, we want to give a neutral gain to states that make legitimate clients succeed, and a slightly lower gain to states that

make bots succeed. To achieve this, we assign a score of 1 for bot failure states, a score of 0 for legitimate clients failure states, a score of 0.5 for legitimate-client success states, and a score of 0.2 for bot success states. Notice that what we need here is a lattice of values that respect the stated precedence; therefore, any set of numbers that maintain this relationship would work.

When performing classification, we want to be as aggressive as possible in reducing the number of possible dialect candidates. In other words, we want to have high gains for states that allow us to make a decision on which dialect is spoken by a given client. Such states are those with a single possible client in them, or with different clients, each one with a different command label. To achieve this property, we set the gain for each state that includes a final label as $G = \frac{d}{n}$, where n is the total number of labels in that state, and d is the number of unique labels.

Reply selection. At each iteration of the algorithm explained in Section 5.2, we decide which reply to send by evaluating the gain for every possible reply from the states in A . For all the states reachable in one transition from the states in A , we first select the states S_a that still have at least an active client in their label table. We group together those states in S_a that are connected to the active states by transitions with the same label. For each label group, we pick the minimum gain among the states in that group. We consider this number as the gain we would get by sending that reply. After calculating the gain for all possible replies, we send the reply that has the highest gain associated to it. In case more than one reply yields the same gain we pick one randomly.

6 The Botnet Feedback Mechanism

Modern spamming botnets typically use *template-based spamming* to send out emails [22, 31, 38]. With this technique, the botnet C&C infrastructure tells the bots what kind of emails to send out, and the bots relay back information about the delivery as they received it from the SMTP server. This server feedback is an important piece of information to the botmaster, since it enables him to monitor if his botnet is working correctly.

Of course, a legitimate sender is also interested in information about the delivery process. However, she is interested in different information compared to the botmaster. In particular, a legitimate user wants to know whether the delivery of her emails failed (e.g., due to a typo in the email address). In such a case, the user wants to correct the mistake and send the message again. In contrast, a spammer usually sends emails in batches, and typically does not care about sending an email again in case of failure.

Nonetheless, there are three main pieces of information related to server feedback that a rational spammer is interested in: (i) whether the delivery failed because the IP address of the bot is blacklisted; (ii) whether the delivery failed because of specific policies in place at the receiving end (e.g., greylisting); (iii) whether the delivery failed because the recipient address does not exist. In all three cases, the spammer can leverage the information obtained from the mail server to make his operation more effective and profitable. In the case of a blacklisted bot, he can stop sending spam using that IP address, and wait for it to be whitelisted again after several hours or days. Empirical evidence suggests that spammers already collect this information and act accordingly [38]. If the recipient server replied with an SMTP non-critical error (i.e., the ones used in greylisting), the spammer can send the email again after some minutes to comply with the recipient's policy.

The third case, in which the recipient address does not exist, is the most interesting, because it implies that the spammer can *permanently* remove that email address from his email lists, and avoid using it during subsequent campaigns. Recent research suggests that bot feedback is an important part of a spamming botnet operation. For example, Stone-Gross et al. [38] showed that about 35% of the email addresses used by the *Cutwail* botnet were in fact non-existent. By leveraging the server feedback received by the bots, a rational botmaster can get rid of those non-existing addresses, and optimize his spamming performance significantly.

Breaking the Loop: Providing False Responses to Spam Emails. Based on these insights, we want to study how we can manipulate the SMTP delivery process of bots to influence their sending behavior. We want to investigate what would happen if mail servers started giving erroneous feedback to bots. In particular, we are interested in the third case, since influencing the first two pieces of information has only a limited, short-term impact on a spammer. However, if we provide false information about the status of a recipient's address, this leads to a double bind for the spammer: on the one hand, if a spammer considers server feedback, he will remove a valid recipient address from his email list. Effectively, this leads to a reduced number of spam emails received at this particular address. On the other hand, if the spammer does not consider server feedback, this reduces the effectiveness of his spam campaigns since emails are sent to non-existent addresses. In the long run, this will significantly degrade the freshness of his email lists and reduce the number of successfully sent emails. In the following, we discuss how we can take advantage of this situation.

As a first step, we need to identify that a given SMTP conversation belongs to a bot. To this end, a mail server

can either use traditional, IP-based blacklists or leverage the analysis of SMTP dialects introduced previously. Once we have identified a bot, a mail server can (instead of closing the connection) start sending erroneous feedback to the bot, which will relay this information to the C&C infrastructure. Specifically, the mail server could, for example, report that the recipient of that email does not exist. By doing this, the email server would lead the botmaster to the lose-lose situation discussed before. For a rational botmaster, we expect that this technique would reduce the amount of spam the email address receives. We have implemented this approach as a second instance of our technique to leverage the email delivery for spam mitigation and report on the empirical results in Section 7.3.

7 Evaluation

In this section, we evaluate the effectiveness of our approach. First, we describe our analysis environment. Then, we evaluate both the dialects and the feedback manipulation techniques. Finally, we analyze the limitations and the possible evasion techniques against our system.

7.1 Analysis Environment

We implemented our approach in a tool, called `B@bel`. `B@bel` runs email clients (legitimate or malicious) in virtual machines, and applies the learning techniques explained in Section 4 to learn the SMTP dialect of each client. Then, it leverages the learned dialects to build a decision machine M_D , and uses it to perform malware classification or spam mitigation.

The first component of `B@bel` is a virtual machine zoo. Each of the virtual machines in the zoo runs a different *email client*⁴. Clients can be legitimate email programs, mail transfer agents, or spambots.

The second component of `B@bel` is a gateway, used to confine suspicious network traffic. Since the clients that we run in the virtual machines are potentially malicious, we need to make sure that they do not harm the outside world. To this end, while still allowing the clients to connect to the Internet, we use restricting firewall rules, and we throttle their bandwidth, to make sure that they will not be able to launch denial of service attacks. Furthermore, we sinkhole all SMTP connections, redirecting them to local mail servers under our control.

We use three different mail servers in `B@bel`. The first email server is a regular server that speaks plain SMTP, and will perform passive observation of the client's SMTP conversation. The second server is instru-

⁴We used VirtualBox as our virtualization environment, and Windows XP SP3, Windows Server 2008, Windows 7, Ubuntu Linux 11.10, or Mac OS X Lion as operating systems on the virtual machines, depending on the operating system needed to run each of the legitimate clients or MTAs. We used Windows XP SP3 to run the malware samples

mented to perform active probing, as described in Section 4.2. Finally, the third server is configured to always report to the client that the recipient of an email does not exist, and is used to study how spammers use the feedback they receive from their bots.

The third component of `B@bel` is the learner. This component analyzes the active or passive observations generated between the clients in the zoo and the mail servers, learns an SMTP dialect for each client, and generates the decision state machine using the various dialects as input, as explained in Section 5. According to the task we want to perform (dialect classification or spam mitigation), `B@bel` tags the states in the decision state machine with the appropriate gain.

The last component of `B@bel` is the decision maker. This component analyzes an SMTP conversation, by either passively observing it or by impersonating the server, and makes a decision about which dialect is spoken by the client, using the process described in Section 5.2.

7.2 Evaluating the Dialects

Evaluating Dialects for Classification We trained `B@bel` by running active probing on a variety of popular Mail User Agents, Mail Transfer Agents, and bot samples. Table 1 lists the clients we used for dialect learning. Since we are extracting dialects by looking at the SMTP conversations only, `B@bel` is agnostic to the family a bot belongs to. However, for legibility purposes, Table 1 groups bots according to the most frequent label assigned by the anti-virus products deployed by VirusTotal [44]. Our dataset contained 13 legitimate MUAs and MTAs, and 91 distinct malware samples⁵. We picked the spambot samples to be representative of the largest active spamming botnets according to a recent report [26] (the report lists *Lethic*, *Cutwail*, *Mazben*, *Cutwail*, *Tedroo*, *Bagle*). We also picked worm samples that spread through email, such as *Mydoom*. In total, the malware samples we selected belonged to 11 families. The dialect learning phase resulted in a total of 60 dialects. We explain the reason for the high number of discovered dialects later in this section.

We then wanted to assess whether a dialect (i.e., a state machine) is unique or not. For each combination of dialects $\langle d_1, d_2 \rangle$, we merged their state machines together as explained in Section 5.1. We consider two dialects as distinct if any state of the merged state machine has two different labels in the label table for the dialects d_1 and d_2 , or if any state has a single possible dialect in it.

The results show that the dialects spoken by the legitimate MUAs and MTAs are distinct from the ones spo-

⁵The MD5 checksums of the malware samples are available at <http://cs.ucsb.edu/~gianluca/files/babel.txt>

Mail User Agents	Mail Transfer Agents	Bots (by AV labels)
Eudora, Opera, Outlook 2010, Outlook Express, Pegasus, The Bat!, Thunderbird, Windows Live Mail	Exchange 2010, Exim, Postfix, Qmail, Sendmail	Waledac, Donbot, Grum, Klez Buzus, Bagle, Lethic, Cutwail, Mydoom, Mazben, Tedroo

Table 1: MTAs, MUAs, and bots used to learn dialects.

ken by the bots. By analyzing the set of dialects spoken by legitimate MUAs and MTAs, we found that they all speak distinct dialects, except for Outlook Express and Windows Live Mail. We believe that Microsoft used the same email engine for these two products.

The 91 malware samples resulted in 48 unique dialects. We manually analyzed the spambots that use the same dialect, and we found that they always belong to the same family, with the exception of six samples. These samples were either not flagged by any anti-virus at the time of our analysis, or match a dropper that downloaded the spambot at a later time [8]. This shows that `B@bel` is able to classify spambot samples by looking at their email behavior, and label them more accurately than anti-virus products.

We then wanted to understand the reason for the high number of dialects we discovered. To this end, we considered clusters of malware samples that were talking the same dialect. For each cluster, we assigned a label to it, based on the most common anti-virus label among the samples in the cluster. All the clusters were unique, with the exception of eleven clusters marked as Lethic and two clusters marked as Mydoom. By manual inspection, we found that Lethic randomly closes the connection after sending the EHLO message. Since our dialect state machines are nondeterministic, our approach handles this case, in principle. However, in some cases, this nondeterministic behavior made it impossible to record a reply for a particular test case during our active probing. We found that each cluster labeled as Lethic differs for at most five non-recorded test cases with every other Lethic cluster. This gives us confidence to say that the dialect spoken by Lethic is indeed unique. For the two clusters labeled as Mydoom, we believe this is a common label assigned to unknown worms. In fact, the two dialects spoken by the samples in the clusters are very different. This is another indicator that `B@bel` can be used to classify spamming malware in a more precise fashion than is possible by relying on anti-virus labels only.

Evaluating Dialects for Spam Detection To evaluate how the learned dialects can be used for spam detection, we collected the SMTP conversations for 621,919 email messages on four mail servers in our department, spanning 40 days of activity.

For each email received by the department servers, we extracted the SMTP conversation associated with it, and then ran `B@bel` on it to perform spam detection. To this

end, we used the conversations logged by the *Anubis* system [4] during a period of one year (corresponding to 7,114 samples) to build the bot dialects, and the dialects learned in Section 7.2 for MUAs and MTAs as legitimate clients. In addition, we manually extracted the dialects spoken by popular web mail services from the conversations logged by our department mail servers, and added them to the legitimate MTAs dialects. Note that, since the goal of this experiment is to perform passive spam detection, learning the dialects by passively observing SMTP conversations is sufficient.

During our experiment, `B@bel` marked any conversation as spam if, at the end of the conversation, the dialects in C_D were all associated with bots. Furthermore, if the dialects in C_D were all associated with MUAs or MTAs, `B@bel` marked the conversation as legitimate (ham). If there were both good and malicious clients in C_D , `B@bel` did not make a decision. Finally, if the decision state machine did not recognize the SMTP conversation at all, `B@bel` considered that conversation as spam. This could happen when we observe a conversation from a client that was not in our training set. As we will show later, considering it as spam is a reasonable assumption, and is not a major source of false positives.

In total, `B@bel` flagged 260,074 conversations as spam, and 218,675 as ham. For 143,170 emails, `B@bel` could not make a decision, because the decision process ended up in a state where there were both legitimate clients and bots in C_D .

To verify how accurate our decisions were, we used a number of techniques. First, we checked whether the email was blocked by the department mail servers in the first place. These servers have a common configuration, where incoming emails are first checked against an IP blacklist, and then against more expensive content-analysis techniques. In particular, these servers used a commercial blacklist for discarding emails coming from known spamming IP addresses, and SpamAssassin and ClamAV for content analysis. Any time one of these techniques and `B@bel` agreed on flagging a conversation as spam, we consider this as a true positive of our system. We also consider as a true positive those conversations `B@bel` marked as spam, and that lead to an NXDOMAIN or to a timeout when we tried to resolve the domain associated to the sender email address. In addition, we checked the sender IP address against 30 addi-

tional IP blacklists⁶, and considered any match as a true positive. According to this ground truth, the true positive rate for the emails B@bel flagged as being sent by bots is 99.32%. Surprisingly, 98% of the 24,757 conversations that were not recognized by our decision state machine were flagged as spam by existing methods. This shows that, even if the set of clients from which B@bel learned the dialects from is not complete, there are no widely-used legitimate clients we missed, and that it is safe to consider any conversation generated by a non-observed dialect as spam. For the remaining 2,074 emails that B@bel flagged as spam, we could not assess if they were spam or not. They might have been a false positive of B@bel, or a false negative of the existing methods. To remain on the safe side, we consider them as false positives. This results in B@bel having a precision of 99.3%.

We then looked at our false negatives. We consider as false negatives those conversations that B@bel classified as belonging to a legitimate client dialect, but that have been flagged as spam by any of the previously mentioned techniques. In total, the other spam detection mechanisms flagged 71,342 emails as spam, among the ones that B@bel flagged as legitimate. Considering these emails as false negatives, this results in B@bel having a false negative rate of 21%. The number of false negatives might appear large at first. However, we need to consider the sources of these spam messages. While the vast majority of spam comes from botnets, spam can also be sent by dedicated MTAs, as well as through misused web mail accounts. Since B@bel is designed to detect email clients, we are able to detect which MTA or web mail application the email comes from, but we cannot assess whether that email is ham or spam. To show that this is the case, we investigated these 71,342 messages, which originated from 7,041 unique IP addresses. Assuming these are legitimate MTAs, we connected to each IP address on TCP port 25 and observed greeting messages for popular MTAs. For 3,183 IP addresses, one of the MTAs that we used to learn the dialects responded. The remaining 3,858 IP addresses did not respond within a 10 second timeout. We performed reverse DNS lookups on these IP addresses and assessed whether their assigned DNS names contained indicative names such as smtp or mail. 1,654 DNS names were in this group. We could not find any conclusive proof that the remaining 2,204 addresses belong to legitimate MTAs.

For those dialects for which B@bel could not make a decision (because the conversation lead to a state where both one or more legitimate clients and bots were active),

⁶The blacklists we leveraged come from these services: Barracuda, CBL, Spamhaus, Atma, Spamcop, Manitu, AHBL, DroneBL, DShield, Emerging Threats, malc0de, McAfee, mdl, OpenBL, SORBS, Sucuri Security, TrendMicro, UCEPROTECT, and ZeusTracker. Note that some services provide multiple blacklists

we investigated if we could have assessed whether the client was a bot or not by using active probing. Since the spambot and legitimate client dialects that we observed are disjoint, this is always possible. In particular, B@bel found that it is always possible to distinguish between the dialects spoken by a spambot and by a legitimate email client that look identical from passive analysis by sending a single SMTP reply. For example, the SMTP RFC specifies that multi-line replies are allowed, in the case all the lines in the reply have the same code, and all the reply codes but the last one are followed by a dash character. Therefore, multi-line replies that use different reply codes are not allowed by the standard. We can leverage different handling of this corner case to disambiguate between Qmail and Mydoom. More precisely, if we send the reply 250-OK<CR><LF>550 Error, Qmail will take the first reply code as the right one, and continue the SMTP transaction, while Mydoom will take the second reply code as the right one, and close the connection. Based on these observations, we can say that if we ran B@bel in active mode, we could distinguish between these ambiguous cases, and make the right decision. Unfortunately, we could run B@bel only in passive mode on our department mail servers.

Our results show that B@bel can detect (and possibly block) spam emails sent by bots with high accuracy. However, B@bel is unable to detect those spam emails sent by dedicated MTAs or by compromised webmail accounts. For this reason, similar to the other state-of-the-art mitigation techniques, B@bel is not a silver bullet, but should be used in combination with other anti-spam mechanisms. To show what would be the advantage of deploying B@bel on a mail server, we studied how much spam would have been blocked on our department server if B@bel was used in addition to or in substitution to the commercial blacklist and the content analysis systems that are currently in use on those servers.

Similarly to IP blacklists, B@bel is a lightweight technique. Such techniques are typically used as a first spam-mitigation step to make quick decisions, as they avoid having to apply resource-intensive content analysis techniques to most emails. For this reason, the first configuration we studied is substituting the commercial blacklist with B@bel. In this case, 259,974 emails would have been dropped as spam, instead of the 219,726 that were blocked by the IP blacklist. This would have resulted in 15.5% less emails being sent to the content analysis system, reducing the load on the servers. Moreover, the emails detected as spam by B@bel and the IP blacklist do not overlap completely. For example, the IP blacklist flags as spam emails sent by known misused MTAs. Therefore, we analyzed the amount of spam that the two techniques could have caught if used together. In this scenario, 278,664 emails would have been blocked,

resulting in 26.8% less emails being forwarded to the content analysis system compared to using the blacklist alone. As a last experiment, we studied how much spam would have been blocked on our servers by using B@bel in combination with both the commercial blacklist and the content analysis systems. In this scenario, 297,595 emails would have been flagged as spam, which constitutes an improvement of 3.9% compared to the servers' original configuration.

7.3 Evaluating the Feedback Manipulation

To investigate the effects of wrong server feedback to bots, we set up the following experiment. We ran 32 malware samples from four large spamming botnet families (*Cutwail*, *Lethic*, *Grum*, and *Bagle*) in a controlled environment, and redirected all of their SMTP activity to the third mail server in the B@bel architecture. We configured this server to report that *any* recipient of the emails the bots were sending to was non-existent, as described in Section 7.1.

To assess whether the different botnets stopped sending emails to those addresses, we leveraged a *spamtrap* under our control. A spamtrap is a set of email addresses that do not belong to real users, and, therefore, collect only spam mails. To evaluate our approach, we leverage the following idea: if an email address is successfully removed from an email list used by a spam campaign, we will not observe the same campaign targeting that address again. We define as a spam campaign the set of emails that share the same URL templates in their links, similar to the work of Xie et al. [48]. While there are more advanced methods to detect spam campaigns [31], the chosen approach leads to sufficiently good results for our purposes.

We ran our experiment for 73 days, from June 18 to August 30, 2011. During this period, our mail server replied with false server feedback for 3,632 destination email addresses covered by our spamtrap, which were targeted by 29 distinct spam campaigns. We call the set of campaigns C_f and the set of email addresses S_f . Of these, five campaigns never targeted the addresses for which we gave erroneous feedback again. To estimate the probability P_c that the spammer running campaign c in C_f actually removed the addresses from his list, and that our observation is not random, we use the following formula:

$$P_c = 1 - \left(1 - \frac{n}{t_f - t_b}\right)^{t_e - t_f},$$

where n is the total number of emails received by S_f for c , t_f is the time at which we first gave a negative feedback for an email address targeted by c , t_b is the first email for c which we ever observed targeting our spam trap, and t_e is the last email we observed for c . This formula calculates the probability that, given a certain

number n of emails observed for a certain campaign c , no email was sent to the email addresses in S_f after we sent a poisoned feedback for them. We calculate P_c for the five campaigns mentioned above. For three of them, the confidence was above 0.99. For the remaining two, we did not observe enough emails in our spamtrap to be able to make a final estimate.

To assess the impact we would have had when sending erroneous feedback to all the addresses in the spamtrap, we look at how many emails the whole spamtrap received from the campaigns in C_f . In total, 2,864,474 emails belonged to campaigns in C_f . Of these, 550,776 belonged to the three campaigns for which we are confident that our technique works and reduced the amount of spam emails received at these addresses. Surprisingly, this accounts for 19% of the total number of emails received, indicating that this approach could have impact in practice.

We acknowledge that these results are preliminary and provide only a first insight into the large-scale application of server feedback poisoning. Nevertheless, we are confident that this approach is reasonable since it leads to a lose-lose situation for the botmaster, as discussed in Section 6. We argue that the uncertainty about server feedback introduced by our method is beneficial since it reduces the amount of information a spammer can obtain when sending spam.

7.4 Limitations and Evasion

Our results demonstrate that B@bel is successful in detecting current spambots. However, spam detection is an adversarial game. Thus, once B@bel is deployed, we have to expect that spammers will evolve and try to bypass our systems. In this section, we discuss potential paths for evasion.

Evading dialects detection. The most immediate path to avoid detection by dialects is to implement an SMTP engine that precisely follows the specification. Alternatively, a bot author could make use of an existing (open source) SMTP engine that is used by legitimate email clients. We argue that this has a negative impact on the effectiveness and flexibility of spamming botnets.

Many spambots are built for performance; their aim is to distribute as many messages as possible. In some cases, spambots even send multiple messages without waiting for any server response. Clearly, any additional checks and parsing of server replies incurs overhead that might slow down the sender. We performed a simple experiment to measure the speed difference between a malware program sending spam (*Bagle*) and a legitimate email library on Windows (*Collaboration Data Objects - CDO*). We found that *Bagle* can send an email every 20 ms to a local mail server. When trying to send emails as fast as possible using the Windows library

(in a tight loop), we measured that a single email required 200 ms, an order of magnitude longer. Thus, when bots are forced to faithfully implement large portions of the SMTP specification (because otherwise, active probing will detect differences), spammers suffer a performance penalty.

Spammers could still decide to adopt a well-known SMTP implementation for their bots, run a full, parallelized, SMTP implementation, or revert to a well-known SMTP library when they detect that the recipient server is using B@bel for detection. In this case, another aspect of spamming botnets has to be taken into account. Typically, cyber criminals who infect machines with bots are not the same as the spammers who rent botnets to distribute their messages. Modern spamming botnets allow their customers to customize the email headers to mimic legitimate clients. In this scenario, B@bel could exploit possible discrepancies between the email client identified by the SMTP dialect and the one announced in the body of an email (for example, via the X-Mailer header). When these two dialects do not match (and the SMTP dialect does not indicate an MTA), we can detect that the sender pretends to speak a dialect that is inconsistent with the content of the (spam) message. Of course, the botmasters could take away the possibility for their customers to customize the headers of their emails, and force them to match the ones typical of a certain legitimate client (e.g., Outlook Express). However, while this would make spam detection harder for B@bel, it would make it easier for other systems that rely on email-header analysis, such as *Botnet Judo* [31], because spammers would be less flexible in the way they vary their templates.

Mitigating feedback manipulation. As we discussed in Section 6, spammers can decide to either discard any feedback they receive from the bots, or trust this feedback. To avoid this, attackers could guess whether the receiving mail server is performing feedback manipulation. For example, when all emails to a particular domain are rejected because no recipient exists, maybe all feedback from this server can be discarded. In this case, we would need to update our feedback mechanism to return invalid feedback only in a fraction of the cases.

8 Related Work

Email spam is a well-known problem that has attracted a substantial amount of research over the past years. In the following, we briefly discuss how our approach is related to previous work in this area and elaborate on the novel aspects of our proposed methods.

Spam Filtering: Existing work on spam filtering can be broadly classified in two categories: *post-acceptance methods* and *pre-acceptance methods*. Post-acceptance

methods receive the full message and then rely on *content analysis* to detect spam emails. There are many approaches that allow one to differentiate between spam and legitimate emails: popular methods include Naive Bayes, Support Vector Machines (SVMs), or similar methods from the field of machine learning [16, 27, 35, 36]. Other approaches for content-based filtering rely on identifying the URLs used in spam emails [2, 48]. A third method is *DomainKeys Identified Mail* (DKIM), a system that verifies that an email has been sent by a certain domain by using cryptographic signatures [23]. In practice, performing content analysis or computing cryptographic checksums on every incoming email can be expensive and might lead to high load on busy servers [41]. Furthermore, an attacker might attempt to bypass the content analysis system by crafting spam messages in specific ways [25, 28]. In general, the drawback of post-acceptance methods is that an email has to be received before it can be analyzed.

Pre-acceptance methods attempt to detect spam before actually receiving the full message. Some analysis techniques take the *origin* of an email into account and analyze distinctive features about the sender of an email (e.g., the IP address or autonomous system the email is sent from, or the geographical distance between the sender and the receiver) [17, 34, 39, 43]. In practice, these sender-based techniques have coverage problems: previous work showed how IP blacklists miss detecting a large fraction of the IP addresses that are actually sending spam, especially due to the highly dynamic nature of the machines that send spam (typically botnets) [32, 37, 38].

Our method is a novel, third approach that focuses on *how* messages are sent. This avoids costly content analysis, and does not require the design and implementation of a reputation metric or blacklist. In contrast, we attempt to recognize the SMTP dialect during the actual SMTP transaction, and our empirical results show that this approach can successfully discriminate between spam and ham emails. This complements both pre-acceptance and post-acceptance approaches. Another work that went in this direction was done by Beverly et al. [5] and Kakavelakis et al. [19]. The authors of these two papers leveraged the fact that spambots have often bad connections to the Internet, and perform spam detection by looking at TCP-level features such as retransmissions and connection resets. Our system is more robust, because it does not rely on assumptions based on the network connectivity of a mail client.

Moreover, to the best of our knowledge, we are the first to study the effects of manipulating server feedback to poison the information sent by a bot to the botmaster.

Protocol Analysis: The core idea behind our approach is to learn the SMTP dialect spoken by a particular client. This problem is closely related to the problem of

automated protocol reverse-engineering, where an (unknown) protocol is analyzed to determine the individual records/elements and the protocol's structure [6, 13]. Initial work in this area focused on clustering of network traces to group similar messages [14], while later methods extracted protocol information by analyzing the execution of a program while it performs network communication [10, 15, 24, 45, 47]. Sophisticated methods can also handle multiple messages and recover the protocol's state machine. For example, *Dispatcher* is a tool capable of extracting the format of protocol messages when having access to only one endpoint, namely the bot binary [9]. Cho et al. leverage the information extracted by *Dispatcher* to learn C&C protocols [11]. Brumley et al. studied how deviations in the implementation of a given protocol specification can be used to detect errors or generate fingerprints [7]. The differences in how a given program checks and processes inputs are identified with the help of binary analysis (more specifically, symbolic execution).

Our problem is related to previous work on protocol analysis, in the sense that we extract different SMTP protocol variations, and use these variations to build fingerprints. However, in this work, we treat the speaker of the protocol (the bot) as a blackbox, and we do not perform any code analysis or instrumentation to find protocol formats or deviations. This is important because (i) malware is notoriously difficult to analyze and (ii) we might not always have a malware sample available. Instead, our technique allows us to build SMTP dialect state machines even when interacting with a previously-unknown spambot.

There is also a line of research on fingerprinting protocols [12, 30, 49]. Initial work in this area leveraged manual analysis. Nonetheless, there are methods, such as *FiG*, that automatically generate fingerprints for DNS servers [42]. The main difference between our work and *FiG* is that our dialects are stateful while *FiG* operates on individual messages. This entirely avoids the need to merge and explore protocol state machines. However, as discussed previously, individual messages are typically not sufficient to distinguish between SMTP engines.

9 Conclusion

In this paper, we introduced a novel way to detect and mitigate spam emails that complements content- and sender-based analysis methods. We focus on *how* email messages are sent and derive methods to influence the spam delivery mechanism during SMTP transactions. On the one hand, we show how small deviations in the SMTP implementation of different email agents (so called *SMTP dialects*) allow us to detect spambots during the actual SMTP communication. On the other hand, we study how the feedback mechanism used by botnets

can be poisoned, which can be used to have a negative impact on the effectiveness of botnets.

Empirical results confirm that both aspects of our approach can be used to detect and mitigate spam emails. While spammers might adapt their spam-sending practices as a result of our findings, we argue that this reduces their performance and flexibility.

Acknowledgments

This work was supported by the Office of Naval Research (ONR) under Grant N000140911042, the National Science Foundation (NSF) under grants CNS-0845559 and CNS-0905537, by Secure Business Austria, and by the German Federal Ministry of Education and Research under grant 01BY1111 / MoBE. We want to thank our shepherd Alex Moshchuk and the anonymous reviewers for their valuable comments, and Andreas Boschke for his help in setting up some of our experiments.

References

- [1] RFC 821: Simple Mail Transfer Protocol. <http://tools.ietf.org/html/rfc821>.
- [2] SURBL URI reputation data. <http://www.surbl.org/>.
- [3] The Spamhaus Project. <http://www.spamhaus.org>.
- [4] BAYER, U., MOSER, A., KRUEGEL, C., AND KIRDA, E. Dynamic analysis of malicious code. *Journal in Computer Virology* 2, 1 (2006), 67–77.
- [5] BEVERLY, R., AND SOLLINS, K. Exploiting Transport-level Characteristics of Spam. In *Collaboration, Electronic messaging, Anti-Abuse and Spam Conference (CEAS)* (2008).
- [6] BORISOV, N., BRUMLEY, D., WANG, H. J., DUNAGAN, J., JOSHI, P., AND GUO, C. Generic Application-Level Protocol Analyzer and its Language. In *Symposium on Network and Distributed System Security (NDSS)* (2007).
- [7] BRUMLEY, D., CABALLERO, J., LIANG, Z., NEWSOM, J., AND SONG, D. Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation. In *USENIX Security Symposium* (2007).
- [8] CABALLERO, J., GRIER, C., KREIBICH, C., AND PAXSON, V. Measuring Pay-per-Install: The Commoditization of Malware Distribution. In *USENIX Security Symposium* (2011).
- [9] CABALLERO, J., POOSANKAM, P., KREIBICH, C., AND SONG, D. X. *Dispatcher*: Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-Engineering. In *ACM Conference on Computer and Communications Security (CCS)* (2009).
- [10] CABALLERO, J., YIN, H., LIANG, Z., AND SONG, D. X. Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. In *ACM Conference on Computer and Communications Security (CCS)* (2007).
- [11] CHO, C. BABIC, D. S. D. Inference and Analysis of Formal Models of Botnet Command and Control Protocols. In *ACM Conference on Computer and Communications Security (CCS)* (2010).
- [12] COMER, D. E., AND LIN, J. C. Probing TCP Implementations. In *USENIX Summer Technical Conference* (1994).
- [13] COMPARETTI, P. M., WONDRAČEK, G., KRUEGEL, C., AND KIRDA, E. Prospex: Protocol Specification Extraction. In *IEEE Symposium on Security and Privacy* (2009).
- [14] CUI, W., KANNAN, J., AND WANG, H. J. Discoverer: Automatic Protocol Reverse Engineering from Network Traces. In *USENIX Security Symposium* (2007).

- [15] CUI, W., PEINADO, M., CHEN, K., WANG, H. J., AND IRUNBRIZ, L. Tupni: automatic reverse engineering of input formats. In *ACM Conference on Computer and Communications Security (CCS)* (2008).
- [16] DRUCKER, H., WU, D., AND VAPNIK, V. N. Support vector machines for spam categorization. In *IEEE transactions on neural networks* (1999).
- [17] HAO, S., SYED, N. A., FEAMSTER, N., GRAY, A. G., AND KRASSER, S. Detecting Spammers with SNARE: Spatio-temporal Network-level Automatic Reputation Engine. In *USENIX Security Symposium* (2009).
- [18] JOHN, J. P., MOSHCHUK, A., GRIBBLE, S. D., AND KRISHNAMURTHY, A. Studying Spamming Botnets Using Botlab. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2009).
- [19] KAKAVELAKIS, G., BEVERLY, R., AND J., Y. Auto-learning of SMTP TCP Transport-Layer Features for Spam and Abusive Message Detection. In *USENIX Large Installation System Administration Conference* (2011).
- [20] KANICH, C., WEAVER, N., MCCOY, D., HALVORSON, T., KREIBICH, C., LEVCHENKO, K., PAXSON, V., VOELKER, G., AND SAVAGE, S. Show Me the Money: Characterizing Spam-advertised Revenue. *USENIX Security Symposium* (2011).
- [21] KASPERSKY LAB. Spam Report: April 2012. https://www.securelist.com/en/analysis/204792230/Spam_Report_April_2012, 2012.
- [22] KREIBICH, C., KANICH, C., LEVCHENKO, K., ENRIGHT, B., VOELKER, G. M., PAXSON, V., AND SAVAGE, S. On the Spam Campaign Trail. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)* (2008).
- [23] LEIBA, B. DomainKeys Identified Mail (DKIM): Using digital signatures for domain verification. In *Collaboration, Electronic messaging, Anti-Abuse and Spam Conference (CEAS)* (2007).
- [24] LIN, Z., JIANG, X., XU, D., AND ZHANG, X. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. In *Symposium on Network and Distributed System Security (NDSS)* (2008).
- [25] LOWD, D., AND MEEK, C. Good word attacks on statistical spam filters. In *Collaboration, Electronic messaging, Anti-Abuse and Spam Conference (CEAS)* (2005).
- [26] M86 LABS. Security labs report. http://www.m86security.com/documents/pdfs/security_labs/m86_security_labs_report_2h2011.pdf, 2011.
- [27] MEYER, T., AND WHATELEY, B. SpamBayes: Effective open-source, Bayesian based, email classification system. In *Collaboration, Electronic messaging, Anti-Abuse and Spam Conference (CEAS)* (2004).
- [28] NELSON, B., BARRENO, M., CHI, F. J., JOSEPH, A. D., RUBINSTEIN, B. I. P., SAINI, U., SUTTON, C., TYGAR, J. D., AND XIA, K. Exploiting Machine Learning to Subvert Your Spam Filter. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2008).
- [29] PATHAK, A., HU, Y. C., AND MAO, Z. M. Peeking into spammer behavior from a unique vantage point. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)* (2008).
- [30] PAXSON, V. Automated Packet Trace Analysis of TCP Implementations. In *ACM SIGCOMM Conference* (1997).
- [31] PITSILLIDIS, A., LEVCHENKO, K., KREIBICH, C., KANICH, C., VOELKER, G. M., PAXSON, V., WEAVER, N., AND SAVAGE, S. botnet Judo: Fighting Spam with Itself. In *Symposium on Network and Distributed System Security (NDSS)* (2010).
- [32] RAMACHANDRAN, A., DAGON, D., AND FEAMSTER, N. Can DNS-based blacklists keep up with bots? In *Collaboration, Electronic messaging, Anti-Abuse and Spam Conference (CEAS)* (2006).
- [33] RAMACHANDRAN, A., AND FEAMSTER, N. Understanding the Network-level Behavior of Spammers. *SIGCOMM Comput. Commun. Rev.* 36 (August 2006).
- [34] RAMACHANDRAN, A., FEAMSTER, N., AND VEMPALA, S. Filtering Spam with Behavioral Blacklisting. In *ACM Conference on Computer and Communications Security (CCS)* (2007).
- [35] SAHAMI, M., DUMAIS, S., HECKERMAN, D., AND HORVITZ, E. A Bayesian approach to filtering junk e-mail. *Learning for Text Categorization* (1998).
- [36] SCULLEY, D., AND WACHMAN, G. M. Relaxed Online SVMs for Spam Filtering. In *ACM SIGIR Conference on Research and Development in Information Retrieval* (2007).
- [37] SINHA, S., BAILEY, M., AND JAHANIAN, F. Shades of Grey: On the Effectiveness of Reputation-based “Blacklists”. In *International Conference on Malicious and Unwanted Software* (2008).
- [38] STONE-GROSS, B., HOLZ, T., STRINGHINI, G., AND VIGNA, G. The Underground Economy of Spam: A Botmaster’s Perspective of Coordinating Large-Scale Spam Campaigns. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)* (2011).
- [39] STRINGHINI, G., HOLZ, T., STONE-GROSS, B., KRUEGEL, C., AND VIGNA, G. BotMagnifier: Locating Spammers on the Internet. In *USENIX Security Symposium* (2011).
- [40] SYMANTEC CORP. State of spam & phishing report. http://www.symantec.com/business/theme.jsp?themeid=state_of_spam,2010.
- [41] TAYLOR, B. Sender reputation in a large webmail service. In *Collaboration, Electronic messaging, Anti-Abuse and Spam Conference (CEAS)* (2006).
- [42] VENKATARAMAN, S., CABALLERO, J., POOSANKAM, P., KANG, M. G., AND SONG, D. X. FiG: Automatic Fingerprint Generation. In *Symposium on Network and Distributed System Security (NDSS)* (2007).
- [43] VENKATARAMAN, S., SEN, S., SPATSCHECK, O., HAFFNER, P., AND SONG, D. Exploiting Network Structure for Proactive Spam Mitigation. In *USENIX Security Symposium* (2007).
- [44] VIRUSTOTAL. Free Online Virus, Malware and URL Scanner. <https://www.virustotal.com/>.
- [45] WANG, Z., JIANG, X., CUI, W., WANG, X., AND GRACE, M. ReFormat: Automatic Reverse Engineering of Encrypted Messages. In *European Symposium on Research in Computer Security (ESORICS)* (2009).
- [46] WOLF, W. An Algorithm for Nearly-Minimal Collapsing of Finite-State Machine Networks. In *IEEE International Conference on Computer-Aided Design (ICCAD)* (1990).
- [47] WONDRAČEK, G., COMPARETTI, P. M., KRUEGEL, C., AND KIRDA, E. Automatic Network Protocol Analysis. In *Symposium on Network and Distributed System Security (NDSS)* (2008).
- [48] XIE, Y., YU, F., ACHAN, K., PANIGRAHY, R., HULTEN, G., AND OSIPKOV, I. Spamming Botnets: Signatures and Characteristics. *SIGCOMM Comput. Commun. Rev.* 38 (August 2008).
- [49] ZALEWSKI, M. p0f v3. <http://lcamtuf.coredump.cx/p0f3/>, 2012.
- [50] ZHUANG, L., DUNAGAN, J., SIMON, D. R., WANG, H. J., AND TYGAR, J. D. Characterizing Botnets From Email Spam Records. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)* (2008).

Impact of Spam Exposure on User Engagement

Anirban Dasgupta[†], Kunal Punera[‡], Justin M. Rao[⊖], Xuanhui Wang[§]

[†]*Yahoo! Labs, Sunnyvale CA*

anirban@yahoo-inc.com

[‡]*RelateIQ Inc., Palo Alto, CA*

kunal.punera@utexas.edu

[⊖]*Microsoft Research, New York, NY*

justin.rao@microsoft.com

[§]*Facebook, Menlo Park CA*

xuanhui@gmail.com

Abstract

In this paper we quantify the effect of unsolicited emails (spam) on behavior and engagement of email users. Since performing randomized experiments in this setting is rife with practical and moral issues, we seek to determine causal relationships using observational data, something that is difficult in many cases. Using a novel modification of a user matching method combined with a time series regression on matched user pairs, we develop a framework for such causal inference that is particularly suited for the spam exposure use case. Using our matching technique, we objectively quantify the effect that continued exposure to spam has on user engagement in Yahoo! Mail. We find that indeed spam exposure leads to significantly, both statistically and economically, lower user engagement. The impact is non-linear; large changes impact users in a progressively more negative fashion. The impact is the strongest on “voluntary” categories of engagement such as composed emails and lowest on “responsive” engagement metrics. Our estimation technique and results not only quantify the negative impact of abuse, but also allow decision makers to estimate potential engagement gains from proposed investments in abuse mitigation.

1 Introduction

Over the last several years, as email has steadily become the dominant mode of text-based online communication, unsolicited bulk email, generally referred to as “email-spam” or simply “spam”, has increased in lockstep [33]. By some estimates the total fraction of all emails that can be considered spam is higher than 90% [33, 10]. Moreover, while email-spam began as a way for unscrupulous marketers to advertise their products, it has now become the main vector for phishing [4, 14], installing malware, and stealing information [22]. In short, email-spam has morphed from being a mild irritant to an outright danger

to the users.

This has led to major efforts both in the industry and the research community to develop better spam filters [5, 12, 13, 39, 40]. However, spammers are known to quickly adapt their email messages in order to circumvent these filters [16]. This has resulted in an adversarial game of “cat-and-mouse” between email service providers (ESPs) and spammers: (1) Spammers send out bulk emails designed to bypass the spam filters of major email service providers; (2) In time, spam filters adapt using machine learning and crowdsourcing techniques and block the offending emails; (3) Spammers re-tune message content, change the sending locations and so forth, and the cycle continues. This results in email-spam reaching user inboxes for the duration between the bulk mails being sent and the spam-filters adapting. Unfortunately, even though filters have improved dramatically, spam is so cheap to send that the required conversation rates for profitability, which are below 1 in 5 million, can still be sustained [22].

Barring some fundamental change in the spam market (such as legal or technological solutions), the chief way to combat spam is to invest more resources to make the spammers’ response cycle less economically viable, which would force some spammers out of the market. Characterizing this ecosystem is thus essential not just for making both policy decisions but also in making decisions that on the surface seem to be purely machine learning in nature—e.g. how to design spam filters that exploit signatures that are the hardest to game.

Although qualitative arguments about spam being a negative social externality have been often made, it is much harder to quantify the intuited numbers [1, 21, 27]. Since botnets form the main spam-delivery infrastructure, researchers interested in understanding the economics of spam have made significant efforts in understanding the market behind the creation and renting of botnets [32, 41, 3]. Kanich et al. [23] measure how successful product-oriented spam ultimately is in marketing

and selling the corresponding products. Similar studies have provided quantitative estimates on the economics of account phishing [17], the market behind “human-farms” [31] and malware distributions [6]. Rao and Reiley [34] review a large fraction of this literature from an economic perspective. Such quantitative studies have collectively thrown valuable light on various aspects of the underground economy, thereby providing guidance to both the policy-designers and designers of spam-filters.

Given this extensive literature, it is perhaps surprising that seemingly little attention has been paid to the interplay between email users and email service providers and the associated responses to problems of email-spam. For example, we are not aware of any work that quantifies the long-term effects of spam reaching the inbox on user engagement. In terms of the interplay, changes in user engagement have a direct impact on ESP revenue and are thus an important decision metric for anti-spam investment. Economic theory tells us that a profit maximizing firm will invest in anti-spam technology only if there is a compensating return in terms of increased user engagement or retention. For instance, simply because we all think spam is a bad thing does not mean service providers will go broke fighting it! Being able to provide a quantitative estimate on how the long-term user engagement is affected as a result of spam would provide an added concrete incentive for the ESP to fight spam.

Some econometric studies [7, 42] have approached the problem from the firm perspective (the client of the email provider) and have shown that spam has a significant cost in terms of the working time spent by users in dealing with email. In particular, Caliendo et al. [7] use a survey approach and find that the average employee in their sample spent 1200 minutes per year in dealing with spam. However, these small-scale studies cannot quantify the effect of spam on *longer-term user engagement*. Does getting more spam cause a user to stop using the email service? It seems intuitive to assume “yes”. However, it has never been established whether this causal effect exists, *how strong the effect is* if it exists, *what types of engagement* would it affect, and *how to measure this in a statistically robust manner*. More explicitly, answering these questions is useful for multiple reasons—it helps our broad understanding of the total negative externality of spam, which could potentially have implications in deciding how to deal with spam at the policy-level. Also, as spam filters get better, making additional improvements in spam catch-rate becomes harder and hence more expensive, and often involves difficult trade-offs either regarding total investment or about false-positive rates (i.e. in deciding the operating point of the spam classifiers). In terms of social efficiency spam is clearly a negative [34]—the consensus view is that spam should be mitigated far below current levels in

order to raise social welfare because the social costs of spam clearly outweigh the monetary returns from spamming. However, since the government cannot compel ESPs to invest more heavily in anti-spam technology, obtaining estimates of the negative impact of spam, such as ones in this paper, is important. Accurately quantifying the impact of spam allows firms to make informed, well-targeted investments. In turn, these investments can potentially lead to improvements in service quality for the end-users. While our study does not provide authoritative answers to all these questions, it certainly builds many of the tools and the necessary formalizations for it.

The gold standard for estimating causal effects is randomized experimentation, also referred to as “A/B testing” [24]. If we can expose users to spam completely at random, then we can safely assume that any effect we observe is due to spam. In the real world, however, performing such experimentation is difficult because exposing users to spam is problematic for both user experience and the ESP’s reputation. Estimating causal effects is typically difficult in the absence of randomized experiments because most actions reflect something about the user in terms of their type or future intentions. These circumstances lead to the classic problem of correlation in the absence of causation. For example, since users tend to get spam when they give out their emails to third party services and active users tend to do so more often than less active users, a naive plot of engagement-vs-spam would show activity and spam exposure being positively correlated.

An alternate method of estimating such effects is to conduct in-depth surveys or in-lab tests of a smaller set of users. In-lab methods are inadequate for our problem as we are looking to estimate potentially small, but long-term effects. The size of the surveys or lab studies is necessarily limited by cost, which makes it hard to estimate small and long-term effects. More importantly, what users report in a survey may not be reflected in their actual behavior. In particular, rounding error can severely bias estimates. For example, answering in a survey that one spends 5 minutes a day dealing with spam might seem like a “small” amount, but over the course of a year, that is 1250 minutes, or about 20 hours. For a \$30 an hour employee, this means it is a \$600 per year problem. If the true value was 1.5 minutes, but the user rounded up, the resulting estimate could be off by a wide margin.

An extensive literature in econometrics has focused on developing techniques such covariate matching, regression-coefficient methods, bias reduction, neighbor matching, propensity score matching (PSM) etc. [35, 20, 9, 29] to deal with selection bias in observational data. Among these, PSM and neighbor matching techniques are considered more robust in estimating effects

of a categorical treatment variable [29] than regression-coefficient methods—in both of these the intuition is to be able to match a untreated user with a treated one based on a set of pre-defined user attributes. PSM creates the matching using only a single propensity score that is obtained by a weighted combination of the user attributes—the weights are learnt by modeling the exposure treatment as a categorical variable directly using a first stage logistic (or similar) regression. For nearest neighbor methods user matching is done by treating them as points in a high dimensional space. It is commonly believed that PSM is more robust than nearest neighbor matching methods when the number of user attributes is large since finding nearest neighbors in high dimensions is not robust (see e.g. [29] for detailed discussion). Yet, for PSM one has to assume that the first stage regression is correctly specified. The non-parametric nature of nearest neighbor matching methods makes them more reliable with respect to the fact that one does not have to correctly specify a first stage regression—in small samples and with high data dimensionality, the benefits of PSM outweigh the drawbacks.

In our setting, the popularity of Yahoo! Mail gives us a huge set of users to match over, compared to the number of user attributes. Also, existing PSM methods typically assume the ability to model the probability that a particular user falls into the *categorical* “treatment” group. However, in our application, spam exposure is a continuous variable, leaving the treatment group ill-defined, and hence this assumption fails. For both reasons, the nearest neighbor matching is more appropriate in this setting.

In this paper we describe a large-scale *nearest neighbor matching* method to infer causal relationship from observational data for which the exposure is a continuous variable. We apply this technique to the spam-engagement setting. Overall the results provide strong empirical support for the commonsense notion that spam has a negative impact on user engagement. We provide quantitative estimates that show that the impact of spam in the inbox can have serious revenue implications and can contribute to a large percentage drop in user engagement. The effect is largest for more “volitional” user activities such as composing and sending emails. The function mapping spam changes to engagement appears to be convex, with the marginal impact increasing with the size of the exposure change. User characteristics are not particularly informative in predicting the response to spam — notably light users are equally affected in absolute terms by a piece of spam in the inbox, meaning that percentage-wise the impact is far greater for these users. Thus, although the intuition that spam causes decreased user engagement is commonplace, the main insight supplied by this study is to extend and formalize this intuition in a quantitative way.

Our Contributions.

- We conduct a principled and thorough study of the causal relationship between spam exposure and long-term user engagement. We find that, indeed, exposure to spam results in long-term reduction in user engagement in terms of logins, page views, and emails sent. As far as we know, this is the first such study to *quantitatively* establish this link between spam exposure and user engagement.
- We propose the use of a variant of propensity score matching, namely *nearest neighbor matching*, in combination with regression based techniques in establishing causal relationships in large-scale observational data settings when the exposure metric is continuous. This contribution of our paper is of interest *independent* of its particular application in this study. Our simulations (described in the Appendix) indicate that this method is indeed superior to (variants of) propensity score matching for continuous exposure metrics.

Organization. In Section 2 we present our approach for estimating causal relationships in large-scale observational data settings. Then in Section 3 we instantiate our proposed approach to the case study of estimating the effect of spam exposure on long-term user engagement. The results of this case study are given in Section 4. In Section 5 we review prior work in causality estimation and spam exposure studies. In Section 6 we conclude. Finally, in the Appendix we compare our proposed methodology with variants of propensity score matching and on simulated data show that our approach performs better at estimating a hidden relationship between variables.

2 Measuring the Effect of Spam on User Engagement

In this section, we first define the problem of estimating the effect of spam exposure on user engagement. We start with a description of the aspects of the problem that make it unique from other works in measuring effects. We then present a formalization of the continuous exposure setting and describe how to map our problem to this formalization.

2.1 Aspects of the Problem Setting

Our problem of measuring engagement as a function of spam exposure has the following characteristics that make it unique, and hence requiring modifications to established methodology.

Continuous Exposure: In our problem, the exposure variable is continuous—there is no clear definition of a “treatment” vs. “control” group. We cannot identify a set of users and consider them as “treated,” i.e. having been sufficiently exposed to spam because nearly everyone is exposed to some degree. One solution would be using an arbitrary threshold to define a treatment class. But in some sense this is just asking the same question back again: what is a critical level of spam such that a person receiving that amount can be considered to be sufficiently exposed? Thus, the continuous exposure is not just an artifact of the data, incorporating that into the modeling and estimating process is absolutely essential.

Engagement as a function of Exposure: Having defined exposure to be a continuous variable, computing a single number as the expected size of the effect is not meaningful any more. Instead we want to answer the following question: what is the expected effect if the amount of exposure is increased by an amount Δs . We intend to approximate the function that captures the change in the effect as a result of the change in the exposure for an average user.

Infeasibility of Randomized Testing: Randomized experiments are clearly the gold standard for measuring effects. Suppose we intend to estimate the effect on a user receiving Δs more spam messages in a month. Ideally, we would be able to select a small random set of users, and then tune their spam filters such that they receive Δs more spam for this month. We could then measure the resulting effect against a randomized control group.

For the spam-setting, however, performing such experimentation is difficult on many levels: (1) exposing users to spam is problematic from both a user experience and Yahoo!’s reputation point of view. The negative effects of spam does in fact often extend beyond a minor nuisance, since a majority of these messages contain URLs that tempt users to either conduct commercial transactions or to give out their personal information; (2) even if we could filter out the most pernicious types of spam, the revenue risk associated with user defection would cause the size of the study to be limited, both in terms of the amount of exposure and the number of users; 3) spam that does leak into inbox is, by definition, currently undetectable before the user has interacted with it. Thus, any randomized experiments would have to account for exposure of this kind anyway.

2.2 Formal Problem Definition

We now define the problem formally and point out the empirical quantities for which we would like to create unbiased estimators. Suppose for each user i , \mathbf{x}_i denotes the set of features we observe. Let s_i denote her exposure variable and y_i denote the response (or effect) variable.

Note that s_i is continuous. If we want to study the impact of spam on the user, then the exposure variable would be the amount of spam received by the user in a particular time period, the same for all users — we call this the *exposure period*. Abusing notation, we write $y(\mathbf{x}, s)$ to denote that the response is a function of the user features and the exposure. Let Δs denote a certain amount of change in the exposure variable, and $\Delta y(\Delta s)$ denote the function that measures the *average* change in y due to an increase Δs in the exposure. Formally, we define $\Delta y(\Delta s)$ as follows. Let $E[\cdot]$ denote the expectation operator.

$$\Delta y(\Delta s) = E_{(\mathbf{x}, s)}[y(\mathbf{x}, s + \Delta s) - y(\mathbf{x}, s)]. \quad (1)$$

The expectation in the above expression is taken over all the user features and all the previous value of exposure. This of course is not an observable quantity, since one user has only one value of s . Thus, a more feasible quantity to measure is the following – difference over pairs who differ only in exposure, but have the same feature vector.

$$\Delta y(\Delta s) = E_{i, i'}[y_i - y_{i'} | \exists(\mathbf{x}, s_i, y_i), (\mathbf{x}, s_{i'}, y_{i'}), s_i - s_{i'} = \Delta s] \quad (2)$$

Note how this quantity generalizes the effect measurement for binary treatment variables. If $s \in \{0, 1\}$, then the standard question of measuring the average treatment effect would be

$$\begin{aligned} & y(s = 1) - y(s = 0) \\ &= E_{\mathbf{x}}[y_i - y_{i'} | \exists(\mathbf{x}, s = 1, y_i), (\mathbf{x}, s = 0, y_{i'})] \end{aligned}$$

In our case, we are thus interested in the function $\Delta y(\Delta s)$ instead of a single value that measures the treatment vs. non-treatment. This makes the application of the standard propensity score matching techniques [35] impossible: we can no longer define a treatment class.

One naive way of creating the estimate would be to compute the following difference—essentially just take the differences in the effect levels of users whose exposure is s and those whose exposure is $s + \Delta s$.

$$f(\Delta s) = E_i[y_i | s_i = s] - E_i[y_i | s_i = s + \Delta s]$$

But this would be the wrong quantity, since conditioning on the fact $s_i = s + \Delta s$ is different from conditioning on $s_i = s$ (the corresponding distributions of \mathbf{x} and hence $y(\mathbf{x}, s)$ are different), and thus the above difference does not measure what would happen to the average person if the exposure suffered by that person increased by Δs .

Nearest Neighbor Matching. The essence of nearest neighbor matching is that we can approximate the equation 2 by the following one.

$$\Delta y(\Delta s) = E_{i, i'}[y_i - y_{i'} | s_i - s_{i'} = \Delta s, \mathbf{x} \approx \mathbf{x}'] \quad (3)$$

where $\mathbf{x} \approx \mathbf{x}'$ denotes that \mathbf{x} and \mathbf{x}' are approximately similar, instead of being exactly same. The variants of this definition of approximate similarity define the different variants of the nearest neighbor matching algorithm.

Suppose we have a particular matching function, in which, for each given user $i = (\mathbf{x}_i, s_i, y_i)$, we can find out a set of users N_i such that for each $j \in N(i)$ satisfies $\mathbf{x}_j \approx \mathbf{x}_i$. Further define $\mathbf{1}(X)$ to be the indicator vector for the event X , in particular let $\mathbf{1}(s, s')$ denote the indicator vector such that $|s - s'| = \Delta s$. If there are n users overall, our empirical estimator for the quantity in equation 3 is then given by

$$n(i, \Delta s) = \sum_i \sum_{j \in N(i)} \mathbf{1}(s_i, s_j)$$

$$\Delta y(\Delta s) = \frac{1}{n} \sum_i \frac{1}{n(i, \Delta s)} \sum_{j \in N(i)} \mathbf{1}(s_i, s_j)(y_i - y_j)$$

Essentially, in each neighborhood $N(i)$, we compute the average effect due to an increase of Δs exposure and then average these effects over all the points to get the average effect.

3 Data, Features and Matching for Spam

In this section we describe how to apply the above matching technique for the spam exposure case study. We start with a summary of our overall method. In order to measure how engagement is affected by spam exposure we first need to specify how to measure *user engagement* and *spam exposure* for a user. We then describe *how to create matchings* between users based on user behavior features.

3.1 Technique Summary

In order to measure the effect of spam exposure on user engagement we first create a set of behavioral features per user for a 2 month period, called the “matching period.” These features are then used to create matchings between users. We then observe the spam exposure of these users on the exposure month (month 3) immediately following the matching period. Due to random variation in spam, the two users in a match are often exposed to different amounts of spam (Δs). We then examine how Δs impacts behavior in the observation period immediately following the exposure month. We look at difference in engagement for both the short-run (only month 4) and long-run (months 5-6), while controlling for how these differences persisted within the pair (e.g. higher month 3 spam likely means higher month 4 spam; in estimating month 4 engagement, we will control for this difference).

The attribution of causality depends on the assumption that within each pair of users, month 3 spam exposure is random. This is known as the “selection on observables” assumption. In general, spam exposure is correlated with user activity. Using your account more actively tends to get the email address “out there” more, making exposure to spam non-random. For example, in a cross-section of users, light users tend to get less spam than heavy users. This is precisely the reason we need to use the matching methodology to estimate causal effects (and overcome spurious correlation). In our case, we match on both the level and linear trend of usage. So the identifying assumption stated more precisely is: conditional upon the level and trend of usage (on all 14 matching criteria) over two months, the spam exposure difference between users within a pair in the following month is related to future usage in only the following ways (a) the direct impact of past spam exposure; (b) the indirect impact of past spam exposure (higher spam today, might mean higher spam tomorrow, which we must control for).

3.2 Data Description and Matching Attributes

Our data comes from the Yahoo! Mail logs of user activity.¹ To ensure accurate results, we first cleaned the data of accounts that were potentially corrupted by phishing attempts or spambots. We dropped any user who showed a change in more than 4 sent messages a day (in average) between the matching months (months 1-2) and the target months. This number was chosen based on an analysis of the distribution to determine what qualified as an improbable outlier. We also dropped a pair of users that had a Euclidean match distance of greater than 0.1 to ensure that we were always very close matches. Finally, we dropped all users that showed near zero mail page views in the matching month(s) and outliers (+3 standard deviations). The former is to increase the strength of our estimator, as it is unreasonable to assume spam impacted a user that never logged in, the latter to reduce the influence of high leverage anomalies.

After performing all the cleaning operations, we took a large random sample of 500,000 users for 6 months, and generated the following features per user per day: *all inbound mail, classified spam, total sent mail, composed mail, replies, forwards, mail time spent, all page views on Yahoo! site, all time spent on Yahoo! site, delete without reading (messages that are removed from the inbox without reading), deletes, spam votes and non-spam*

¹Note that this is purely observational data, no active experimentation or bucket-testing was involved. Furthermore, we use only behavioral statistics aggregated at the anonymized user level. Thus there are no privacy issues related to email content, or the graph of user-user communication.

votes.

To ensure that the matching generated very similar users, we used all the 14 features over 2 months to compute nearest neighbors. In addition, we also ensured that the matched user accounts were registered in the same year. We performed the matching process over the entire mail sample, thus enabling a small enough distance threshold. As a result of the matching, we end up with 486,102 matched pairs (one user could be considered in multiple pairs, and not user-user pairs qualify for matching, as we see below). Using the first two months of data for the matching period ensures that each pair of users had the same level of usage and the same (first order linear) trend.

3.3 Metrics for User Engagement

Yahoo! Mail users interact with the web user interface in a variety of ways. Users can login into the interface and just glance at the list of emails in the various folders (“boxes”), can click on individual emails to open them in a separate panel for reading or delete it without reading. Other email related actions that are instrumented include replying to individual emails, or forwarding them, composing new emails and marking emails as spam or non-spam. Each of these actions represents a different kind of engagement, and naturally certain forms of engagement are more significant than the others. From a short-run revenue calculation perspective, the page view is the primary quantity of interest, as page views can be easily converted to a dollar figure based on the advertising monetization rate. But not all page views are created equal. For example, we have found that the number of sent mails (and resulting pageviews) is a more reliable predictor of future engagement than the pageviews resulting from simply reading mail or reloading one’s inbox. The reason is likely that sending mail both leads to more mail in response and signals that the user is using the account as her primary email. We thus look at a variety of such metrics to measure engagement.

3.4 Quantifying Spam Exposure

Yet another critical point in our study is how to quantify the spam exposure of a user. Typically, the spam that a user has been exposed to lands in her inbox does so precisely because the filters have been unable to recognize it as spam. Consequently, this number is hard to measure for a user. We could rely on the “spam votes” of a user a proxy for this quantity, but it is well known that very few users give any votes. In fact, the average Yahoo! Mail user gives less than one vote in an entire year, whereas some users are extremely proactive in marking emails as spam. To complicate matters, even spammers

and bot accounts give spam-votes, aiming to subvert the machine-learned filters by providing false examples.

The strategy available to us is to use the number of inbound emails classified by the Yahoo! filter as a measure of the spam targeted towards the user and infer “inbox exposure” from this classified spam. Of all delivered mail (not blocked before connection), more than half is classified as spam and sent to the spambox. The false negative rate relates the spambox quantity to implied inbox-exposure. For example, if the false negative rate is 0.10, then for every 9 messages in the spambox, we expect 1 piece of spam to slip into the inbox. For the empirical analysis, we estimate the false negative rate and use it to infer inbox-exposure, which we will use in all our analysis. Due to confidentiality concerns of Yahoo Inc., we cannot report the exact estimates of the false negative rate, but will describe the process through which we model and infer it.

Estimating the False Negative Rate: We estimate the false negative rate in two ways. First, we utilize daily usage logs of users over a 6 month period. Note that if the false negative rate were 0, then conditional on past behavior, daily spam box quantity should be unrelated to inbox quantity, because there is no slippage. In contrast if the rate is non-zero, increases in the spambox will be positively correlated with increases in the inbox. We estimate this relationship using a regression of inbox quantity on spambox quantity and lagged values of both quantities, all on the daily level. This gives us an estimate, lets call it *FN*.

To confirm this estimate, we examine how spambox levels correlate with “delete without reading” in the inbox. “Delete without reading” is a strong sign of spam, but many legitimate mails are deleted without reading as well. In fact 53% of all inbox messages are deleted in this fashion. If the false negative rate was 0, then there should not be a relationship between spambox and delete without reading, conditional on inbox volume (inbox volume and spambox volume could be related, so we control for this). We estimate the empirical relationship using a time-series regression and find that 1 message in the spam box leads to $.8FN$ deletes without reading. That is, very close to our initial estimate of the false negative rate using the other methodology and consistent with the idea that not all users simply delete spam, but most do. Given the mutual consistency of both approaches, we proceed with our estimate of the false negative rate in all analysis.

Maintained Assumptions on the False Negative Rate: The assumption of a constant false negative rate might seem too strong when we consider the fact that users have different propensities to sign-up for email mailing lists. In our analysis, however, the individual variations are less important for following reasons. First, we only use this estimate to normalize in the aggregate sense —

obtaining the aggregate inbox-spam in terms of the classified spam. Thus, in our case, all we require is that within a pair of users, there are no systematic differences in false negative rate; this is essentially assured by our bi-directional matching procedure. When examining the differential impact of large increases in exposure vs. small increases (non-linearities), the assumption requires that when a user experiences a large increase in spam, the classification rate stays the same. Indeed, given how machine classification benefits from large quantities, one might think that large quantities of spam are classified with less error. We will see that we actually find an increasing marginal impact of exposure, meaning that either this is not an issue, or the real pattern is even more convex.

The area that is most hampered by the constant false negative rate assumption is the analysis of user characteristics. For instance, if Yahoo! does a better job of classifying spam for older users, then we will overstate the inbox-exposure for these users. In the results section, we note these concerns where applicable.

3.5 Creating the Matching

In this section, we describe the method of nearest neighbor matching that we used. The basic framework is to match users who are very similar to each other in the matching period, and then analyze how their behaviors differ in subsequent time periods. We first discuss how to create the neighborhood set $N(i)$ for each user.

Using kNN for Matching: In order to define the matching, we use two criteria to define the neighborhoods $N(i)$ — a distance based threshold and a k -nearest neighbor based threshold. The distance between the vectors is measured in ℓ_2 norm. We have a distance threshold d that we use to filter our pairs that do not lie within d distance of each other. On top of this, we apply a k -nearest neighbor based threshold — each point i contains no more than k of its nearest neighbors in $N(i)$. This ensures that a dense region of the \mathbf{x} manifold is not over-represented in our estimate.

Using Bi-directional Matching: To avoid bias, we only use bi-directional matches. What this means is that dyad $i-j$ is only included in the analysis if i is j 's nearest neighbor and j is also i 's nearest neighbor. The nearest neighbor property is not generally bi-directional (i 's nearest neighbor might be j , but there is a node closer to j , say r , that is further from i). The most important reason we include only bi-directional pairs is that it ensures that in the exposure period, the average difference within a pair of users is 0 for all attributes we match on, by construction, because the labeling of users within the pair is purely nominal. In our estimation, this means that we can reliably link differences in spam exposure within

the pair to differences in engagement, knowing that there is no other reasons for a systematic difference.

An additional reason is that it naturally eliminates a known issue with matching or propensity score estimators that occurs when relatively few users are the “unexposed match” to relatively many exposed users. For instance, consider a job training analysis in which we predict the probability (propensity score) of receiving training. PSM matches a pair in which one person actually received training and one did not, but had similar predicted probabilities of receiving training. By construction, there are relatively few individuals who have a high predicted probability of receiving training but in reality do not receive it. This means that these people are the “controls” for a relatively large number of treated individuals, thus increasing the impact of their behavior on observed estimates. In our routine, we get around this problem by only using bi-directional matches. In our case, the problem that would arise is that some users in the less dense portion of the kNN graph match to users in a denser portion. These users in the less dense portion might be different in ways that induce bias (for instance if they are always slightly more engaged).

Using Locality Sensitive Hashing: Computing the matching efficiently for a large number of data points and a moderately large number of dimensions is a non-trivial task. In order to compute this, we utilize the locality sensitive hashing technique [2]. Essentially, the idea is to compute a hash function h such that the probability of two points falling into the same hash bucket is inversely proportional to the distance between them.

$$\Pr[h(i) \neq h(j)] \propto \|\mathbf{x}_i - \mathbf{x}_j\|$$

We first bucket all points using this hash function and then do an exhaustive search inside each bucket to find the k -nearest neighbors for each point that fall within the distance threshold. We tune our LSH construction such that with high probability we get all neighbors for all points within the distance threshold.

4 Empirical Results

In this section we present the results of our empirical application. We start by linearly modeling the short-run (1-month in the future) impact of spam exposure on the various metrics of webmail engagement. We then examine the effect more closely using a flexible non-linear model. Next, we examine how mail spam impacts non-mail usage of properties on the Yahoo! network of sites (contagion effects). We then proceed to estimate the medium-run (2–3 months) impact of spam exposure on future engagement. Finally we examine how user characteristics modulate the impact of spam.

4.1 Short-run Impact of Spam on Mail Engagement

Estimating Equation: In this subsection, we look at the impact of month 3 spam on month 4 engagement. Recall month 3 is our first post-match month, and thus the first time spam exposure will meaningfully vary within a pair of users. In our baseline specification, for each pair of users i , we estimate the following equation with robust ordinary least squares. Let y equal the engagement metric we are interested in (page views, sent mail, etc) and s the number of spam messages that reach an user’s inbox. Let the months be denoted by 1, 2.. etc. Let Δy_{it} , Δs_{it} denote the differences in the engagement and the exposure metric for the i^{th} user-pair for the t^{th} month. Recall that months 1 and 2 were used to find matching users (thus, the average Δy_{it} , Δs_{it} values are essentially zero for $t = 1, 2$). We run the following regression to estimate the relation between $\Delta y_{i,4}$ and $\Delta s_{i,3}$.

$$\Delta y_{i,4} = \beta \Delta s_{i,3} + \rho y_{i,4} + \gamma_1 \Delta y_{i,3} + \gamma_2 \Delta s_{i,4} + \gamma_3 \Delta s_{i,4}^2 + \gamma_4 \Delta s_{i,4}^3 + \varepsilon_i$$

This specification controls for month 4 spam exposure using a cubic polynomial and includes a lagged value of the dependent variable, to control for the contemporaneous impact of spam last month and activity bias (see [25]). β is the quantity of interest, as it gives the first order impact of spam exposure on engagement 1 month in the future. Table 1 gives the estimates of β for the our key engagement metrics.

Absolute Impact: As the results in Table 1 show, across all metrics, the relationship between exposure and engagement is consistent with the hypothesis that spam exposure discourages usage. That spam has a negative impact is perhaps obvious; however Table 1 gives a quantitative estimates for all metrics, not just the sign of the effect. In Column (1), we see that the impact of one spam message in the inbox reduces mail page views next month by 0.472 pageviews. For a webmail provider, page views are the primary metric to gage the revenue impact, as they can be converted to dollars based on the ad revenue from each page view. The R-squared numbers show that these regressors account typically account for 10% of the variation in the dependent variable.

However page views do not tell the whole story, as other metrics, such as sent mail, are thought to be better long-term predictors of engagement. In column (2), we estimate that a spam message in the inbox reduces webmail time spent next month by 24 seconds. Column 3 shows that about 1/4 of the page view impact comes through reading fewer messages. Column (4) shows sent mail impact. Sent mail includes composed emails (written from scratch), replies and forwards. Overall,

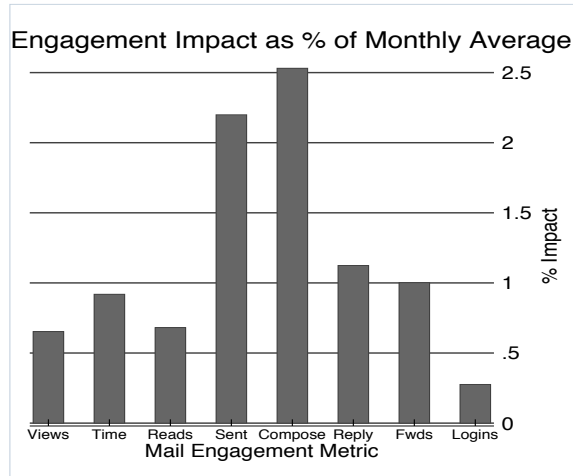


Figure 1: Differential impact of spam exposure magnitude on sent mail and mail page views.

users send much less mail than they receive or read, as mass/automated emails are a large fraction of legitimate email traffic as well. The impact on sent mail is negative with most of the impact coming through composed messages. This makes sense from a disengagement/frustration perspective. One still replies to emails, but perhaps looks for other communication outlets to send new messages if the account is inundated with spam. In Column 8, we see that spam leads to fewer session logins as well.

Impact as Percentage of Baseline Usage: In Figure 1, we show the relative size of the impact on each of the engagement metrics. We create this by converting the impact of 1 spam message in the inbox last month, estimated in Table 1, to percentages as a function of the averages for each metric in the matching months. The largest percentage impact occurs for composed messages, consistent with the story that this sort voluntary user engagement is the most susceptible to a negative experience. The percentage impact on composed emails is more than twice as large as the impact on replies and forwards. Monthly “consumption” metrics, views, time spent and reads, show between a 0.5–1% decline as a result of a spam message in the inbox. Logins show the lowest relative impact — although users engage less heavily after spam exposure, in general they still login to the webmail client with close to the same frequency.

4.2 Differential Impact by Exposure Change Size

In the previous section we modeled the impact of spam exposure as a linear function. This was mainly to facilitate interpretation and comparisons across engagement

	Exposure Metric							
	(1) Page Views	(2) Time	(3) Reads	(4) Sent	(5) Composed	(6) Reply	(7) Fwd	(8) Login
$\Delta s_{t-1} (\beta)$	-0.472*** (0.0236)	-24.20*** (1.614)	-0.108*** (0.0250)	-0.0305*** (0.00289)	-0.0251*** (0.00234)	-0.00326*** (0.000912)	-0.00104*** (0.000228)	-0.0572*** (0.010)
Δy_{t-1}	0.414*** (0.00703)	0.483*** (0.0185)	0.113*** (0.0263)	0.402*** (0.0741)	0.335*** (0.0923)	0.509*** (0.0341)	0.261*** (0.0140)	0.74*** (.0001)
R-squared	0.162	0.177	0.10	0.089	0.065	0.123	0.048	

Table 1: Impact of spam exposure on engagement 1-month in the future. Robust standard errors are in parentheses and *** means p-value < 0.01.

metrics. In this subsection we examine how the impact of the change in spam exposure depends on the *magnitude* of the change. To do so, we make use of the Frisch-Waugh theorem from linear regression [15]. We first regress the exposure metric on the control variables (the variables other than past spam difference) and then take the residual. We then regress the independent variable of interest, last month’s spam exposure, on the control variables, and take the residual. The relationship between the residuals of the dependent variable (engagement metrics) and the residuals of the independent variable (last month spam exposure) gives the relationship between these two variables, net of the impact of the control variables.

Non-linear Impact on Sent Mail, Logins and Mail Page Views:

In Figure 2 we plot the relationship using a local polynomial smoother (Epanechnikov kernel, bandwidth=10) for three key engagement metrics: sent mail (left axis), mail logins (left axis) and mail page views (right axis). All three metrics display the same pattern. The y-intercept at zero is almost exactly zero for all metrics, which is comforting, because it means that we (correctly) estimate that if a pair has no exposure difference, there is not an engagement difference. This can be seen as a confirmation of the validity of our matching procedure (we also do this via simulation runs in the following section). The slope close to zero is negative, but significantly less than the slope for large differences in exposure — relatively small changes in exposure tend to discourage engagement, but the impact is muted. For all metrics, at about 15 spam messages in the inbox in a one-month period, the negative impact shows a sharp increase (gets more negative). For sent emails and logins, this slope increase levels off near 25 spam messages, but for mail page views, the steep slope persists over all ranges of values for which we have sufficient data.

Key Takeaways: The differential impact in Figure 2 gives insight into how spam negatively impacts the user experience. Note that the x-axis in Figure 2 is the absolute difference in number of spam received by the two users in a pair over 1 month. Small changes in spam

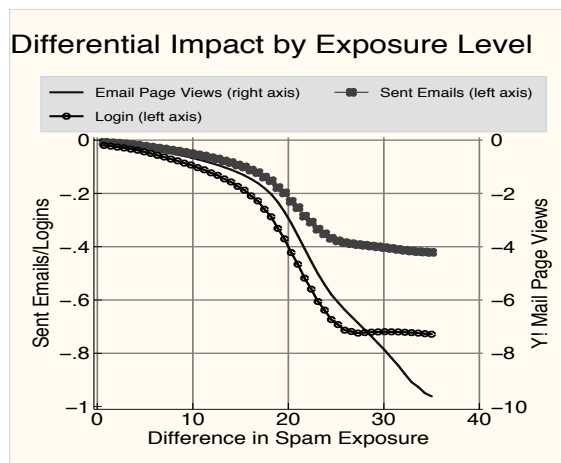


Figure 2: Differential impact of spam exposure magnitude on sent mail and mail page views.

exposure has a muted impact on the user, whereas large changes have a much more pronounced effect. When the increase in spam exposure reaches the level of once every other day, the marginal impact ticks up considerably. This disengagement is likely the result of a disruption of the user experience. Since small changes are less disruptive, the marginal effect is lower. One possible conclusion to draw from this nonlinear trend is the following: it is likely more worthwhile to make a relatively large investment for a big increase in filtration accuracy (and thus obtain a super-linear improvement in engagement), rather than pay a relatively modest sum for an incremental improvement.

4.3 Contagion effects

So far we have documented a negative impact of mail spam on many facets of webmail engagement provided a quantitative estimates the magnitudes. The next natural question is “Does exposure to online abuse in one domain carry over to engagement in a firm’s other web properties?” These so-called “contagion effects” or

	Aggregate Effect		Controlling for Mail	
	(1) Non-mail Page Views	(2) Non-mail Time Spent	(3) Non-mail Page Views	(4) Non-mail Time Spent
Contagion effect, Δs_{t-1}	-0.064** (0.03)	-4.33*** (0.03)	-0.0176 (1.72)	-1.470 (1.71)
Δ Mail page views t			0.117*** (0.003)	
Δ Mail time spent t				0.136*** (0.006)
Δy_{t-1}	0.639*** (0.028)	0.640*** (0.027)	0.711*** (0.055)	0.703*** (0.056)
R-squared	0.253	0.214	0.265	0.226

Table 2: Contagion effects of mail spam on other network activities. $p < 0.01$: ***, $p < 0.05$: **.

“brand damage effects” are often used as justification for investment in anti-abuse technology. Our empirical framework allows us to examine this question by looking at engagement across the Yahoo! network of sites.

Contagion Estimates: In Table 2 we estimate the impact of Yahoo! Mail spam on page views and time spent occurring on other parts of Yahoo!. In columns (1) and (2), we do not control for the contemporaneous impact on mail activity – this is why there are empty spaces for these regressors. The estimated contagion effects in this case are negative and statistically significant coming in around 17% (13%) of the direct effect magnitude for time-spent (resp. pageviews), as given in Table 1. In evaluating the revenue impact of a proposed change in the spam filter, these spillover effects should indeed be taken into account. However, to qualify as a pure contagion effect, we would want to be sure they are not mechanically due to lower Yahoo! Mail engagement. The reason is that Yahoo! Mail uses various techniques to get the user to engage with the rest of the Yahoo! network. For example, news stories are shown in the “welcome screen” and there is a web search bar. In column (3) and (4), we control for contemporaneous Yahoo! mail usage. Controlling for mail usage reduces the estimated impact of spam exposure by 80% – the remaining figures are no longer statistically significant. The conclusion is that while there measurable spillover effects, the direct cause seems to be lower mail engagement itself. Since mail use creates positive spillovers on the rest of the site, lowering mail engagement has a more than 1:1 effect on engagement. Once we control for this effect, nearly all of the supposed contagion effects go away.

Key Takeaways: Our conclusion is thus that while in the short term there are economically meaningful spillovers of mail spam on the non-mail network activity, the spillovers do not seem to be driven purely by contagion or brand-damage reasons. Rather, they seem to be more mechanically linked to the decreased mail engagement. This is not to say that contagion effects do not exist, just that in this case they are swamped by the direct neg-

ative impact. Our careful analysis allows us to separate these subtle differences.

4.4 Medium-run impact

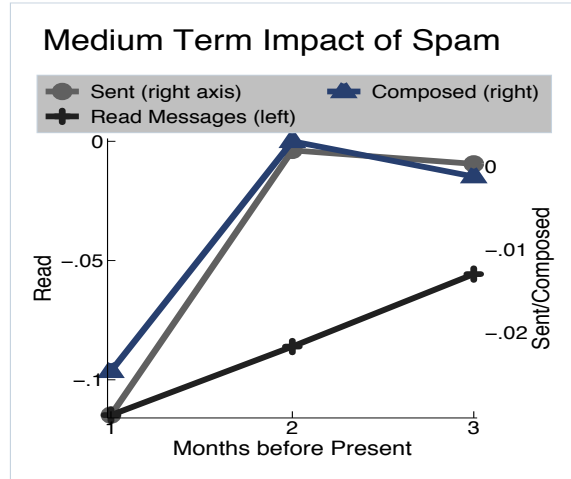


Figure 3: Direct impact of spam on future behavior 1–3 months post-exposure.

In this subsection we examine the impact of spam exposure on engagement up to 3 months in the future. In Figure 3 we plot the impact coefficient of spam exposure on sent mail, composed messages and read messages for the range of 1 to 3 months in the past. The estimates use the same specification as Table 1. The regressions control for any short-run impacts that have already occurred. For instance, in estimating the 3-month impact (impact of spam 3 months ago), we control for the immediate change in behavior this had (the short-run effect) by including lagged dependent variables in the regression. What this means is we are estimating the direct impact. For example, if the 2-month effect is estimated to be zero, say, this does not mean the effect goes away, it only means that there is no *additional* effect as compared to the 1-month impact.

Engagement Estimates: Examining Figure 3 a few trends are immediately clear. The first is that the effect decays over time. For sent mail and composed mail, the negative impact occurs entirely in the first month following exposure. Recall that percentage-wise, these two metrics saw the largest short-run declines. Evidently part of the reason for this is that the total impact is felt in the first month following exposure. The graph also confirms the analysis of the previous section that the impact on sent mail occurs primarily through composed messages, not replies or forwards. For reading messages, the decline is less steep as there is still significant impact 3-months out. We thus conclude that while spam can have a

	(1) Page Views	(2) Sent Mail	(3) Reads
1{Male}=1	-0.0037 (0.0029)	-0.00015 (0.0003)	-0.0065*** (0.0020)
1{New user}=1	-0.0107 (0.0076)	-9.63e-06 (0.0006)	-0.0014 (0.0053)
1{Light}=1	0.0036 (0.0029)	-0.0006** (0.0003)	0.0011 (0.0020)
1{Heavy}=1	-0.0027 (0.0038)	-0.0005 (0.0003)	-0.0013 (0.0027)
1{User <30}=1	-0.00194 (0.0030)	0.00123*** (0.0003)	0.0106*** (0.0020)
1{User >50}=1	-0.0090* (0.0051)	0.0009* (0.0005)	-0.0043 (0.0035)
1{High baseline exposure}=1	-0.0568 (0.0410)	-0.0007 (0.00369)	0.0605** (0.0286)
R-squared	0.162	0.089	0.010

Table 3: Differential impact of spam exposure by user characteristics. $p < 0.01$: ***, $p < 0.05$: **, $p < 0.1$: *.

direct impact on behavior up to 3-months down the road, this is not the case for “volitional” categories in which the initial impact is large, such as sent/composed mail.

4.5 Breakdown by user characteristics

In this subsection we augment the regression specification used in Table 1 by interacting dummy variables for user characteristics with spam exposure. The interaction terms give the differential impact of spam based on the characteristic in question. The results are summarized in Table 3. All of the characteristics except gender and user age (self-reported age of the user) were used in matching. For the two measures that were not used in matching, the indicator variable only equals 1 if both users fall under the designation. For example, the variable $1\{\text{User} < 30\}$ is defined as 1 if both users are under the age of 30. High baseline exposure is defined as being in the top 1/3 of spam exposure in the matching months. Light users are those that had page views in the bottom third during the matching months, heavy is top third. All other variables are self-explanatory.

Sent Mail and Page Views: We see that for sent mail and page views, user characteristics do not appear to predict the response to spam. However, the fact that heavy users do not show a higher *absolute* impact of spam exposure, means that percentage-wise, light users are the most adversely affected. Spam exposure is likely an important feature in retention, as it is known that decreased usage among light users is an important predictor of quitting.

Reading Messages: For reading messages, we find that

the impact is significantly larger for males (more negative) and smaller for young (in calendar age) users. Users with higher baseline spam exposure respond slightly less to changes in spam exposure, however as we noted, this analysis is tenuous because we assume that spam classification accuracy is not a function of past exposure, when in reality it might be, due to user votes, for instance.

Takeaways: Overall we do not see major difference in the impact of spam based on user characteristics. The most notable result is that the percentage impact is highest for light users.

5 Related Work

There are two broad classes of existing works related to our research. On the methodology side, our work is related to the traditional causality methods literature. On the application side, our work is related to those quantifying the impact of spam. While we cannot cover every work here, we will mention some key works from each side in order to put our paper in context.

Estimating Causality: The study of causality has been an active area for many years. In particular, our work is developed within the framework of causal models developed by Rubin in early 1970s [36]. Our method of matching users by covariates or features is based on the theory developed in [36, 37]. The major steps that distinguish us from this work are the combined use of the matching and the regression to adapt this technique to the continuous setting, the use of criterion such as nearest neighbor matching, bi-directional matching, and locality sensitive hashing to speed up the computation. The propensity score matching method (PSM) uses the propensity score (predicted probability of exposure) to match users instead of actual covariates, and was first proposed in [35] and many follow-up works, nicely surveyed in [8], have proposed different refinements under the framework of the PSM. Besides PSM, other alternative ways to do such matching such as inverse propensity weighting [19, 20] and doubly robust estimation [18, 26] are also popular. As we mentioned earlier, all these works usually require that treatment and untreated/unexposed (control) groups be clearly identified. Thus, it is not directly applicable in our spam study as discussed earlier in Section 2.

Causal effects have been studied in many application scenarios, especially on the Web [9, 38]. For example, [9] applied several PSM to study the effect of online ads. To the best of our knowledge, there is no previous study on the causality effect of email spam on user engagement.

Impact of Email Spam: As discussed before, email spam has become a critical problem, being also related to

various online nefarious activities [28] such as phishing, scamming and spreading malware. Our paper is related to recent works that try to quantify the impact of spam from the economic side. For example, [22] conducted a study to quantify the conversion rate of the spam in order to understand how much spammers earned off bulk email distribution. The focus was thus the economics of the spam campaigns, rather than the user level metrics. [42] studied how much inconvenience of users is caused by the spam mails, by measuring the user’s “willingness to pay” to remain unaffected by spam. [7] studied the cost of spam and the cost saved by spam filtering. The goal of all these papers is to quantify the cost from an organization’s point of view, and their main metric is amount of working time spent in dealing with spam. Our aim was instead to measure the effect on the user engagement metrics from the economic perspective of the email service provider. Since the email service provider is the key entity that invests in anti-spam technology, we feel this is a useful perspective to adopt.

Studying the impact of spam on users is part of a broader trend trying to characterize the economic incentives each of the stakeholders has in combating spam. Understanding the underground economy is the counterpart of what we are doing here. As mentioned before, researchers have concentrated on individual parts of this economy—the supply chain [22, 23], the labor market [31, 30] and malware distribution [6]. We consider our work as complementary to this thread, shedding light onto the ESP-centric part of the economic cycle.

6 Discussion and Summary

In this paper we described a large scale matching method, along with the corresponding regression method, in order to infer causal effects from observational data, specifically applicable in the case when the exposure variable is continuous. In situations where exposure is not a decision of the user but is correlated with engagement metrics, observational methods run into the correlation without causation problem. The gold standard to measure causality of course is a randomized experiment, but they are often too risky from a revenue or brand management perspective (the negative impact might outweigh the knowledge gains), unethical (involve exposing users to bad outcomes) or not ideal because the underlying behavior requires large changes in the independent variable of interest to measure a behavioral response. Mail spam runs afoul of all these requirements of A/B testing and is inherently interesting to study, given how pervasive it is in email-based communication.

We provide quantitative estimates that show that the impact of spam in the inbox can have serious revenue implications and can contribute to a large percentage drop

in user engagement. The effect is largest for more voluntary user activities such as sending and especially composing emails. The function mapping spam changes to engagement appears to be convex, with the marginal impact increasing with the size of the exposure change. We carefully looked for contagion effects and found that while there are meaningful spillovers (reduced engagement across the Yahoo! site) the spillovers can be mechanically linked to decreased webmail activity so are thus not pure “brand-loss” effects, even though they are still relevant in evaluating the revenue impact. User characteristics are not particularly informative in predicting the response to spam; the most notable result is that light users are equally affected in absolute terms by a piece of spam in the inbox, meaning that percentage-wise the impact is far greater for these users.

Our result shows why it is important to quantitatively estimate a behavior even when the sign of the impact is “obvious.” Merely documenting that mail spam has negative impact on engagement would not be particularly informative, but pinning the magnitude of the impact and the channels through which it operates can help the firm make investment decisions in filtration technology and optimize the user-interface to mitigate the effects. We believe the method can be fruitfully applied to other forms of abuse, such as abusive user-generated content, and other online experiences, such as pop-up ads.

References

- [1] E. Allman. The economics of spam. *Queue*, 1(9):80, 2003.
- [2] A. Andoni. Nearest neighbor search: the old, the new, and the impossible, 2009.
- [3] P. Barford and V. Yegneswaran. An inside look at botnets. *Malware Detection*, pages 171–191, 2007.
- [4] A. Bergholz, J. De Beer, S. Glahn, M.-F. Moens, G. Paaß, and S. Strobel. New filtering approaches for phishing email. *J. Comput. Secur.*, 18(1):7–35, 2010.
- [5] A. Bratko, B. Filipič, G. V. Cormack, T. R. Lynam, and B. Zupan. Spam filtering using statistical data compression models. *Journal of Machine Learning Research*, 7:2673–2698, 2006.
- [6] J. Caballero, C. Grier, C. Kreibich, and V. Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *Proceedings of the 20th USENIX Security Symposium*, 2011.
- [7] M. Caliendo, M. Clement, D. Papiés, and S. Scheel-Kopeinig. The cost impact of spam filters: Measuring the effect of information system technologies in organizations. *IZA Discussion Paper No. 3755*, 2008.
- [8] M. Caliendo and S. Kopeinig. Some practical guidance for the implementation of propensity score matching. *Journal of Economic Surveys*, 22(1):31–72, 2008.

- [9] D. Chan, R. Ge, O. Gershony, T. Hesterberg, and D. Lambert. Evaluating online ad campaigns in a pipeline: causal models at scale. In *16th ACM SIGKDD*, pages 7–16. ACM, 2010.
- [10] T. Claburn. Spam made up 94% of all e-mail in december. Technical report, Information Week, <http://www.informationweek.com/news/internet/showArticle.jhtml?articleID=197001430>, 2007.
- [11] W. S. Cleveland. Robust locally weighted regression and smoothing scatterplots. *Journal of the American Statistical Association*, 74(368):829–836, 1979.
- [12] G. V. Cormack. Email spam filtering: A systematic review. In *Foundations and Trends in Information Retrieval*, 2008.
- [13] A. Dasgupta, M. Gurevich, and K. Punera. Enhanced email spam filtering through combining similarity graphs. In *WSDM*, pages 785–794, 2011.
- [14] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *CHI*, pages 581–590, New York, NY, USA, 2006. ACM.
- [15] D. Fiebig and R. Bartels. The frisch-waugh theorem and generalized least squares. *Econometric Reviews*, 15(4):431–443, 1996.
- [16] J. Goodman, G. V. Cormack, and D. Heckerman. Spam and the ongoing battle for the inbox. *Commun. ACM*, 50(2):24–33, 2007.
- [17] C. Herley and D. Florêncio. A profitless endeavor: phishing as tragedy of the commons. In *Proceedings of the 2008 workshop on New security paradigms*, NSPW '08, pages 59–70, New York, NY, USA, 2008. ACM.
- [18] K. Hirano, G. W. Imbens, and G. Ridder. Efficient estimation of average treatment effects using the estimated propensity score. *Econometrica*, 71(4):1161–1189, 07 2003.
- [19] D. G. Horvitz and D. J. Thompson. A Generalization of Sampling Without Replacement From a Finite Universe. *Journal of the American Statistical Association*, 47(260):663–685, 1952.
- [20] J. Huang, A. J. Smola, A. Gretton, K. M. Borgwardt, and B. Schölkopf. Correcting Sample Selection Bias by Unlabeled Data. In *Advances in Neural Information Processing Systems 19*, pages 601–608, 2007.
- [21] C. Ivey. <http://www.shoestringmillionaire.com/the-asymmetrical-economy-of-spam/>, 2011.
- [22] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. Spamalytics: an empirical analysis of spam marketing conversion. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 3–14, 2008.
- [23] C. Kanich, N. Weaver, D. McCoy, T. Halvorson, C. Kreibich, K. Levchenko, V. Paxson, G. Voelker, and S. Savage. Show me the money: characterizing spam-advertised revenue. In *Proceedings of the 20th USENIX Security Symposium*, pages 8–12, 2011.
- [24] R. Kohavi, R. Longbotham, D. Sommerfield, and R. M. Henne. Controlled experiments on the web: survey and practical guide. *KDD*, 18(1):140–181, 2009.
- [25] R. Lewis, J. Rao, and D. Reiley. Here, there, and everywhere: correlated online behaviors can lead to overestimates of the effects of advertising. In *WWW 2011*, pages 157–166. ACM, 2011.
- [26] J. K. Lunceford and M. Davidian. Stratification and weighting via the propensity score in estimation of causal treatment effects: a comparative study. *Statistics in Medicine*, 23(19):2937–2960, 2004.
- [27] S. Malinin. Spammers earn millions and cause damages of billions. <http://english.pravda.ru/russia/economics/15-09-2005/8908-spam-0/>, 2005.
- [28] T. Moore, R. Clayton, and R. Anderson. The economics of online crime. *Journal of Economic Perspectives*, 23(3):3–20, 2009.
- [29] S. Morgan and C. Winship. *Counterfactuals and Causal Inference: Methods and Principles for Social Research*. Analytical Methods for Social Research. Cambridge University Press, 2007.
- [30] M. Motoyama, K. Levchenko, C. Kanich, D. McCoy, G. M. Voelker, and S. Savage. Re: Captchas – understanding captcha-solving from an economic context. In *Proceedings of the USENIX Security Symposium*, August 2010.
- [31] M. Motoyama, D. McCoy, K. Levchenko, S. Savage, and G. Voelker. Dirty jobs: The role of freelance labor in web service abuse. In *Proceedings of the 20th USENIX Security Symposium*, 2011.
- [32] Y. Namestnikov. The economics of botnets. *Analysis on Viruslist.com, Kapersky Lab*, 2009.
- [33] E. Park. Update on global spam volume. <http://www.symantec.com/connect/blogs/update-global-spam-volume>.
- [34] J. M. Rao and D. H. Reiley. The economics of spam. *Journal of Economic Perspectives*, Forthcoming Summer, 2012.
- [35] P. Rosenbaum and D. Rubin. The central role of the propensity score in observational studies for causal effects. *Biometrika*, 70(1):41, 1983.
- [36] D. B. Rubin. Estimating causal effects of treatments in randomized and nonrandomized studies. *Journal of Educational Psychology*, 66(5):688–701, 1974.
- [37] D. B. Rubin. Assignment to treatment group on the basis of a covariate. *Journal Of Educational Statistics*, 2(1):1–26, 1977.
- [38] D. B. Rubin and R. P. Waterman. Estimating the causal effects of marketing interventions using propensity score methodology. *Statistical Science*, 21(2):206–222, 2006.
- [39] D. Sculley and G. M. Wachman. Relaxed online svms for spam filtering. In *SIGIR*, pages 415–422, New York, NY, USA, 2007. ACM.

- [40] A. K. Seewald. An evaluation of naive bayes variants in content-based learning for spam filtering. *Intelligent Data Analysis*, 11(5):497–524, 2007.
- [41] B. Stone-Gross, T. Holz, G. Stringhini, and G. Vigna. The underground economy of spam: a botmaster’s perspective of coordinating large-scale spam campaigns. In *Proceedings of the 4th USENIX conference on Large-scale exploits and emergent threats*, LEET’11, pages 4–4, Berkeley, CA, USA, 2011. USENIX Association.
- [42] S.-H. Yoo, C.-O. Shin, and S.-J. Kwak. Inconvenience cost of spam mail: a contingent valuation study. *Applied Economics Letters*, 13(14):933–936, November 2006.

7 Appendix: Comparison to Propensity Score Matching via Simulations

There has also been much research into developing techniques, e.g., covariate matching, bias reduction, propensity score matching (PSM) [35, 20, 9], etc, which have shown promising results in removing this bias in observational studies. In this section, we outline the basic framework of propensity score matching and then discuss why the basic framework is unsuitable for us. We then compare our proposed method, nearest neighbor matching, with two variants of propensity score matching model based on a simulation data set with ground truth. Although our use of nearest neighbor matching method was prompted by concerns e.g. continuous exposure variable that make the naive PSM inapplicable, nevertheless we want to test whether there exist variants of PSM that are more adapted for our purposes. In order to do such a test, we needed to simulate the actual ground truth measure so that we can compare the effects unearthed by each method to the ground truth. In what follows, we first give an outline of PSM and then describe a variant we develop, stratified-PSM, that we compare with the nearest neighbor matching technique that we use. We then describe how we created the simulation dataset and compared the different algorithms.

7.1 Propensity Score Matching

In this section, we first briefly explain the PSM method of estimating effects before describing the modifications. In the classical PSM model, we have clearly defined treated and untreated (unexposed) groups—denote them by U_1 and U_0 respectively. The goal is to study the effect or outcome y on the treated users. For each user u , we use $y_u(s = 1)$ or $y_u(s = 0)$ to represent the effect on user u depending on whether the user is treated or remains untreated. Thus, we are interested in measuring the effect of treatment as $\Delta y = E[y_u(s = 1) - y_u(s = 0)|u \in U_1]$. However, a single user u can either be in the treated or the untreated group, but not both. A naive estimator of

the above effect would thus be $\Delta y = E[y_u(s = 1)|u \in U_1] - E[y_u(s = 0)|u \in U_0]$ —this faces the problem of selection bias, since the populations in U_1 and U_0 are different, and have different properties which can be correlated with outcome y . The basic idea in PSM to overcome this bias is to select one or more users in the control group for each treated user, based on some pre-exposure features \mathbf{x}_u . Under the condition of unconfoundedness,

$$Pr(y_u(s = 0)|\mathbf{x}_u, u \in U_0) = Pr(y_u(s = 0)|\mathbf{x}_u, u \in U_1),$$

we have the following estimator

$$\Delta y = E[y_u(s = 1)|u \in U_1] - E_{\mathbf{z} \in U_1}[y_u(s = 0)|u \in U_0, \mathbf{x}_u = \mathbf{z}],$$

where $\mathbf{z} \in U_1$ means \mathbf{z} is a feature vector of a treated user. To avoid matching on the whole feature vector \mathbf{x}_u , we can match on the one-dimensional propensity score $p(\mathbf{x}_u)$ which is the probability that a user with vector \mathbf{x}_u belongs to the treatment group. Then we have

$$\Delta y = E[y_u(s = 1)|u \in U_1] - E_{v \in p(U_1)}[y_u(s = 0)|u \in U_0, p(\mathbf{x}_u) = v],$$

where $v \in p(U_1)$ means that v is a propensity score of a treated user.

7.2 Unsuitability of PSM

As described above, the main aim in PSM is to try to learn a consistent estimator of $p(\mathbf{x})$, the probability the user has been exposed to a certain amount of spam, based on the all the feature we have constructed. In our case, we proceed differently due to a couple of reasons as pointed out – the basic underpinning of propensity score matching methods is being able to model the probability that a particular user falls into the treatment group. If the exposure variable is continuous, this assumption, and hence the modeling falls apart. We instead have to have a variant where we would have to create separate models for each value of the exposure. Secondly, the primary reason for propensity score matching is because matching users becomes difficult if the activity vector is high dimensional and the number of users is small – this is not the case for us: we have tens of features and we have over a million users; and we are able to find close matches. Lastly, being able to create a model that is a consistent estimator of $p(\mathbf{x})$ is very important, else we could be subject to un-intended biases that arise from this modeling.

In the presence of these issues, the commonly used ways of applying propensity score matching (PSM) does not apply to us. In the next subsection we describe a variant of PSM, where we stratify the dataset into multiple exposure levels and solve a PSM for each level.

PAIR	PSM1	PSM2	PSM2-W
1.579	3.376	4.578	5.878

Table 4: The L1 difference from the ground-truth. The smaller the value the better.

7.3 Variants of PSM for continuous exposure

In our problem, we care about the effect on engagement difference Δy if the spam fraction increases by Δs . To adapt PSM in our setting, we start out by first grouping users by discretizing their spam fraction values. Given a set of user U and their spam fraction range $[a, b]$, we have the following two ways of grouping users:

- **Equal-depth grouping.** In this method, we order all the users based on their spam fraction values increasingly. We then split the order list equally into m segments. In this method, each group has the same number of users.
- **Equal-width grouping.** In this method, we cut the spam fraction $[a, b]$ equally into m segments, each with a width of $(b - a)/m$. Users are grouped accordingly. In this method, each group can have different number of users.

Given a grouping method, for each pair of user segments, we use the segment with the lower spam fraction as the treated group and the one with the higher spam fraction as the control group – we compute Δs , the difference of the spam fraction between these two groups, as the difference of the average over the users in the two groups. We can then use a PSM model to compute the effect Δy . At the end, we will have a set of $(\Delta s, \Delta y)$ pairs.

To get the estimation function between the effect difference and spam fraction difference, we use the local regression method [11] to fit a curve on the set of $(\Delta s, \Delta y)$ pairs. We use PSM1 to denote Equal-depth grouping and PSM2 to denote Equal-width grouping. Please note that we have the same number of users for each $(\Delta s, \Delta y)$ in PSM1 but we have different numbers of users for PSM2. Thus for PSM2, we have a weighed version PSM2-W by weighing each point proportional the number of users in the treatment group before fitting the curve.

7.4 Simulation Results

To test the validity of our method by comparing it against ground truth, as well as to compare different variants of PSM with our method, we generate a simulation data with ground truth by the following procedure: we sub-sample 50K users from the mail-spam data that described in Section 4. For each user, we only kept 8 matching features – the mail pageviews, the incoming mail, incoming

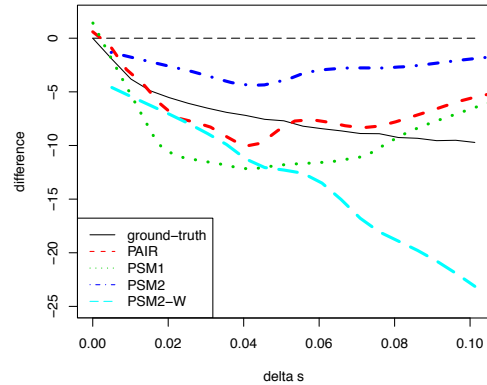


Figure 4: Comparison of our method PAIR and the variants of PSM methods.

spam, the outgoing mails for two months. The spam-fraction in exposure month is the exposure variable, and the mail page-views in the post-exposure month is the effect variable. Because we want to generate the ground truth effect as close to the real effect as possible, we then learnt a gradient boosted decision tree model that tries to fit the effect variable in terms of the matching features and the exposure variable. This model that we learnt of user behavior was then used to create the new values of the effect variable for each user – as the user-set was sub-sampled, we strengthened the impact of exposure on the mail-pageviews by adding in another component to the model – this was a log-normally distributed random variable whose expectation depends on the logarithm of the difference of the spam exposure of this user from the mean spam exposure of all users: this changed each predicted effect value by around 10%. This aggregated model was then used to generate the new data, and also to create the ground truth curve for each value of Δs by predicting the new effect and then averaging over all user with the same matching features.

We show the comparison results in Figure 4. For PSM methods, we set the number of user groups $m = 20$. (We tried different values for m and found the results are not very sensitive.) For our method, we obtain 1.17M pairs after our nearest neighbor matching and filtering steps. Each pair gives us a $(\Delta y, \Delta s)$ point and we use the same local regression method [11] to get a fitted curve. In Figure 4, we show the ground truth curve for $\Delta y(\Delta s)$, as well as the estimated curves for every method. Each of the estimates does capture the negative correlation between Δs and Δy . But, the estimates produced by the PSM methods are certainly worse than the one created by the nearest neighbor matching method. This is measured quantitatively by the L1 difference between the each estimated curve with the ground truth one – which we compute using 20 sampled points of

$\Delta s = \{0.005, 0.01, \dots, 0.095, 0.1\}$. The L1 differences are shown in Table 4. One of the reasons of PSM performing worse is that when Δs becomes large, the resulting buckets have small number of users, and hence the variance is high. This simulation provides evidence that the matching method provides a reasonable set of estimates to ground truth, and that it performs better than some obvious variants of PSM, when dealing with continuous treatment values.

Security and Usability Challenges of Moving-Object CAPTCHAs: Decoding Codewords in Motion

Y. Xu[†], G. Reynaga[‡], S. Chiasson[‡], J-M. Frahm[†], F. Monrose[†] and P. van Oorschot[‡]

[†]Department of Computer Science, University of North Carolina at Chapel Hill, USA

[‡]School of Computer Science, Carleton University, Canada

email: {yix,jmf,fabian}@cs.unc.edu, {gerardor,chiasson,paulv}@scs.carleton.ca

Abstract

We explore the robustness and usability of moving-image object recognition (video) captchas, designing and implementing automated attacks based on computer vision techniques. Our approach is suitable for broad classes of moving-image captchas involving rigid objects. We first present an attack that defeats instances of such a captcha (NuCaptcha) representing the state-of-the-art, involving dynamic text strings called codewords. We then consider design modifications to mitigate the attacks (e.g., overlapping characters more closely). We implement the modified captchas and test if designs modified for greater robustness maintain usability. Our lab-based studies show that the modified captchas fail to offer viable usability, even when the captcha strength is reduced below acceptable targets—signaling that the modified designs are not viable. We also implement and test another variant of moving text strings using the known *emerging images* idea. This variant is resilient to our attacks and also offers similar usability to commercially available approaches. We explain why fundamental elements of the emerging images concept resist our current attack where others fails.

1 Introduction

Humans can recognize a wide variety of objects at a glance, with no apparent effort, despite tremendous variations in the appearance of visual objects; and we can answer a variety of questions regarding shape properties and spatial relationships of what we see. The apparent ease with which we recognize objects belies the magnitude of this feat. We can also do so with astonishing speed (e.g., in a fraction of a second) [41]. Indeed, the Cognitive Science literature abounds with studies on visual perception showing that, for the most part, people do not require noticeably more processing time for object categorization (e.g., deciding whether the object is

a bird, a flower, a car) than for more fine grained object classification (e.g., an eagle, a rose) [13]. Grill et al. [20] showed that by the time subjects knew that a picture contained an object at all, they already knew its class. If such easy-for-human tasks are, in contrast, difficult for computers, then they are strong candidates for distinguishing humans from machines.

Since understanding what we see requires *cognitive* ability, it is unsurprising that the decoding of motion-based challenges has been adopted as a security mechanism: various forms of motion-based object recognition tasks have been suggested as reverse Turing tests, or what are called Completely Automated Public Turing tests to tell Computers and Humans Apart (captchas). Among the key properties of captchas are: they must be easily solved by humans; they should be usable; correct solutions should only be attainable by solving the underlying AI problem they are based on; they should be robust (i.e., resist automated attacks); and the cost of answering challenges with automated programs should exceed that of soliciting humans to do the same task [1, 46]. To date, a myriad of text, audio, and video-based captchas have been suggested [22], many of which have succumbed to different attacks [6, 7, 19, 32, 47, 48, 53].

While text-based captchas that prompt users to recognize distorted characters have been the most popular form to date, motion-based or video captchas that provide some form of moving challenge have recently been proposed as the successor to static captchas. One prominent and contemporary example of this new breed of captchas is NuCaptcha [35], which asserts to be “*the most secure and usable captcha*,” and serves millions of video captchas per day. The general idea embodied in these approaches is to exploit the remarkable perceptual abilities of humans to unravel structure-from-motion [30]. For example, users are shown a video with a series of characters (so-called random *codewords*) moving across a dynamic scene, and solve the captcha by entering the correct codeword. For enhanced security, the

codewords are presented among adversarial clutter [32] (e.g., moving backgrounds and other objects with different trajectories), and consecutive characters may even overlap significantly. The underlying assumption is that attacks based on state-of-the-art computer vision techniques are likely to fail at uncovering these challenges within video sequences, whereas real users will be able to solve the challenges with little effort.

However, unlike in humans, it turns out that object *classification*, not recognition of known objects, is the more challenging problem in Computer Vision [43]. That is, it is considerably more difficult to capture in a computer recognition system the essence of a dog, a horse, or a tree—i.e., the kind of classification that is natural and immediate for the human visual system [29]. To this day, classification of objects in real-world scenes remains an open and difficult problem. Recognizing known objects, on the other hand, is more tractable, especially where it involves specific shapes undergoing transformations that are easy to compensate for. As we show later, many of these well-defined transformations hold in current motion-based captcha designs, due in part to design choices that increase usability.

In what follows, we present an automated attack to defeat the current state-of-the-art in moving-image object recognition captchas. Through extensive evaluation of several thousand real-world captchas, our attack can completely undermine the security of the most prominent examples of these, namely those currently generated by NuCaptcha. After examining properties that enable our attack, we explore a series of security countermeasures designed to reduce the success of our attacks, including natural extensions to the scheme under examination, as well as an implementation of a recently proposed idea (called Emerging Images [31]) for which attacks do not appear as readily available. Rather than idle conjecture about the efficacy of countermeasures, we implement captchas embedding them and evaluate these strengthened variations of moving-image captchas by carrying out and reporting on a usability study with subjects asked to solve such captchas.

Our findings highlight the well-known tension between security and usability, which often have subtle influences on each other. In particular, we show that the design of robust and usable moving-image captchas is much harder than it looks. For example, while such captchas may be more usable than their still-based counterparts, they provide an attacker with a significant number of views of the target, each providing opportunities to increase the confidence of guesses. Thus the challenge is limiting the volume of visual cues available to automated attacks, without adversely impacting usability.

2 Captcha Taxonomy and Related Work

Most captchas in commercial use today are character-recognition (CR) captchas involving still images of distorted characters; attacks essentially involve building on optical character recognition advances. Audio captchas (AUD) are a distinct second category, though unrelated to our present work. A third major category, image-recognition (IR) captchas, involves classification or recognition, of images or objects other than characters. A well-known example, proposed and then broken, is the Asirra captcha [16, 19] which involves object classification (e.g., distinguishing cats from other animals such as dogs). CR and IR schemes may involve still images (CR-still, IR-still), or various types of dynamic images (CR-dynamic, IR-dynamic). Dynamic text and objects are of main interest in the present paper, and contribute to a cross-class category: moving-image object recognition (MIOR) captchas, involving objects in motion through animations, emergent-image schemes, and video [10–12, 26, 31, 35, 38]. A fourth category, cognitive-based captchas (COG), include puzzles, questions, and other challenges related to the semantics of images or language constructs. We include here content-based video-labeling of YouTube videos [24].

The most comprehensive surveys of captchas to date are those by Hidalgo and Maranon [22] and Basso and Bergadano [2]. We also recommend other comprehensive summaries: for defeating classes of AUD captchas, Soupionis [40] and Bursztein et al. [4, 6]; for defeating CR captchas, Yan et al. [47, 50] and Bursztein [7]; for a systematic treatment of IR captchas and attacks, Zhu et al. [53], as well as for robustness guidelines.

Usability has also been a central focus, for example, including a large user study of CR and AUD captchas involving Amazon Mechanical Turk users [5], a user study of video-tagging [24], usability guidelines and frameworks related to CR captchas [49]. Chellapilla et al. [8, 9] also address robustness. Hidalgo et al. [22] and Bursztein et al. [7] also review evaluation guidelines including usability. Lastly, research on underground markets for solving captchas [33], and malware-based captcha farms [15], raise interesting questions about the long-term viability of captchas.

Lastly, concurrent to our own work, Bursztein [3] presents an approach to break the video captchas used by NuCaptcha. The technique exploits the video by treating it as a series of independent frames, and then applies a frame-based background removal process [7] to discard the video background. Next, frame characteristics (e.g., spatial salient feature density and text aspect ratio of the overlapping letters) are used to detect the codeword, after which a clustering technique is used to help segment the characters of the codeword. As a final step,

traditional CR-still based attacks are used to recognize the characters in each of the segmented frames. The approach taken by Bursztein is closely related to our baseline method (§4.1) as it only uses single frame segmentation and recognition. In contrast, our subsequent techniques inherently use temporal information contained in the video to identify the codeword, to improve the segmentation, and to enhance the recognition step during the codeword recovery process.

3 Background

In the human brain, it is generally assumed that an image is represented by the activity of “units” tuned to local features (e.g., small line and edge fragments). It is also widely believed that objects appearing in a consistent or familiar background are detected more accurately, and processed more quickly, than objects appearing in an inconsistent scene [36]. In either case, we must somehow separate as much as possible of the image once we see it. This feat is believed to be done via a *segmentation* process that attempts to find the different objects in the image that “go together” [43].

As with other aspects of our visual system, segmentation involves different processes using a multitude of sources of information (e.g., texture and color), which makes it difficult to establish which spatial properties and relations are important for different visual tasks. While there is evidence that human vision contains processes that perform grouping and segmentation prior to, and independent of, subsequent recognition processes, the exact processes involved are still being debated [36].

Given the complexity of the visual system, it is not surprising that this feat remains unmatched by computer vision algorithms. One of the many reasons why this task remains elusive is that perception of seemingly simple spatial relations often requires complex computations that are difficult to unravel. This is due, in part, to the fact that object classification (that is, the ability to accurately discriminate each object of an object class from all other possible objects in the scene) is computationally difficult because even a single individual object can already produce an infinite set of different images (on the retina) due to variations in position, scale, pose, illumination, etc. Discriminating objects of a certain class is further complicated by the often very large inner class variability, which significantly changes the appearance beyond the factors encountered for a single object. Hence, vision operates in a high-dimensional space, making it difficult to build useful forms of visual representation.

In computer vision, the somewhat simpler process of recognizing known objects is simulated by first analyzing an image locally to produce an edge map composed of a large collection of local edge elements, from which

we proceed to identify larger structures. In this paper, we are primarily interested in techniques for object segmentation and tracking. In its simplest form, *object tracking* can be defined as the problem of estimating the trajectory of an object in the image plane as it moves around a scene. Tracking makes use of temporal information computed from a sequence of frames. This task can be difficult for computer vision algorithms because of issues related to noise in the image, complex object motion, the nonrigid nature of objects, etc. However, the tracking problem can be simplified if one can assume that object motion is smooth, the motion is of constant velocity, knowledge of the number and the size of the objects, or even appearance and shape information. In NuCaptcha, for example, many of these simplifications hold and so several features (e.g., edges, optical flow) can be used to help track objects. The correspondence search from one frame to the next is performed by using tracking.

In video, this correspondence can be achieved by building a representation of the scene (called the *background model*) and then finding deviations from the model for each incoming frame. Intuitively, any significant change in the image region from the background model signifies a moving object. The pixels constituting the regions undergoing change are marked for further processing, and a connected component algorithm is applied to obtain connected regions. This process is typically referred to as *background subtraction*. At this point, all that is needed is a way to partition the image into perceptually similar regions, and then infer what each of those regions represent. In §4, we discuss the approach we take for tackling the problems of background subtraction, object tracking, segmentation, and classification of the extracted regions.

4 Our Automated Approach

The aforementioned processes of segmentation, object tracking, and region identification are possible in today’s MIOR captchas because of several design decisions that promote rapid visual identification [14]. NuCaptcha, for instance, presents a streaming video containing moving text against a dynamic background. The videos have four noticeable characteristics, namely: (1) the letters are presented as rigid objects in order to improve a user’s ability to recognize the characters; (2) the background video and the foreground character color are nearly constant in color and always maintain a high contrast—we posit that this is done to ease cognitive burden on users; (3) the random “codewords” each have independent (but overlapping trajectories) which better enable users to distinguish adjacent characters; (4) lastly, the codewords are chosen from a reduced alphabet where easily confused characters are omitted. Some examples of a state-of-the-



Figure 1: Example moving-image object recognition (MIOR) captchas from NuCaptcha (see <http://nucaptcha.com/demo>).

art MIOR captcha are given in Figure 1.

Before delving into the specifics of our most successful attack, we first present a naïve approach for automatically decoding the challenges shown in MIOR captchas. To see how this attack would work, we remind the reader that a video can be seen as a stream of single pictures that simply provides multiple views of a temporally evolving scene. It is well known that human observers perceive a naturally moving scene at a level of about thirty frames per second, and for this reason, video captchas tend to use a comparable frame rate to provide a natural video experience that is not too jerky. Similarly, the challenge shown in the captcha is rendered in multiple frames to allow users to perceive and decode the codewords in an effortless manner. In the NuCaptcha scheme, for example, a single frame may contain the full codeword.

4.1 A Naïve Attack

Given this observation, one way to attack such schemes is to simply apply traditional OCR-based techniques that

work well at defeating CR-still captchas (e.g., [32, 47]). More specifically, choose k frames at random, and identify the foreground pixels of the codeword by comparing their color with a given reference color; notice the attacker would likely know this value since the users are asked to, for example, “type the RED moving characters”. Next, the length of the codeword can be inferred by finding the leftmost and rightmost pixels on the foreground. This in essence defines a line spanning over the foreground pixels (see Figure 2). The positions of the characters along the line can be determined by dividing the line into n equidistant segments, where n denotes the desired number of characters in the codeword. For each of the segments, compute the center of gravity of the foreground pixels in the vertical area of the image belonging to the segment. Lastly, select an image patch (of the expected size of the characters) around the centers of gravity of the segments, and feed each patch to a classifier. In our work, we use a neural network approach [39] because it is known to perform well at this object identification task. The neural network is trained in a manner similar to what we discuss in §4.3.

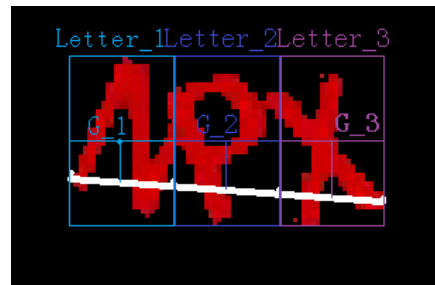


Figure 2: Naïve attack: Based on the foreground pixels, we find the longest horizontal distance (white line) and the mean value of vertical area (the respective bounding boxes above).

The above process yields a guess for each of the characters of the codeword in the chosen frames of the video. Let i denote the number of possible answers for each character. By transforming the score from the neural network into the probability p_{ijk} where the j -th character of the codeword corresponds to the i -th character in the k -th frame, we calculate the probability P_{ij} for each character $j = 1, \dots, n$ of the codeword over all k frames as $P_{ij} = \frac{1}{s_p} \sum_k p_{ijk}$ with $s_p = \sum_{i,j,k} p_{ijk}$. The choice that has the highest probability is selected as the corresponding character. With $k = 10$, this naïve attack resulted in a success rate of approximately 36% accuracy in correctly deducing all three characters in the codewords of 4000 captchas from NuCaptcha. While this relatively simple attack already raises doubts about the robustness of this new MIOR captcha, we now present a significantly improved attack that makes fewer assumptions about pixel invariants [50] in the videos.

4.2 Exploiting Temporal Information

A clear limitation of the naïve attack is the fact that it is not easily generalizable and it is not robust to slight changes in the videos. In what follows, we make no assumption about a priori knowledge of the color of the codewords, nor do we assume that the centers of gravity for each patch are equidistant. To do so, we apply a robust segmentation method that utilizes temporal information to improve our ability to recognize the characters in the video.

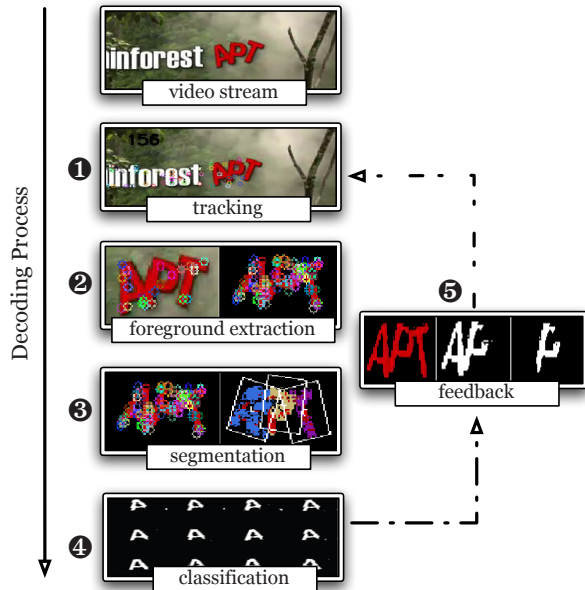


Figure 3: High-level overview of our attack. (This, and other figures, are best viewed in color.)

A basic overview of our attack is shown in Figure 3. Given a MIOR captcha we extract the motion contained in the video using the concept of salient features. Salient features are characteristic areas of an image that can be reliably detected in several frames. To infer the motion of the salient feature points, we apply object tracking techniques (stage ①). With a set of salient features at hand, we then use these features to estimate the color statistics of the background. Specifically, we use a Gaussian mixture model [18], which represents the color statistics of the background through a limited set of Gaussian distributions. We use the color model of the background to measure, for all pixels in each frame, their likelihood of belonging to the background. Pixels with low likelihoods are then extracted as foreground pixels (stage ②). The trajectories of the foreground pixels are then refined using information inferred about the color of these pixels, and a foreground color model is built. Next, to account for the fact that all characters of the codewords move independently, we segment the foreground into n

segments as in the naïve attack (stage ③). We select each image patch containing a candidate character and evaluate the patch using a neural network based classifier [39] (stage ④). The classifier outputs a likelihood score that the patch contains a character. As a final enhancement, we incorporate a feedback mechanism in which we use high confidence inferences to improve low confidence detections of other patches. The net effect is that we reduce the distractions caused by mutually overlapping characters. Once all segments have been classified, we output our guess for all characters of the codeword. We now discuss the stages of our approach in more detail.



Figure 4: The circles depict salient features. These salient features are usually corners of an object or texture areas.

Detecting Salient Features and Their Motion (Stage ①)

A well-known class of salient features in the computer vision community is gray value corners in images. In this paper, we use the Harris corner detector [21] for computing salient features, which uses the image gradient to identify points in the image with two orthogonal gradients of significant magnitude. An example of the detected corners is shown in Figure 4.

After identifying salient features in one frame of the video we now need to identify their respective position in the subsequent frames of the video. In general, there are two choices for identifying the corresponding salient features in the subsequent frames of the video. The first choice is to independently detect salient features in all frames and then compare them by using their image neighborhoods (patches) to identify correlating patches through an image based correlation (commonly called matching). The second class of methods leverages the small motion occurring in between two frames for an iterative search (commonly called tracking).

We opt for a tracking method given that tracking results for video are superior in accuracy and precision to matching results. Specifically, we deploy the well known KLT-tracking method [28], which is based on the assumption that the image of a scene object has a constant appearance in the different frames capturing the object (brightness constancy). The MIOR captchas by NuCaptcha use constant colors on the characters of the codewords. This implies that the NuCaptcha frames are

well suited for our method. Note that no assumption about the specific color is made; only constant appearance of each of the salient features is assumed. We return to this assumption later in Section 5.2.

Motion Trajectory Clustering (Stage ②)

In a typical video, the detected salient features will be spread throughout the image. In the case of NuCaptcha, the detected features are either on the background, the plain (i.e., non-codeword) characters or the codeword characters. We are foremost interested in obtaining the information of the codeword characters. To identify the codeword characters we use their distinctive motion patterns as their motion is the most irregular motion in the video captcha. In the case of NuCaptcha, we take advantage of the fact that the motion trajectories of the background are significantly less stable (i.e., across consecutive frames) than the trajectories of the features on the characters. Hence we can identify background features by finding motion trajectories covering only a fraction of the sequence; specifically we assume presence for less than $l = 20$ frames. In our analysis, we observed little sensitivity with respect to l .

Additionally, given that all characters (plain and codeword) move along a common trajectory, we can further identify this common component by linearly fitting a trajectory to their path. Note that the centers of the rotating codeword characters still move along this trajectory. Accordingly, we use the distinctive rotation of the codeword characters to identify any of their associated patterns by simply searching for the trajectories with the largest deviation from the more common motion trajectory. This identifies the pixels belonging to the codeword characters as well as the plain characters. Additionally, the features on the identified codeword characters allow us to obtain the specific color of the codeword characters without knowing the color a priori (see Figure 5).

Knowing the position of the codeword characters allows us to learn a foreground color model. We use a Gaussian mixture model for the foreground learning, which in our case has a single moment corresponding to the foreground color.¹ Additionally, given the above identified salient features on the background, we also learn a Gaussian mixture for the background, thereby further separating the characters from the background.

At this point, we have isolated the trajectories of codeword characters, and separated the codewords from the background (see Figure 6). However, to decide which salient features on the codeword characters belong together, we required additional trajectories. To acquire these, we simply relax the constraint on the sharpness of corners we care about (i.e., we lower the threshold for the Harris corner detection algorithm) and rerun the



Figure 5: (Top): Initial optical flow. (Middle): salient points with short trajectories in background are discarded. (Lower): Trajectories on non-codeword characters are also discarded.

KLT-tracking on the new salient features. This yields significantly more trajectories for use by our segmentation algorithm. Notice how dense the salient features are in Figure 7. Note also that since the foreground extraction step provides patches that are not related to the background, we can automatically generate training samples for our classifier, irrespective of the various backgrounds the characters are contained in.



Figure 6: Example foreground extraction.



Figure 7: re-running tracking with a lower threshold on corner quality: Left: before modification. Right: after modification.

Segmentation (Stage ③)

To segment the derived trajectories into groups, we use k -means clustering [23]. We chose this approach over other considerations (e.g., mean-shift [37] based clustering, or

RANSAC [17] based clustering [51]) because of its simplicity, coupled with the fact that we can take advantage of our knowledge of the desired number of characters (i.e., k), and use that to help guide the clustering procedure. We cannot, however, apply the standard k -means approach directly since it relies on Euclidean distances, where each sample is a point. In our case, we need to take the relationship between frames of the video sequence into consideration, and so we must instead use each trajectory as an observation. That is, we cluster the different trajectories. However, this results in a non-Euclidean space because different trajectories have different beginning and ending frames. To address this problem, we utilize the rigidity assumption [42] and define a distance metric for trajectories that takes into consideration their spatial distance, as well as the variation of their spatial distance. The result is a robust technique that typically converges within 5 iterations when $k = 3$, and 20 iterations (on average) when $k = 23$. A sample output of this stage is shown in Figure 8.

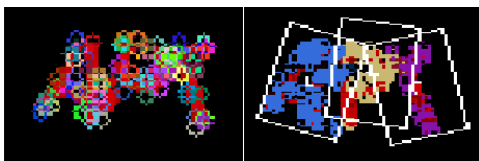


Figure 8: Left: before segmentation. Right: trajectories are marked with different colors and bounding boxes are calculated based on the center of the trajectories and the orientation of the points. The red points denote areas with no trajectories.

4.3 Codeword Extraction and Classification (Stage ④)

Given the center and orientation of each codeword character, the goal is to figure out exactly what that character is. For this task, we extract a fixed-sized area around each character (as in Figure 8), and supply that to our classification stage. Before doing so, however, we refine the patches by deleting pixels that are too close to the trajectories of adjacent characters.

As mentioned earlier, we use a neural network for classifying the refined patches. A neural network is a mathematical model or computational model that is inspired by the structure of a biological neural network. The training of a neural network is based on the notion of the possibility of learning. Given a specific task to solve, and a class of functions, learning in this context means using a set of observations to find a function which solves the task in some optimal sense.

Optimization: While the process outlined in stages ①-④ works surprisingly well, there are several opportuni-

ties for improvement. Perhaps one of the most natural extensions is to utilize a feedback mechanism to boost recognition accuracy. The idea we pursue is based on the observation that an adversary can leverage her confidence about what particular patches represent to improve her overall ability to break the captcha. Specifically, we find and block the character that we are most confident about. The basic idea is that although we may not be able to infer all the characters at once, it is very likely that we can infer some of the characters. By masking the character that we are most confident about, we can simplify the problem into one of decoding a codeword with fewer characters; which is easier to segment and recognize.



Figure 9: Iterative decoding of a captcha.

The most confident character can be found using the probability score provided by the classifier, although it is non-trivial to do so without masking out too much of the other characters. We solve this problem as follows. In order to block a character, we try to match it with templates of each character that can be gained by learning. One way to do that is to match scale-invariant feature transforms (SIFT) between the patch and a reference template. While SIFT features can deal with scaling, rotation and translation of characters, there are times when some frames have insufficient SIFT features. Our solution is to find a frame with enough features to apply SIFT, and then warp the template to mask the target character in that frame. Once found, this frame is used as the initial position in an incremental alignment approach based on KLT tracking. Essentially, we combine the benefits of SIFT and KLT to provide a video sequence where the character we are most confident about is omitted. At that point, we rerun our attack, but with one fewer character. This process is repeated until we have no characters left to decode. This process is illustrated in Figure 9.

Runtime: Our implementation is based on a collection of modules written in a mix of C++ and Matlab code. We make extensive use of the Open Source Computer Vision library (OpenCV). Our un-optimized code takes approximately 30s to decode the three characters in a MIOR captcha when the feedback loop optimization (in stage ④) is disabled. With feedback enabled, processing time increases to 250s. The bottleneck is in the incremental alignment procedure (written in Matlab).

5 Evaluation

We now discuss the results of experiments we performed on MIOR captchas. Specifically, the first set of experiments are based on video sequences downloaded off the demo page of NuCaptcha’s website. On each visit to the demo page, a captcha with a random 3-character codeword is displayed for 6 seconds before the video loops. The displayed captchas were saved locally using a Firefox plugin called NetVideoHunter. We downloaded 4500 captchas during November and December of 2011.

The collected videos contain captchas with all 19 backgrounds in use by NuCaptcha as of December 2011. In each of these videos, the backgrounds are of moving scenes (e.g., waves on a beach, kids playing baseball, etc.) and the text in the foreground either moves across the field of view or in-place. We painstakingly labeled each of the videos by hand to obtain the ground truth. We note that while NuCaptcha provides an API for obtaining captchas, we opted not to use that service as we did not want to interfere with their service in any way. In addition, our second set of experiments examine several countermeasures against our attacks, and so for ethical reasons, we opted to perform such experiments in a controlled manner rather than with any in-the-wild experimentation. These countermeasures are also evaluated in our user study (§6).

5.1 Results

The naïve attack was analyzed on 4000 captchas. Due to time constraints, the extended attack (with and without the feedback optimization) were each analyzed on a random sample of 500 captchas. To determine an appropriate training set size, we varied the number of videos as well as the number of extracted frames and examined the recognition rate. The results (not shown) show that while accuracy steadily increased with more training videos (e.g., 50 versus 100 videos), we only observed marginal improvement when the number of training patches taken from each video exceeded 1500. In the subsequent analyses, we use 300 video sequences for training (i.e., 900 codeword characters) and for each detected character, we select 2 frames containing that character (yielding 1800 training patches in total). We use dense SIFT descriptors [44] as the features for each patch (i.e., a SIFT descriptor is extracted for each pixel in the patch, and concatenated to form a feature vector). The feature vectors are used to train the neural network. For testing, we choose a *different* set of 200 captchas, almost evenly distributed among the 19 backgrounds. The accuracy of the attacks (in §4) are given in Table 1.

The result indicate that the robustness of these MIOR captchas are far weaker than one would hope. In par-

ticular, our automated attacks can completely decode the captchas *more than three quarters of the time*. In fact, our success rates are even higher than some of the OCR-based attacks on CR-still captchas [7, 19, 32, 47]. There are, however, some obvious countermeasures that designers of MIOR captchas might employ.

5.2 Mitigation

To highlight some of the tensions that exists between the security and usability of MIOR captchas, we explore a series of possible mitigations to our attacks. In order to do so, we generate video captchas that closely mimic those from NuCaptcha. In particular, we built a framework for generating videos with characters that move across a background scene with constant velocity in the horizontal direction, and move up and down harmonically. Similar to NuCaptcha, the characters of the codeword also rotate. Our framework is tunable, and all the parameters are set to the defaults calculated from the original videos from NuCaptcha (denoted *Standard*). We refer the interested reader to Appendix A for more details on how we set the parameters. Given this framework, we explore the following defenses:

- *Extended*: the codeword consists of $m > 3$ random characters moving across a dynamic scene.
- *Overlapping*: same as the *Standard* case (i.e., $m = 3$), except characters are more closely overlapped.
- *Semi-Transparent*: identical to the *Standard* case, except that the characters are semi-transparent.
- *Emerging objects*: a different MIOR captcha where the codewords are 3 characters but created using an “Emerging Images” [31] concept (see below).

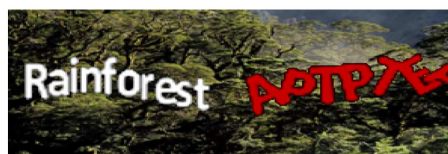


Figure 10: Extended case. Top: scrolling; bottom: in-place.

Increasing the number of random characters shown in the captcha would be a natural way to mitigate our attack. Hence, the *Extended* characters case is meant to investigate the point at which the success rate of our attacks fall

Attack Strategy	Single Character Accuracy	3-Character Accuracy
Naïve	68.5% (8216/12000)	36.3% (1450/4000)
Enhanced (<i>no feedback</i>)	90.0% (540/600)	75.5% (151/200)
Enhanced (<i>with feedback</i>)	90.3% (542/600)	77.0% (154/200)

Table 1: Reconstruction accuracy for various attacks.

below a predefined threshold. An example is shown in Figure 10. Similarly, we initially thought that increasing the overlap between consecutive characters (i.e., the *Overlapping* defense, Fig. 11) might be a viable alternative. We estimate the degree that two characters overlap by the ratio of the horizontal distance of their centers and their average width. That is, suppose that one character is 20 pixels wide, and the other is 30 pixels wide. If the horizontal distance of their centers is 20, then their overlap ratio is computed as $20/\frac{20+30}{2} = 0.8$. The smaller this overlap ratio, the more the characters overlap. A ratio of 0.5 means that the middle character is completely overlapped in the horizontal direction. In both the original captchas from NuCaptcha and our *Standard* case, the overlap ratio is 0.95 for any two adjacent characters.



Figure 11: Overlapping characters (with ratio = 0.49).

The *Semi-Transparent* defense is an attempt to break the assumption that the foreground is of constant color. In this case, foreground extraction (stage 2) will be difficult. To mimic this defense strategy, we adjust the background-to-foreground pixel ratio. An example is shown in figure 12.



Figure 12: Semi-transparent: 80% background to 20% foreground pixel ratio. (Best viewed in color.)

The final countermeasure is based on the notion of *Emerging Images* proposed by Mitra et al. [31]. Emergence refers to “the unique human ability to aggregate information from seemingly meaningless pieces, and to perceive a whole that is meaningful” [31].² The concept has been exploited in Computer Graphics to prevent

automated tracking by computers, while simultaneously allowing for high recognition rates in humans because of our remarkable visual system. We apply the concepts outlined by Mitra et al. [31] to generate captchas that are resilient to our attacks. The key differences between our implementation and the original paper is that our input is 2D characters instead of 3D objects, and we do not have the luxury of incorporating shadow information. Our Emerging captchas are constructed as follows:

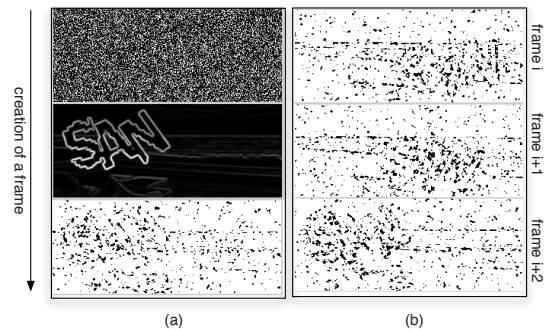


Figure 13: Emerging captcha. (a) Top: noisy background frame. Middle: derivative of foreground image. Bottom: single frame for an Emerging captcha. (b) Successive frames.

1. We build a noisy frame I_{bg} by creating an image with each pixel following a Gaussian distribution. We blur the image such that the value of each pixel is related to nearby pixels. We also include time correspondence by filtering in the time domain. That is, each frame is a mixture of a new noisy image and the last frame.
2. We generate an image I_{fg} similar to that in NuCaptcha. We then find the edges in the image by calculating the norm of derivatives of the image.
3. We combine I_{bg} and I_{fg} by creating a new image I where each pixel in I is defined as $I(x,y) := I_{bg}(x,y) * \exp(\frac{I_{fg}}{const})$, where $\exp(x)$ is the exponential function. In this way, the pixels near the boundary of characters in I are made more noisy than other pixels.
4. We define a constant threshold $t < 0$. All pixel values in I that are larger than t are made white. All

the other pixels in I are made black.

The above procedure results in a series of frames where no single frame contains the codeword in a way that is easy to segment. The pixels near the boundaries of the characters are also more likely to be blacker than other pixels, which the human visual system somehow uses to identify the structure from motion. This feat remains challenging for computers since the points near the boundaries change color randomly, making it difficult, if not impossible, to track, using existing techniques. An illustration is shown in Figure 13. To the best of our knowledge, we provide the first concrete instantiation of the notion of Emerging Images applied to captchas, as well as a corresponding lab-based usability study (§6).

We refer interested readers to <http://www.cs.unc.edu/videocaptcha/> for examples of the mitigation strategies we explored.

5.2.1 Results

We now report on the results of running attacks on captchas employing the aforementioned defenses. Figure 14 depicts the results for the *Extended* defense strategy. In these experiments, we generated 100 random captchas for each $m \in [3, 23]$. Our results clearly show that simply increasing the codeword length is not necessarily a viable defense. In fact, even at 23 characters, our success rate is still 5%, on average.

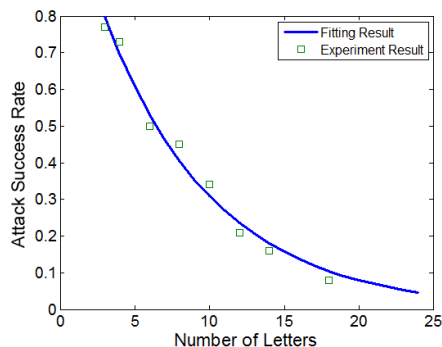


Figure 14: Attack success as a function of codeword length.

Figure 15 shows the results for the *Overlapping* defense strategy. As before, the results are averaged over 100 sequences per point. The graph shows that the success rate drops steadily as the overlap ratio decreases (denoted as “sensitivity” level in that plot). Interestingly, NuCaptcha mentions that this defense strategy is in fact one of the security features enabled by its behavioral analysis engine. The images provided on their website for the “*very secure*” mode, however, have an overlap ratio of 0.78, which our attacks would still be able to break

more than 50% of the time.³ Our success rate is still relatively high (at 5%) even when the overlap ratio is as low as 0.49. Recall that, at that point, the middle character is 100% overlapped, and others are 51% overlapped.

Figure 15 also shows the results for the *Semi-Transparent* experiment. In that case, we varied the transparency of the foreground pixel from 100% down to 20%. Even when the codewords are barely visible (to the human eye), we are still able to break the captchas 5% of the time. An example of one such captcha (with a background to foreground ratio of 80 to 20 percent) was shown earlier in Figure 12.

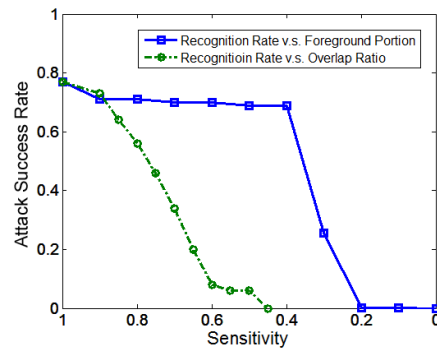


Figure 15: Attack success rate against *Overlapping* and *Semi-Transparent* defenses. Sensitivity refers to the overlap ratio (circles) or the background-to-foreground ratio (squares).

Lastly, we generated 100 captchas based on our implementation of the Emerging Images concept. It comes as no surprise that the attacks in this paper were not able to decode a single one of these challenges — precisely because these captchas were specifically designed to make optical flow tracking and object segmentation difficult. From a security perspective, these MIOR captchas are more robust than the other defenses we examined. We return to that discussion in §7.

5.2.2 Discussion

The question remains, however, whether for any of the defenses, parameters could be tuned to increase the robustness and still retain *usability*. We explore precisely that question next. That said, the forthcoming analysis raises interesting questions, especially as it relates to the robustness of captchas. In particular, there is presently no consensus on the required adversarial effort a captcha should present, or the security threshold in terms of success rate that adversaries should be held below. For example, Chellapilla et al. [8] state: “automated attacks should not be more than 0.01% successful but the human success rate should be at least 90%”. Others argue that “if it is at least as expensive for an attacker to break the

challenge by machine than it would be to pay a human to take the captcha, the test can be considered secure” [22]. Zhu et al. [53] use the metric that the bot success rate should not exceed 0.6%.

In the course of our pilot studies, it became clear that if the parameters for the *Extended*, *Overlapping*, and *Semi-Transparent* countermeasures are set too stringently (e.g., to defeat automated attacks 99% of the time), then the resulting MIOR captchas would be exceedingly difficult for humans to solve. Therefore, to better measure the tension between usability and security, we set the parameters for the videos (in §6) to values where our attacks have a 5% success rate, despite that being intolerably high for practical security. Any captcha at this parametrization, which is found to be unusable, is thus entirely unviable.

6 User study

We now report on an IRB-approved user study with 25 participants that we conducted to assess the usability of the aforementioned countermeasures. If the challenges produced by the countermeasures prove too difficult for both computers and humans to solve, then they are not viable as captcha challenges. We chose a controlled lab study because besides collecting quantitative performance data, it gave us the opportunity to collect participants’ impromptu reactions and comments, and allowed us to interview participants about their experience. This type of information is invaluable in learning *why* certain mitigation strategies are unacceptable or difficult for users and learning which strategies are deemed most acceptable. Additionally, while web-based or Mechanical Turk studies may have allowed us to collect data from more participants, such approaches lack the richness of data available when the experimenter has the opportunity to interact with the participants one-on-one. Mechanical Turk studies have previously been used in captcha research [5] when the goal of the studies are entirely performance-based. However, since we are studying new mitigation strategies, we felt that it was important to gather both qualitative and quantitative data for a more holistic perspective.

6.1 Methodology

We compared the defenses in §5.2 to a *Standard* approach which mimics NuCaptcha’s design. In these captchas the video contains scrolling text with 2-3 words in white font, followed by 3 random red characters that move along the same trajectory as the white words. Similar to NuCaptcha, the red characters (i.e., the codewords) also independently rotate as they move. For the *Extended*

strategy, we set $m = 23$. All 23 characters are continuously visible on the screen. During pilot testing, we also tried a scrolling 23-character variation of the *Extended* scheme. However, this proved extremely difficult for users to solve and they voiced strong dislike (and outrage) for the variation. For the *Overlapping* strategy, we set the ratio to be 0.49. Recall that at this ratio, the middle character is overlapped 100% of the time, and the others are 51% overlapped. For the *Semi-Transparent* strategy, we set the ratio to be 80% background and 20% foreground. For all experiments, we use the same alphabet (of 20 characters) in NuCaptcha’s original videos.

A *challenge* refers to a single captcha puzzle to be solved by the user. Each challenge was displayed on a 6-second video clip that used a canvas of size 300×126 and looped continuously. This is the same specification used in NuCaptcha’s videos. Three different HD video backgrounds (of a forest, a beach, and a sky) were used. Some examples are shown in Figure 16. Sixty challenges were generated for each variation (20 for each background, as applicable).

We also tested the *Emerging* strategy. The three-character codeword was represented by black and white pixel-based noise as described in §5.2. Sixty challenges were generated using the same video parameters as the other conditions.

The twenty-five participants were undergraduate, graduate students, staff and faculty (15 males, 10 females, mean age 26) from a variety of disciplines. A within-subjects experimental design was used, where each participant had a chance to complete a set of 10 captchas for each strategy. The order of presentation for the variations was counterbalanced according to a 5×5 Latin Square to eliminate biases from learning effects; Latin Squares are preferred over random ordering of conditions because randomization could lead to a situation where one condition is favored (e.g., appearing in the last position more frequently than other conditions, giving participants more chance to practice). Within each variation, challenges were randomly selected.

A simple web-based user interface was designed where users could enter their response in the textbox and press submit, could request a new challenge, or could access the help file. Indication of correctness was provided when users submitted their responses, and users were randomly shown the next challenge in the set. Immediately after completing the 10 challenges for a variation, users were asked to complete a paper-based questionnaire collecting their perception and opinion of that variation. At the end of the session, a brief interview was conducted to gather any overall comments. Each participant completed their session one-on-one with the experimenter. A session lasted at most 45 minutes and users were compensated \$15 for their time.



Figure 16: Three backgrounds used for the challenges, shown for the *Semi-Transparent* variant.

6.2 Data Collection

The user interface was instrumented to log each user’s interactions with the system. For each challenge, the user’s textual response, the timing information, and the outcome was recorded. A challenge could result in three possible outcomes: success, error, or skipped. Questionnaire and interview data was also collected.

6.3 Analysis

Our analysis focused on the effects of five different captcha variants on outcomes and solving times. We also analyzed and reviewed questionnaire data representing participant perceptions of the five variants. We used several statistical tests and the within-subjects design of our study impacted our choice of statistical tests; in each case the chosen test accounted for the fact that we had multiple data points from each participant. In all of our tests, we chose $p < 0.05$ as the threshold for determining statistical significance.

One-way repeated-measures ANOVAs [25] were used to evaluate aggregate differences between the means for success rates and times. When the ANOVA revealed a significant difference, we used post-hoc Tukey HSD tests [27] to determine between which pairs the differences occurred. Here, we were interested only in whether the four proposed mitigation strategies differed from the *Standard* variant, so we report only on these four cases.

Our questionnaires used Likert-scale responses to assess agreement with particular statements (1 - Strongly Disagree, 10 - Strongly Agree). To compare this ordinal data, we used the non-parametric Friedman’s Test [27]. When overall significant differences were found, we used post-hoc Pairwise Wilcoxon tests with Bonferroni correction to see which of the four proposed variants differed from the *Standard* variant.

Outcomes: Participants were presented with 10 challenges of each variant. Figure 17 shows a stacked bar graph representing the mean number of success, error, and skipped outcomes. To be identified as a *Success*, the user’s response had to be entirely correct. An *Error* occurred when the user’s response did not match the challenge’s solution. A *Skipped* outcome occurred when the participant pressed the “Get A New Challenge” but-

ton and was presented with a different challenge. We observe differences in the outcomes, with the *Standard* variant being most successful and the *Semi-Transparent* variant resulting in the most skipped outcomes.

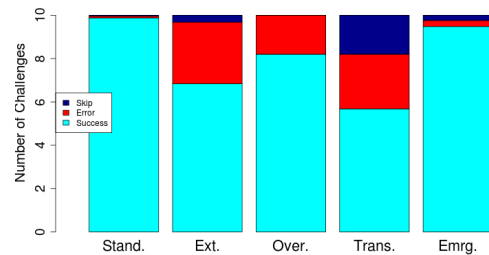


Figure 17: Mean number of success, error, and skipped outcomes for *Standard*, *Extended*, *Overlapping*, *Semi-Transparent* and *Emerging* variants, respectively.

For the purposes of our statistical tests, errors and skipped outcomes were grouped since in both cases the user was unable to solve the challenge. Each participant was given a score comprising the number of successful outcomes for each variant (out of 10 challenges).⁴

A one-way repeated-measure ANOVA showed significant differences between the five variants ($F(4, 120) = 29.12, p < 0.001$). We used post-hoc Tukey HSD tests to see whether any of the differences occurred between the *Standard* variant and any of the other four variants. The tests showed a statistically significant difference between all pairs except for the *Standard*⇔*Emerging* pair. This means that the *Extended*, *Overlapping*, and *Semi-Transparent* variants had a significantly lower number of successes than the *Standard* variant, while *Emerging* variant showed no difference.

Time to Solve: The time to solve was measured as the time between when the challenge was displayed to when the response was received. This included the time to type the answer (correctly or incorrectly), as well as the time it took the system to receive the reply (since the challenges were served from our local server, transmission time was negligible). Times for skipped challenges were not included since users made “skip” decisions very quickly and this may unfairly skew the results towards shorter mean times. We include challenges that resulted in errors because in these cases participants actively tried to

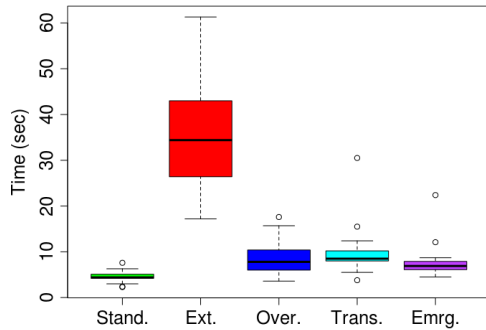


Figure 18: Time taken to solve the MBOR captchas.

solve the challenge. The time distributions are depicted in Figure 18 using boxplots. Notice that the *Extended* variant took considerably longer to solve than the others.

We examined the differences in mean times using a one-way repeated-measure ANOVA. The ANOVA showed overall significant differences between the five variants ($F(4, 120) = 112.95, p < 0.001$). Once again, we compared the *Standard* variant to the others in our post-hoc tests. Tukey HSD tests showed no significant differences between the *Standard*↔*Emerging* or *Standard*↔*Overlapping* pairs. However, significant differences were found for the *Standard*↔*Semi-Transparent* and *Standard*↔*Extended* pairs. This means that the *Semi-Transparent* and *Extended* variants took significantly longer to solve than the *Standard* variant, but the others showed no differences.

Skipped outcomes: The choice of background appears to have especially impacted the usability of the *Semi-Transparent* variant. Participants most frequently skipped challenges for the *Semi-Transparent* variant and found the Forest background especially difficult to use. Many users would immediately skip any challenge that appeared with the Forest background because the transparent letters were simply too difficult to see. For the *Semi-Transparent* variant, 35% of challenges presented on the Forest background were skipped, compared 17-18% of challenges using the other two backgrounds. Participants' verbal and written comments confirm that they found the Forest background very difficult, with some users mentioning that they could not even find the letters as they scrolled over some parts of the image.

Errors: Figure 19 shows the distribution of errors. It shows that the majority of errors were made on the middle characters of the challenge. We also examined the types of errors, and found that most were mistakes between characters that have similar appearances. The most commonly confused pairs were: S/5, P/R, E/F, V/N, C/G, and 7/T. About half of the errors for the *Extended* variant were due to confusing pairs of characters, while

the other half involved either missing letters or including extra ones. For the other variants, nearly all errors were due to confusing pairs of characters.

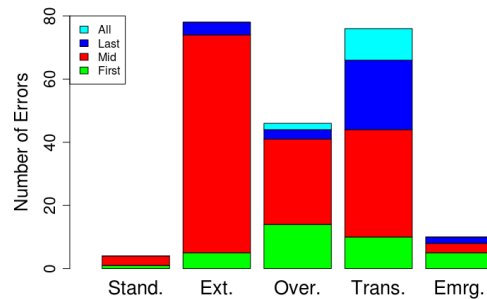


Figure 19: Location of errors within the codewords.

User perception: Immediately after completing the set of challenges for each variant, participants completed a Likert-scale questionnaire to collect their opinion and perception of that variant. For each variant, participants were asked to rate their agreement with the following statements:

1. It was easy to accurately solve the challenge
2. The challenges were easy to understand
3. This captcha mechanism was pleasant to use
4. This captcha mechanism is more prone to mistakes than traditional text-based captchas

Figure 20 shows boxplots representing users' responses. Since Q.4 was negatively worded, responses were inverted for easier comparisons. In all cases, higher values on the y-axis indicate a more favorable response.

The results show that users clearly preferred the *Standard* variant and rated the others considerably lower on all subjective measures. Friedman's Tests showed overall significant differences for each question ($p < 0.001$). Pairwise Wilcoxon Tests with Bonferroni correction were used to assess differences between the *Standard* variant and each of the other variants. Significant differences were found between each pair compared. The only exceptions are that users felt that the *Extended* and *Emerging* variants were no more difficult to understand (Question 2) than the *Standard* variant. This result appears to contradict the results observed in Figure 20 and we believe that this is because the Wilcoxon test compares ranks rather than means or medians.

Comments: Participants had the opportunity to provide free-form comments about each variant and offer verbal comments to the experimenter. Samples are included in Appendix B. Participants clearly preferred the *Standard* variant, and most disliked the *Extended* variant.

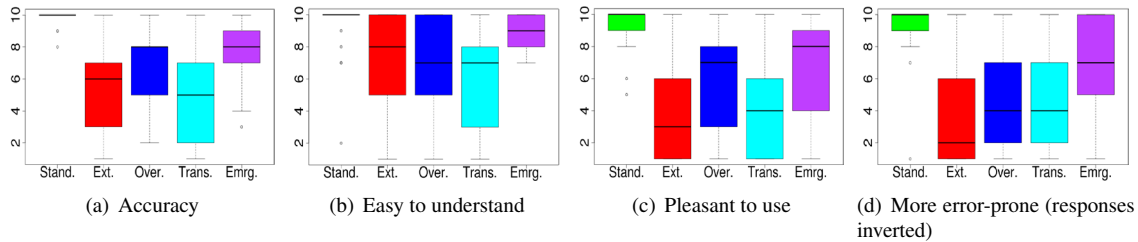


Figure 20: Likert-scale responses: 1 is most negative, 10 is most positive.

Of the remaining schemes, the *Emerging* variant seemed most acceptable although it also had its share of negative reactions (e.g., one subject found it to be hideous).

7 Summary and Concluding Remarks

Our attack inherently leverages the temporal information in the moving-image object recognition (MIOR) captchas, and also exploits the fact that only object recognition of known objects is needed. Our methods also rely on a reasonably consistent appearance or slowly varying appearance over time. That said, they can be applied to any set of known objects or narrowly defined objects under affine transformations that are known to work well with detection methods in computer vision [45]. For the specific case of NuCaptcha, we showed that not only are there inherent weaknesses in the current MIOR captcha design, but that several obvious countermeasures (e.g., extending the length of the codeword) are not viable attack countermeasures. More importantly, our work highlights the fact that the choice of underlying hard problem by NuCaptcha’s designers was misguided; its particular implementation falls into a solvable subclass of computer vision problems.

In the case of emergent captchas, our attacks fail for two main reasons. First, in each frame there are not enough visual cues that help distinguish the characters from the background. Second, the codewords have no temporally consistent appearance. Combined, these two facts pose significant challenges to existing computer vision methods, which typically assume reasonably consistent appearance and visually distinctive foregrounds [52]. Nevertheless, our user study showed that people had little trouble solving these captchas. This bodes well for emergent captchas—per today’s attacks.

Looking towards the future, greater robustness would result if MIOR captchas required automated attacks to perform classification, categorization of classes with large inner class variance, or to identify higher level semantics to understand the presented challenge. Consider, for example, the case where the user is presented with two objects (a person and a truck) at the same scale, and

asked to identify which one is larger. To succeed, the automated attack would need to determine the objects (without prior knowledge of what the objects are of) and then understand the relationship. Humans can perform this task because of the inherent priors learned in daily life, but this feat remains a daunting problem in computer vision. Therefore, this combination seems to offer the right balance and underscores the ideas put forth by Naor [34] and von Ahn et al. [1]—i.e., it is prudent to employ hard (and useful) underlying AI problems in captchas since it leads to a win-win situation: either the captcha is not broken and there is a way to distinguish between humans and computers, or it is broken and a useful problem is solved.

Acknowledgments

The authors thank Pierre Georgel, Joseph Tighe, and Avi Rubin for insightful discussions about this work, and for valuable feedback on an earlier draft of this manuscript. We are also especially grateful to Fletcher Fairey (of the Office of University Counsel at Chapel Hill), and Cindy Cohn and Marcia Hofmann (of the Electronic Frontier Foundation) for their guidance and assistance in making our findings available to NuCaptcha in a timely manner.

Sonia Chiasson holds a Canada Research Chair in Human Oriented Computer Security and Paul Van Oorschot holds a Canada Research Chair in Authentication and Computer Security; both acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC) for funding the Chairs and Discovery Grants, as well as funding from NSERC ISSNet. This work is also supported by the National Science Foundation (NSF) under award number 1148895.

Notes

¹In the case where the foreground characters have varying appearance, we simply use multiple modes.

²Readers can view videos of the Emerging Images concept [31] at http://graphics.stanford.edu/~niloy/research/emergence/emergence_image_siga_09.html.

³See the Security Features discussed at <http://www.nucaptcha.com/features/security-features>, 2012.

⁴One participant opted to view only six challenges in each of the *Extended* and *Emerging* variants. We count the remaining four as skips.

References

- [1] L. V. Ahn, M. Blum, N. J. Hopper, and J. Langford. CAPTCHA: using hard AI problems for security. In *Eurocrypt*, pages 294–311, 2003.
- [2] A. Basso and F. Bergadano. Anti-bot strategies based on human interactive proofs. In P. Stavroulakis and M. Stamp, editors, *Handbook of Information and Communication Security*, pages 273–291. Springer, 2010.
- [3] E. Bursztein. How we broke the NuCaptcha video scheme and what we proposed to fix it. See <http://elie.im/blog/security/how-we-broke-the-nucaptcha-video-scheme-and-what-we-propose-to-fix-it/>. Accessed March, 2012.
- [4] E. Bursztein and S. Bethard. DeCAPTCHA: breaking 75% of ebay audio CAPTCHAs. In *Proceedings of the 3rd USENIX Workshop on Offensive Technologies*, 2009.
- [5] E. Bursztein, S. Bethard, C. Fabry, J. C. Mitchell, and D. Jurafsky. How good are humans at solving CAPTCHAs? a large scale evaluation. In *IEEE Symposium on Security and Privacy*, pages 399–413, 2010.
- [6] E. Bursztein, R. Beauxis, H. Paskov, D. Perito, C. Fabry, and J. C. Mitchell. The failure of noise-based non-continuous audio CAPTCHAs. In *IEEE Symposium on Security and Privacy*, pages 19–31, 2011.
- [7] E. Bursztein, M. Martin, and J. Mitchell. Text-based CAPTCHA strengths and weaknesses. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 125–138, 2011.
- [8] K. Chellapilla, K. Larson, P. Y. Simard, and M. Czerwinski. Designing human friendly human interaction proofs (hips). In *ACM Conference on Human Factors in Computing Systems*, pages 711–720, 2005.
- [9] K. Chellapilla, K. Larson, P. Y. Simard, and M. Czerwinski. Building segmentation based human-friendly human interaction proofs (hips). In *Human Interactive Proofs, Second International Workshop*, pages 1–26, 2005.
- [10] J. Cui, W. Zhang, Y. Peng, Y. Liang, B. Xiao, J. Mei, D. Zhang, and X. Wang. A 3-layer Dynamic CAPTCHA Implementation. In *Workshop on Education Technology and Computer Science*, volume 1, pages 23–26, march 2010.
- [11] J.-S. Cui, J.-T. Mei, X. Wang, D. Zhang, and W.-Z. Zhang. A CAPTCHA Implementation Based on 3D Animation. In *International Conference on Multimedia Information Networking and Security*, volume 2, pages 179–182, nov. 2009.
- [12] J.-S. Cui, J.-T. Mei, W.-Z. Zhang, X. Wang, and D. Zhang. A CAPTCHA Implementation Based on Moving Objects Recognition Problem. In *International Conference on E-Business and E-Government*, pages 1277–1280, may 2010.
- [13] J. J. DiCarlo and D. D. Cox. Untangling invariant object recognition. *Trends in Cognitive Sciences*, 11:333–341, 2007.
- [14] J. Driver and G. Baylis. Edge-assignment and figure-ground segmentation in short-term visual matching. *Cognitive Psychology*, 31:248–306, 1996.
- [15] M. Egele, L. Bilge, E. Kirda, and C. Kruegel. Captcha smuggling: hijacking web browsing sessions to create captcha farms. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1865–1870, 2010.
- [16] J. Elson, J. R. Douceur, J. Howell, and J. Saul. Asirra: a CAPTCHA that exploits interest-aligned manual image categorization. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 366–374, 2007.
- [17] M. Fischler and R. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Comm. of the ACM*, 24(6):381–395, 1981.
- [18] N. Friedman and S. Russell. Image segmentation in video sequences: A probabilistic approach. *University of California, Berkeley*, 94720, 1776.
- [19] P. Golle. Machine learning attacks against the Asirra CAPTCHA. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 535–542, 2008.
- [20] K. Grill-Spector and N. Kanwisher. Visual recognition: as soon as you know it is there, you know what it is. *Psychological Science*, 16(2):152–160, 2005.
- [21] C. Harris and M. Stephens. A combined corner and edge detection. In *Proceedings of The Fourth Alvey Vision Conference*, volume 15, pages 147–151, 1988.
- [22] J. M. G. Hidalgo and G. Alvarez. CAPTCHAs: An Artificial Intelligence Application to Web Security. *Advances in Computers*, 83:109–181, 2011.
- [23] A. Jain, M. Murty, and P. Flynn. Data clustering: a review. *ACM computing Surveys*, 31(3):264–323, 1999.
- [24] K. A. Kluever and R. Zanibbi. Balancing usability and security in a video CAPTCHA. In *Proceedings of the 5th Symposium on Usable Privacy and Security*, pages 1–14, 2009.
- [25] J. Lazar, J. H. Feng, and H. Hochheiser. *Research Methods in Human-Computer Interaction*. John Wiley and Sons, 2010.
- [26] W.-H. Liao and C.-C. Chang. Embedding information within dynamic visual patterns. In *Multimedia and Expo, IEEE International Conference on*, volume 2, pages 895–898, june 2004.
- [27] R. Lowry. *Concepts and Applications of Inferential Statistics*. Vassar College, <http://faculty.vassar.edu/lowry/webtext.html>, 1998.
- [28] B. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 674–679, 1981.
- [29] D. Marr. *Vision: a computational investigation into the human representation and processing of visual information*. W. H. Freeman, San Francisco, 1982.
- [30] D. Marr and T. Poggio. A computational theory of human stereo vision. *Proceedings of the Royal Society of London. Series B, Biological Sciences*, 204(1156):301–328, 1979.
- [31] N. J. Mitra, H.-K. Chu, T.-Y. Lee, L. Wolf, H. Yeshurun, and D. Cohen-Or. Emerging images. *ACM Transactions on Graphics*, 28(5), 2009.
- [32] G. Mori and J. Malik. Recognizing objects in adversarial clutter: breaking a visual CAPTCHA. In *Computer Vision and Pattern Recognition*, volume 1, pages 134–141, june 2003.

- [33] M. Motoyama, K. Levchenko, C. Kanich, D. McCoy, G. M. Voelker, and S. Savage. Re: CAPTCHAs-understanding CAPTCHA-solving services in an economic context. In *USENIX Security Symposium*, pages 435–462, 2010.
- [34] M. Naor. Verification of a human in the loop or identification via the Turing test, 1996.
- [35] NuCaptcha. Whitepaper: NuCaptcha & Traditional Captcha, 2011. <http://nucaptcha.com>.
- [36] A. Oliva and A. Torralba. The role of context in object recognition. *Trends in Cognitive Sciences*, 11(12):520 – 527, 2007.
- [37] S. Ray and R. Turi. Determination of number of clusters in k-means clustering and application in colour image segmentation. In *Proceedings of the International conference on advances in pattern recognition and digital techniques*, pages 137–143, 1999.
- [38] M. Shirali-Shahreza and S. Shirali-Shahreza. Motion CAPTCHA. In *Conference on Human System Interactions*, pages 1042–1044, May 2008.
- [39] P. Simard, D. Steinkraus, and J. Platt. Best practices for convolutional neural networks applied to visual document analysis. In *Proceedings of the Seventh International Conference on Document Analysis and Recognition*, volume 2, pages 958–962, 2003.
- [40] Y. Soudjani and D. Grizalis. Audio CAPTCHA: Existing solutions assessment and a new implementation for voip telephony. *Computers & Security*, 29(5):603–618, 2010.
- [41] S. Thorpe, D. Fize, and C. Marlot. Speed of processing in the human visual system. *Nature*, 381(6582):520–522, 1996.
- [42] S. Ullman. Computational studies in the interpretation of structure and motion: Summary and extension. In *Human and Machine Vision*. Academic Press, 1983.
- [43] S. Ullman. *High-Level Vision: Object Recognition and Visual Cognition*. The MIT Press, 1 edition, July 2000.
- [44] A. Vedaldi and B. Fulkerson. Vfeat: An open and portable library of computer vision algorithms. In *Proceedings of the international conference on Multimedia*, pages 1469–1472, 2010.
- [45] P. A. Viola and M. J. Jones. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition*, 2001.
- [46] L. von Ahn, M. Blum, and J. Langford. Telling humans and computers apart automatically. *Commun. ACM*, 47:56–60, February 2004.
- [47] J. Yan and A. S. E. Ahmad. Breaking visual CAPTCHAs with naive pattern recognition algorithms. In *ACSAC*, pages 279–291, 2007.
- [48] J. Yan and A. S. E. Ahmad. A low-cost attack on a microsoft CAPTCHA. In *ACM Conference on Computer and Communications Security*, pages 543–554, 2008.
- [49] J. Yan and A. S. E. Ahmad. Usability of CAPTCHAs or usability issues in CAPTCHA design. In *SOUPS*, pages 44–52, 2008.
- [50] J. Yan and A. El Ahmad. CAPTCHA robustness: A security engineering perspective. *Computer*, 44(2):54–60, feb. 2011.
- [51] J. Yan and M. Pollefeys. Articulated motion segmentation using RANSAC with priors. *Dynamical Vision*, pages 75–85, 2007.
- [52] A. Yilmaz, O. Javed, and M. Shah. Object tracking: A survey. *ACM Comput. Surv.*, 38, December 2006.
- [53] B. B. Zhu, J. Yan, Q. Li, C. Yang, J. Liu, N. Xu, M. Yi, and K. Cai. Attacks and design of image recognition CAPTCHAs. In *ACM Conference on Computer and Communications Security*, pages 187–200, 2010.

A Parameters for video generation

Similar to NuCaptcha’s videos, our sequences have letters that move across a background scene with constant velocity in the horizontal direction, and move up and down harmonically (i.e., $y(t) = A * \sin(\omega t + \psi)$, y is the vertical position of the letter, t is the frame id, and A, ω, ψ are adjustable parameters). The horizontal distance between two letters is a function of their average width. If their widths are $width_1, width_2$, the distance between their centers are set to be $\alpha * \frac{width_1 + width_2}{2}$, where α is an adjustable parameter that indicates how much two letters overlap. Our letters also rotate and loop around. The angle θ to which a letter rotates is also decided by a sin function $\theta = \theta_0 * \sin(\omega_\theta t + \psi_\theta)$, where $\theta_0, \omega_\theta, \psi_\theta$ are adjustable parameters. For the standard case, we set the parameters the same as in NuCaptcha’s videos. We adjust these parameters based on the type of defenses we explore (in Section 5.2).

B Comments from User Study

Table 2 highlights some of the free-form responses written on the questionnaire used in our study.

Variant	Comments
<i>Standard</i>	<ul style="list-style-type: none"> - User friendly - It was too easy - Much easier than traditional captchas
<i>Extended</i>	<ul style="list-style-type: none"> - My mother would not be able to solve these - Giant Pain in the Butt! Sheer mass of text was overwhelming and I got lost many times - Too long! I would prefer a shorter text - It was very time consuming, and is very prone to mistakes
<i>Overlapping</i>	<ul style="list-style-type: none"> - Letters too bunched – several loops needed to decipher - Takes longer because I had to wait for the letter to move a bit so I can see more of it - Still had a dizzying affect. Not pleasant - Some characters were only partially revealed, ‘Y’ looked like a ‘V’
<i>Semi-Transparent</i>	<ul style="list-style-type: none"> - Tree background is unreadable, any non-solid background creates too much interference - With some backgrounds I almost didn’t realize there were red letters - It was almost faded and very time consuming. I think I made more mistakes in this mechanism
<i>Emerging</i>	<ul style="list-style-type: none"> - Not that complicated - I’d feel dizzy after staring at it for more than 1 min - It was hideous! Like an early 2000s website. But it did do the job. It made my eyes feel ‘fuzzy’ after a while - It was good, better than the challenges with line through letters

Table 2: Sample participant comments for each variant

How Does Your Password Measure Up? The Effect of Strength Meters on Password Creation

Blase Ur, Patrick Gage Kelley, Saranga Komanduri, Joel Lee, Michael Maass,
Michelle L. Mazurek, Timothy Passaro, Richard Shay, Timothy Vidas,
Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor
Carnegie Mellon University
{bur, pgage, sarangak, jlee, mmaass, mmazurek, tpassaro,
rshay, tvidas, lbauer, nicolasc, lorrie}@cmu.edu

Abstract

To help users create stronger text-based passwords, many web sites have deployed password meters that provide visual feedback on password strength. Although these meters are in wide use, their effects on the security and usability of passwords have not been well studied.

We present a 2,931-subject study of password creation in the presence of 14 password meters. We found that meters with a variety of visual appearances led users to create longer passwords. However, significant increases in resistance to a password-cracking algorithm were only achieved using meters that scored passwords stringently. These stringent meters also led participants to include more digits, symbols, and uppercase letters.

Password meters also affected the act of password creation. Participants who saw stringent meters spent longer creating their password and were more likely to change their password while entering it, yet they were also more likely to find the password meter annoying. However, the most stringent meter and those without visual bars caused participants to place less importance on satisfying the meter. Participants who saw more lenient meters tried to fill the meter and were averse to choosing passwords a meter deemed “bad” or “poor.” Our findings can serve as guidelines for administrators seeking to nudge users towards stronger passwords.

1 Introduction

While the premature obituary of passwords has been written time and again [22, 25], text passwords remain ubiquitous [15]. Unfortunately, users often create passwords that are memorable but easy to guess [2, 25, 26]. To combat this behavior, system administrators employ a number of measures, including system-assigned passwords and stringent password-composition policies. System-assigned passwords can easily be made difficult to guess, but users often struggle to remember them [13]

or write them down [28]. Password-composition policies, sets of requirements that every password on a system must meet, can also make passwords more difficult to guess [6, 38]. However, strict policies can lead to user frustration [29], and users may fulfill requirements in ways that are simple and predictable [6].

Another measure for encouraging users to create stronger passwords is the use of password meters. A password meter is a visual representation of password strength, often presented as a colored bar on screen. Password meters employ suggestions to assist users in creating stronger passwords. Many popular websites, from Google to Twitter, employ password meters.

Despite their widespread use, password meters have not been well studied. This paper contributes what we believe to be the first large-scale study of what effect, if any, password meters with different scoring algorithms and visual components, such as color and size, have on the security and usability of passwords users create.

We begin by surveying password meters in use on popular websites. Drawing from our observations, we create a control condition without a meter and 14 conditions with meters varying in visual features or scoring algorithm. The only policy enforced is that passwords contain at least eight characters. However, the meter nudges the user toward more complex or longer passwords.

We found that using any of the tested password meters led users to create passwords that were statistically significantly longer than those created without a meter. Meters that scored passwords more stringently led to even longer passwords than a baseline password meter. These stringent meters also led participants to include a greater number of digits, symbols, and uppercase letters.

We also simulated a state-of-the-art password-cracking algorithm [38] and compared the percentage of passwords cracked in each condition by adversaries making 500 million, 50 billion, and 5 trillion guesses. Passwords created without a meter were cracked at a higher rate than passwords in any of the 14 conditions with me-

ters, although most differences were not statistically significant. Only passwords created in the presence of the two stringent meters with visual bars were cracked at a significantly lower rate than those created without a meter. None of the conditions approximating meters we observed in the wild significantly increased cracking resistance, suggesting that currently deployed meters are not sufficiently aggressive. However, we also found that users have expectations about good passwords and can only be pushed so far before aggressive meters seem to annoy users rather than improve security.

We next review related work and provide background in Section 2. We then survey popular websites' password meters in Section 3 and present our methodology in Section 4. Section 5 contains results related to password composition, cracking, and creation, while Section 6 summarizes participants' attitudes. We discuss these findings in Section 7 and conclude in Section 8.

2 Related Work

Prior work related to password meters has focused on password scoring rather than how meters affect the security and usability of passwords users create. We summarize this prior work on password scoring, and we then discuss more general work on the visual display of indicators. In addition, we review work analyzing security and usability tradeoffs in password-composition policies. Finally, we discuss the "guessability" metric we use to evaluate password strength.

2.1 Password Meters

Algorithms for estimating password strength have been the focus of prior work. Sotirakopoulos et al. investigated a password meter that compares the strength of a user's password with those of other users [31]. Castelluccia et al. argued that traditional rule-based password meters lack sufficient complexity to guide users to diverse passwords, and proposed an adaptive Markov algorithm that considers n-gram probabilities in training data [7]. In contrast, we use simple rule-based algorithms to estimate strength, focusing on how meters affect the usability and security of the passwords users create. To our knowledge, there has been no formal large-scale study of interface design for password meters.

Many password meters guide users toward, but do not strictly require, complex passwords. This approach reflects the behavioral economics concept of nudging or soft paternalism [24, 34]. By helping users make better decisions through known behavioral patterns and biases, corporations, governments, and other entities have induced a range of behavioral changes from investing more toward retirement to eating more fruit.

2.2 Visual Display of Indicators

While the literature on visual design for password meters is sparse, there is a large corpus of work in information design generally. For instance, researchers have studied progress indicators in online questionnaires, finding that indicators can improve user experience if the indicator shows faster progress than a user anticipated. However, progress that lags behind a user's own expectations can cause the user to abandon the task at hand [8].

Much of the past work on small meters has focused on physical and virtual dashboards [11]. Information design has also been studied in consumer-choice situations, such as nutrition labels [19] and over-the-counter drug labels, focusing on whitespace, font size, and format [40].

2.3 Password-Composition Policies

In this paper, we examine security and usability tradeoffs related to nudging users with password meters, rather than imposing strict requirements. Significant work has been done evaluating tradeoffs for enforced password-composition policies.

Without intervention, users tend to create simple passwords [12, 23, 33, 41]. Many organizations use password-composition policies that force users to select more complex passwords to increase password strength. However, users are expected to conform to these policies in predictable ways, potentially reducing password strength [6]. Although prior work has shown that password-composition policies requiring more characters or more character classes can improve resistance to automated guessing attacks, many passwords that meet common policies remain vulnerable [18, 26, 37, 38]. Furthermore, strict policies can frustrate users, inhibit their productivity, and lead users to write their passwords down [1, 14, 16, 21, 32].

2.4 Measuring Guessability

In this work, we use "guessability," or resistance to automated password-cracking attacks, to evaluate the strength of passwords. Guessability cannot be measured as a single statistic for a set of passwords; instead, a given algorithm, with a given set of parameters and training, will crack some percentage of the passwords after a given number of guesses. Weir et al. argue that guessability is a more accurate measure of password strength than the more commonly used entropy metric [38]. Dell'Amico et al. [9], Bonneau [3], and Castelluccia et al. [7] have also used guessability as a metric. We measure guessability using a guess-number calculator, which computes how many guesses a given cracking algorithm will require to crack a specific password without running the algorithm itself [18].

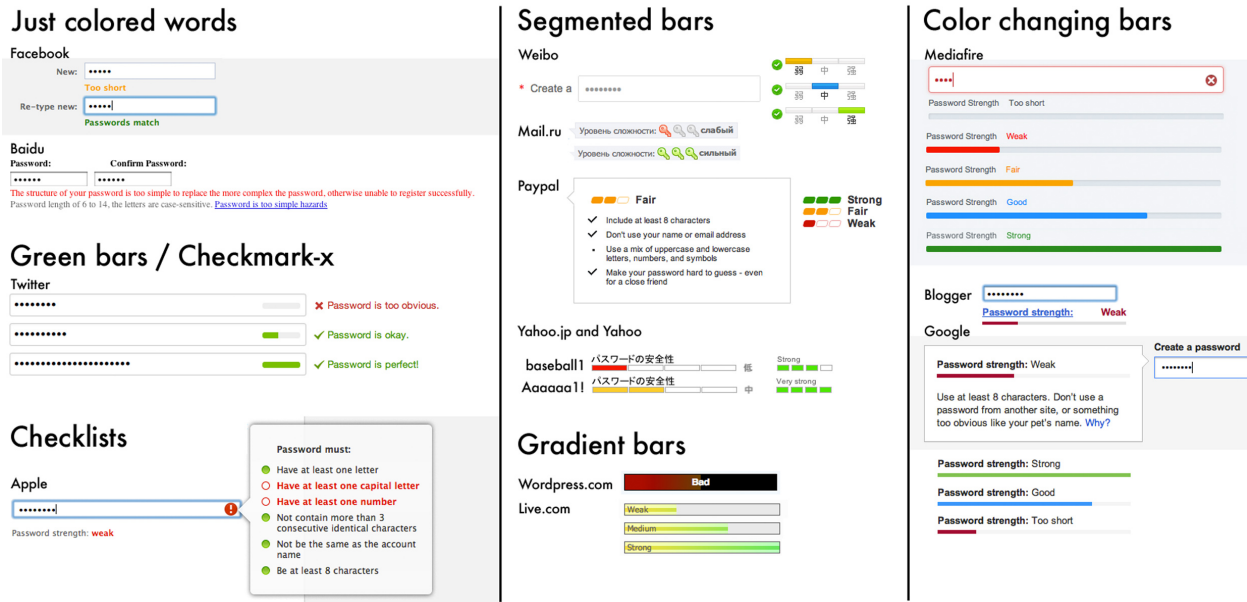


Figure 1: A categorized assortment of the 46 unique indicators we found across Alexa’s 100 most visited global sites.

3 Password Meters “In the Wild”

To understand how password meters are currently used, we examined Alexa’s 100 most visited global sites (collected January 2012). Among these 100 sites, 96 allowed users to register and create a password. Of these 96, 70 sites (73%) gave feedback on a user’s password based either on its length or using a set of heuristics. The remaining 26 sites (27%) provided no feedback. In some cases, all sites owned by the same company used the same meter; for example, Google used the same meter on all 27 of its affiliates that we examined. In other cases, the meters varied; for example, ebay.de used a different mechanism than ebay.com. Removing duplicate indicators and sites without feedback, there were 46 unique indicators. Examples of these indicators are shown in Figure 1.

Indicators included bar-like meters that displayed strength (23, 50%); checkmark-or-x systems (19, 41.3%); and text, often in red, indicating invalid characters and too-short passwords (10, 21.2%). Sites with bar-like meters used either a progress-bar metaphor (13, 56.5%) or a segmented-box metaphor (8, 34.8%). Two sites presented a bar that was always completely filled but changed color (from red to green or blue) as password complexity increased. Three other sites used meters colored with a continuous gradient that was revealed as users typed. Sites commonly warned about insecure passwords using the words “weak” and “bad.”

We examined scoring mechanisms both by reading the Javascript source of the page, when available, and by testing sample passwords in each meter. Across all

meters, general scoring categories included password length, the use of numbers, uppercase letters, and special characters, and the use of blacklisted words. Most meters updated dynamically as characters were typed.

Some meters had unique visual characteristics. Twitter’s bar was always green, while the warning text changed from red to green. Twitter offered phrases such as “Password could be more secure” and “Password is Perfect.” The site mail.ru had a three-segment bar with key-shaped segments, while rakuten.co.jp had a meter with a spring-like animation.

We found some inconsistencies across domains. Both yahoo.com and yahoo.co.jp used a meter with four segments; however, the scoring algorithm differed, as shown in Figure 1. Google used the same meter across all affiliated sites, yet its meter on blogger.com scored passwords more stringently.

4 Methodology

We conducted a two-part online study of password-strength meters, recruiting participants through Amazon’s Mechanical Turk crowdsourcing service (MTurk). Participants, who were paid 55 cents, needed to indicate that they were at least 18 years old and use a web browser with JavaScript enabled. Participants were assigned round-robin to one of 15 conditions, detailed in Section 4.2. We asked each participant to imagine that his or her main email provider had changed its password requirements, and that he or she needed to create a new password. We then asked the participant to create a pass-

word using the interface shown in Figure 2.

Passwords needed to contain at least eight characters, but there were no other requirements. The participant was told he or she would be asked to return in a few days to log in with the password. He or she then completed a survey about the password-creation experience and was asked to reenter his or her password at the end.

Two days later, participants received an email through MTurk inviting them to return for a bonus payment of 70 cents. Participants were asked to log in again with their password and to take another survey about how they handled their password.

4.1 Password-Scoring Algorithms

Password-strength meters utilize a scoring function to judge the strength of a password, displaying this score through visual elements. We assigned passwords a score using heuristics including the password’s length and the character classes it contained. While alternative approaches to scoring have been proposed, as discussed in Section 2, judging a password only on heuristics obviates the need for a large, existing dataset of passwords and can be implemented quickly in Javascript. These heuristics were based on those we observed in the wild.

In our scoring system, a score of 0 points represented a blank password field, while a score of 100 points filled the meter and displayed the text “excellent.” We announced our only password-composition policy in bold text to the participant as an “8-character minimum” requirement. However, we designed our scoring algorithm to assign passwords containing eight lowercase letters a score of 32, displaying “bad.” To receive a score of 100 in most conditions, participants needed to meet one of two policies identified as stronger in the literature [6,21], which we term *Basic16* and *Comprehensive8*. Unless otherwise specified by the condition, passwords were assigned the larger of their *Basic16* and *Comprehensive8* scores. Thus, a password meeting either policy would fill the meter. Each keystroke resulted in a recalculation of the score and update of the meter.

The *Basic16* policy specifies that a password contain at least 16 characters, with no further restrictions. In our scoring system, the first 8 characters entered each received 4 points, while all subsequent characters received 8 points. Thus, passwords such as *aaaaaaaaaaaaaaaa*, *WdH5\$87T5c#hgfd&*, and *passwordpassword* would all fill the meter with scores of exactly 100 points.

The second policy, *Comprehensive8*, specifies that a password contain at least eight characters, including an uppercase letter, a lowercase letter, a digit, and a symbol. Furthermore, this password must not be in the OpenWall Mangled Wordlists, which is a cracking dictionary.¹ In

¹<http://www.openwall.com/wordlists/>

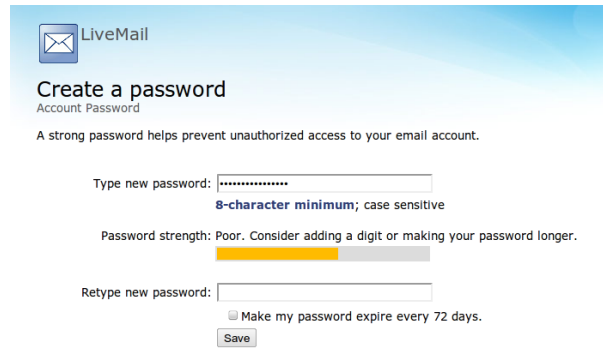


Figure 2: An example of the password creation page. The password meter’s appearance and scoring varied by condition.

our scoring system, 4 points were awarded for each character in the password, and an additional 17 points were awarded each for the inclusion of an uppercase character, a digit, and a symbol; 17 points were deducted if the password contained no lowercase letters. A second unique digit, symbol, or uppercase character would add an additional 8 points, while a third would add an additional 4 points. Passing the dictionary check conferred 17 points. Therefore, passwords such as *P4\$word*, *gT7fas#g*, and *N!ck1ebk* would fill the meter with a score of exactly 100. In addition, passwords that were hybrids of the two policies, such as a 13-character password meeting *Comprehensive8* except containing no symbols, could also fill the meter.

4.2 Conditions

Our 15 conditions fall into four main categories. The first category contains the two conditions to which we compared the others: having no password meter and having a baseline password meter. Conditions in the next category differ from the baseline meter in only one aspect of visual presentation, but the scoring remains the same. In contrast, conditions in the third category have the same visual presentation as the baseline meter, but are scored differently. Finally, we group together three conditions that differ in multiple dimensions from the baseline meter. In addition, we collectively refer to *half-score*, *one-third-score*, *text-only half-score*, and *text-only half-score* as the *stringent* conditions throughout the paper. Each participant was assigned round-robin to one condition.

4.2.1 Control Conditions

No meter. This condition, our control, uses no visual feedback mechanism. 26 of the Alexa Top 100 websites provided no feedback on password strength, and this condition allows us to isolate the effect of the visual feedback in our other conditions.

Baseline meter. This condition represents our default password meter. The score is the higher of the scores derived from comparing the password to the Basic16 and Comprehensive8 policies, where a password meeting either policy fills the bar. The color changes from red to yellow to green as the score increases. We also provide a suggestion, such as “Consider adding a digit or making your password longer.” This condition is a synthesis of meters we observed in the wild.

4.2.2 Conditions Differing in Appearance

Three-segment. This condition is similar to *baseline meter*, except the continuously increasing bar is replaced with a bar with three distinct segments, similar to meters from Google and Mediafire.

Green. This condition is similar to *baseline meter*, except instead of changing color as the password score increases, the bar is always green, like Twitter’s meter.

Tiny. This condition is similar to *baseline meter*, but with the meter’s size decreased by 50% horizontally and 60% vertically, similar to the size of Google’s meter.

Huge. This condition is similar to *baseline meter*, but with the size of the meter increased by 50% horizontally and 120% vertically.

No suggestions. This condition is similar to *baseline meter*, but does not offer suggestions for improvement.

Text-only. This condition contains all of the text of *baseline meter*, but has no visual bar graphic.

4.2.3 Conditions Differing in Scoring

Half-score. This condition is similar to *baseline meter*, except that the password’s strength is displayed as if it had received half the rating. A password that would fill the *baseline meter* meter only fills this condition’s meter half way, allowing us to study nudging the participant toward a stronger password. A password with 28 characters, or one with 21 characters that included five different uppercase letters, five different digits, and five different symbols, would fill this meter.

One-third-score. This condition is similar to *half-score*, except that the password’s strength is displayed as if it had received one-third the rating. A password that would fill the *baseline meter* meter only fills one-third of this condition’s meter. A password containing 40 characters would fill this meter.

Nudge-16. This condition is similar to *baseline meter*, except that only the password score for the Basic16 policy is calculated, allowing us to examine nudging the user toward a specific password policy.

Nudge-comp8. As with *nudge-16*, this condition is similar to *baseline meter*, except that only the password score for Comprehensive8 is calculated.

4.2.4 Conditions Differing in Multiple Ways

Text-only half-score. As with *text-only*, this condition contains all of the text of *baseline meter*, yet has no bar. Furthermore, like *half-score*, the password’s strength is displayed as if it had received only half the score.

Bold text-only half-score. This condition mirrors *text-only half-score*, except the text is displayed in bold.

Bunny. In place of a bar, the password score is reflected in the speed at which an animated Bugs Bunny dances. When the score is 0, he stands still. His speed increases with the score; at a score of 100, he dances at 20 frames per second; at a score of 200, he reaches his maximum of 50 frames per second. This condition explores a visual feedback mechanism other than a traditional bar.

4.3 Mechanical Turk

Many researchers have examined using MTurk workers for human-subjects research and found it to be a convenient source of high-quality data [5, 10, 20, 35]. MTurk enables us to have a high volume of participants create passwords, on a web site we control, with better population diversity than would be available in an on-campus laboratory environment [5]. MTurk workers are also more educated, more technical, and younger than the general population [17].

4.4 Statistical Tests

All statistical tests use a significance level of $\alpha = .05$. For each variable, we ran an omnibus test across all conditions. We ran pairwise contrasts comparing each condition to our two control conditions, *no meter* and *baseline meter*. In addition, to investigate hypotheses about the ways in which conditions varied, we ran planned contrasts comparing *tiny* to *huge*, *nudge-16* to *nudge-comp8*, *half-score* to *one-third-score*, *text-only* to *text-only half-score*, *half-score* to *text-only half-score*, and *text-only half-score* to *bold text-only half-score*. If a pairwise contrast is not noted as significant in the results section, it was not found to be statistically significant. To control for Type I error, we ran contrasts only where the omnibus test was significant. Further, we corrected contrasts for multiple testing, accounting for the previous contrasts. We applied multiple testing correction to the p-values of the omnibus tests when multiple tests were run on similar variables, such as the Likert response variables measuring user attitudes.

We analyzed quantitative data using Kruskal-Wallis for the omnibus cases and Mann-Whitney U for the pairwise cases. These tests, identified in our results as K-W and MWU, respectively, are analogues of the ANOVA and *t*-tests without the assumption of normality. We analyze categorical data for equality of proportions with χ^2

tests for both the omnibus and pairwise cases. All multiple testing correction used the Holm-Bonferroni method, indicated as HC throughout the paper.

4.5 Calculating Guess Numbers

We evaluated the strength of passwords created in each condition using a guess-number calculator (see Section 2.4), allowing us to approximate passwords' resistance to automated cracking. Using a password guess calculator similar to that used by Kelley et al. [18], we calculate the guessability of passwords in three different attack scenarios. This calculator simulates the password-cracking algorithm devised by Weir et al. [39], which makes guesses based on the structures, digits, symbols, and alphabetic strings in its training data. The calculator was set to only consider guesses with minimum length 8. For training, we used several "public" datasets, including leaked sets of cracked passwords. In Section 7.2, we discuss ethical issues of using leaked data.

Training data included 40 million passwords from the OpenWall Mangled Wordlist,² 32 million leaked passwords from the website RockYou [36], and about 47,000 passwords leaked from MySpace [27]. We augmented the training data with all strings harvested from the Google Web Corpus,³ resulting in a dictionary of 14 million alphabetic strings.

In the *weak attacker* scenario, we consider an attacker with limited computational resources who can make 500 million (5×10^8) guesses. In the *medium attacker* scenario, we consider an attacker with greater resources who can make 50 billion (5×10^{10}) guesses. Finally, in the *strong attacker* scenario, we examine what percentage of passwords would have been guessed within the first 5 trillion (5×10^{12}) guesses. John the Ripper⁴, a popular password cracker, can crack 500 million hashed passwords in about an hour on a modern desktop machine. Five trillion guesses would require a botnet of several hundred machines working for several days.

5 Results

From January to April 2012, 2,931 people completed the initial task, and 2,016 of these subjects returned for the second part of the study. We begin our evaluation by comparing characteristics of passwords created in each condition, including their length and the character classes used. Next, we simulate a cracking algorithm to evaluate what proportion of passwords in each condition would be cracked by adversaries of varying strength. We

²<http://www.openwall.com/wordlists/>

³<http://googleresearch.blogspot.com/2006/08/all-our-n-gram-are-belong-to-you.html>

⁴<http://www.openwall.com/john/>

then examine the usability of these passwords, followed by data about the process of password creation. Finally, we discuss participant demographics and potential interaction effects. In Section 6, we provide additional results on participants' attitudes and reactions.

5.1 Password Characteristics

The presence of almost any password meter significantly increased password length. In conditions that scored passwords stringently, the meter also increased the use of digits, uppercase letters, and symbols. The length of the passwords varied significantly across conditions, as did the number of digits, uppercase characters, and symbols contained in each password (HC K-W, $p < .001$). Table 1 displays the characteristics of passwords created.

Length The presence of any password meter except *text-only* resulted in significantly longer passwords. Passwords created with *no meter* had a mean length of 10.4, and passwords created in the *text-only* condition had a mean length of 10.9, which was not significantly different. Passwords created in the thirteen other conditions with meters, with mean length ranging from 11.3 to 14.9 characters, were significantly longer than in *no meter* (HC MWU, $p \leq .014$).

Furthermore, passwords created in *half-score*, with mean length 14.9, and in *nudge-16*, with mean length 13.0, were significantly longer than those created in *baseline meter*, which had mean length 12.0 (HC MWU, $p \leq .017$). On the other hand, passwords created in *text-only*, with mean length 10.9, were significantly shorter than in *baseline meter* (HC MWU, $p = .015$). Although passwords created in *one-third-score* had mean length 14.3, they had a high standard deviation (8.1) and did not differ significantly from *baseline meter*.

Digits, Uppercase Characters, and Symbols Compared to *no meter*, passwords in five conditions contained significantly more digits: *half-score*, *one-third-score*, *nudge-comp8*, *bold text-only half-score*, and *bunny* (HC MWU, $p < .028$). In each of these five conditions, passwords contained a mean of 3.2 to 3.4 digits, compared to 2.4 digits in *no meter*. The mean number of digits in all other conditions ranged from 2.5 to 3.1.

In three of these conditions, *half-score*, *one-third-score*, and *bold text-only half-score*, passwords on average contained both more uppercase letters and more symbols (HC MWU, $p < .019$) than in *no meter*. In these three conditions, the mean number of uppercase characters ranged from 1.4 to 1.5 and the mean number of symbols ranged from 0.8 to 1.0, whereas passwords created in *no meter* contained a mean of 0.8 uppercase characters and 0.3 symbols. Furthermore, passwords created in

Table 1: A comparison across conditions of the characteristics of passwords created: the *length*, number of *digits*, number of *uppercase* letters, and number of *symbols*. For each metric, we present the mean, the standard deviation (SD), and the median. Conditions that differ significantly from *no meter* are indicated with an asterisk (*). Conditions that differ significantly from *baseline meter* are indicated with a dagger (†).

Metric	no meter (*)	baseline meter (†)	three-segment	green	tiny	huge	no suggestion	text-only	half-score	one-third-score	nudge-16	nudge-comp8	text-only half	bold text-only half	bunny
Length		*	*	*	*	*	*	†	*,†	*	*,†	*	*	*	*
Mean	10.4	12.0	11.5	11.3	11.4	11.6	11.4	10.9	14.9	14.3	13.0	11.6	12.3	13.0	11.2
SD	2.9	3.7	3.8	3.6	3.2	3.3	3.5	3.2	7.3	8.1	3.7	3.5	6.1	5.5	3.1
Median	9	11	10	10	11	11	11	10	12.5	12	12	11	10.5	11	10
Digits									*	*		*		*	*
Mean	2.4	2.7	2.8	2.6	2.7	2.5	3.0	2.5	3.3	3.4	3.2	3.3	3.1	3.2	3.3
SD	2.8	2.6	2.6	2.5	2.3	2.2	2.8	2.3	3.0	3.2	3.4	2.8	3.5	3.0	3.0
Median	2	2	2	2	3	2	2	2	3	3	3	3	2	3	3
Uppercase									*	*				*,†	
Mean	0.8	0.8	0.9	0.8	0.6	1.0	0.7	0.9	1.5	1.4	0.5	0.8	1.2	1.5	0.8
SD	2.0	1.8	1.7	2.0	1.4	2.3	1.5	1.7	3.4	3.2	1.3	1.5	2.2	2.5	1.5
Median	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.5
Symbols									*	*			*	*	
Mean	0.3	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.8	1.0	0.5	0.5	0.6	0.9	0.4
SD	0.7	1.0	0.8	1.1	0.7	0.8	0.8	0.7	1.6	2.7	1.3	1.0	1.2	1.7	0.7
Median	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

text-only half-score had significantly more symbols, 0.6 on average, than *no meter*, although the mean number of digits did not differ significantly.

While most participants used digits in their passwords, uppercase characters and symbols were not as common. In nearly all conditions, the majority of participants did not use any uppercase characters in their password despite the meter’s prompts to do so. In addition, fewer than half of participants in any condition used symbols.

5.2 Password Guessability

We evaluated the strength of passwords based on their “guessability,” which is the number of guesses an adversary would need to guess that password, as detailed in Section 2.4. We considered three adversaries: a *weak attacker* with limited resources who makes 500 million (5×10^8) guesses, a *medium attacker* who makes 50 billion (5×10^{10}) guesses, and a *strong attacker* who makes 5 trillion (5×10^{12}) guesses. Table 2 and Figure 3 present the proportion of passwords cracked by condition.

We found that all conditions with password meters appeared to provide a small advantage against attackers of all three strengths. In all fourteen conditions with meters, the percentage of passwords cracked by all three adversaries was always smaller than in *no meter*, although most of these differences were not statistically

significant. The only substantial increases in resistance to cracking were provided by the two stringent meters with visual bars, *half-score* and *one-third-score*.

A *weak adversary* cracked 21.0% of passwords in the *no meter* condition, which was significantly larger than the 5.8% of passwords cracked in the *half-score* condition and the 4.7% of passwords cracked in *one-third-score* (HC χ^2 , $p < 0.001$). Furthermore, only 7.8% of passwords were cracked in *bunny*, which was also significantly less than in *no meter* (HC χ^2 , $p = 0.008$). Between 9.5% and 15.3% of passwords were cracked in all other conditions with meters, none of which were statistically significantly different than *no meter*.

In the *medium adversary* scenario, significantly more passwords were cracked in the *no meter* condition than in the *half-score* and *one-third-score* conditions (HC χ^2 , $p \leq 0.017$). 35.4% of the passwords in the *no meter* condition were cracked, compared with 19.5% of passwords in *half-score* and 16.8% of passwords in *one-third-score*. None of the other conditions differed significantly from *no meter*; between 23.7% and 34.4% of passwords were cracked in these conditions.

The *half-score* and *one-third-score* meters were again significantly better than *no meter* against a *strong adversary*. In *no meter*, 46.7% of passwords were cracked, compared with 26.3% in *half-score* and 27.9% in *one-third-score* (HC χ^2 , $p \leq 0.005$). Between 33.7% and

46.2% of passwords in all other conditions were cracked.

After the completion of the experiment, we ran additional conditions to explore how meters consisting of only a visual bar, without accompanying text, would compare to text-only conditions and conditions containing both text and visual features. Since this data was collected two months after the rest of our data, we do not include it in our main analyses. However, passwords created in these conditions performed similarly to equivalent text-only conditions and strictly worse than equivalent conditions containing both a bar and text. For instance, a strong adversary cracked 48.3% of passwords created with the *baseline meter* bar without its accompanying text and 33.0% of passwords created with the *half-score* bar without its accompanying text.

5.3 Password Memorability and Storage

To gauge the memorability of the passwords subjects created, we considered the proportion of subjects who returned for the second day of our study, the ability of participants to enter their password both minutes after creation and a few days after creation, and the number of participants who either reported or were observed storing or writing down their password.

2,016 of our participants, 68.8%, returned and completed the second part of the study. The proportion of participants who returned did not differ significantly across conditions (χ^2 , $p=0.241$).

Between the 68.8% of participants who returned for the second part of the study and the 31.2% of participants who did not, there were no significant differences in the length of the passwords created, the number of digits their password contained, or the percentage of passwords cracked by a *medium* or *strong* attacker. However, the *weak* attacker cracked a significantly higher percentage of passwords created by subjects who did not return for the second part of the study than passwords created by participants who did return (HC χ^2 , $p<.001$). 14.5% of passwords created by subjects who did not return and 9.5% of passwords created by subjects who did return were cracked. Participants who returned for the second part of the study also had more uppercase letters and more symbols in their passwords (K-W, $p<.001$). Participants who returned had a mean of 1.0 uppercase letters and 0.6 symbols in their passwords, while those who did not had a mean of 0.8 uppercase letters and 0.5 symbols.

Participants' ability to recall their password also did not differ significantly between conditions, either minutes after creating their password (χ^2 , $p=0.236$) or at least two days later (χ^2 , $p=0.250$). In each condition, 93% or more of participants were able to enter their password correctly within three attempts minutes after creating the password. When they received an email two days

later to return and log in with their password, between 77% and 89% of the subjects in each condition were able to log in successfully within the first three attempts.

As an additional test of password memorability, we asked participants if they had written their password down, either electronically or on paper, or if they had stored their password in their browser. Furthermore, we captured keystroke data as they entered their password, which we examined for evidence of pasting in the password. If a participant answered affirmatively to either question or pasted the password into the password field, he or she was considered as having stored the password. Overall, 767 participants (38.0% of those who returned) reported that they had stored or written down their password. 78 of these 767 participants were also observed to have pasted in their password. An additional 32 participants (1.6%) were observed pasting in their password even though they had said they had not stored it.

The proportion of participants storing their passwords did not differ across conditions (χ^2 , $p=0.364$). In each condition, between 33% and 44% of participants were observed pasting in a password or reported writing down or storing their password.

5.4 Password Creation Process

Based on analysis of participants' keystrokes during password creation, we found that participants behaved differently in the presence of different password meters. Password meters seemed to encourage participants to reach milestones, such as filling the meter or no longer having a "bad" or "poor" password. The majority of participants who saw the most stringent meters changed their mind partway into password creation, erasing what they had typed and creating a different password. Table 3 presents this numerical data about password creation.

Most participants created a new password for this study, although some participants reused or modified an existing password. Between 57% and 71% of subjects in each condition (63% overall) reported creating an entirely new password, between 15% and 26% (21% overall) reported modifying an existing password, between 9% and 19% (14% overall) reported reusing an existing password, and fewer than 4% (2% overall) used some other strategy. The proportion of participants reporting each behavior did not vary significantly across conditions (χ^2 , $p=.876$).

Participants in *nudge-16*, *bunny*, and all four stringent conditions took longer to create their password than those in *no meter* (HC χ^2 , $p<.001$). The mean password creation time, measured from the first to the last keystroke in the password box, was 19.9 seconds in the *no meter* condition. It was 60.8 seconds for *half-score*, 59.8 seconds for *one-third-score*, 57.1 seconds

Table 2: A comparison of the percentage of passwords in each condition cracked by weak (5×10^8 guesses), medium (5×10^{10} guesses), and strong adversaries (5×10^{12} guesses). Each cell contains the percentage of passwords cracked in that threat model. Conditions that differ significantly from *no meter* are indicated with an asterisk (*).

Adversary	no meter (*)	baseline meter (†)	three-segment	green	tiny	huge	no suggestion	text-only	half-score	one-third-score	nudge-16	nudge-comp8	text-only half	bold text-only half	bunny
Weak									*	*					*
% Cracked	21.0	11.1	10.3	12.0	10.7	9.6	11.0	15.1	5.8	4.7	15.3	10.3	9.5	11.4	7.8
Medium									*	*					
% Cracked	35.4	27.2	26.6	30.0	30.0	31.0	25.9	34.4	19.5	16.8	25.0	23.7	24.2	25.7	28.1
Strong									*	*					
% Cracked	46.7	39.4	39.4	45.5	42.1	41.6	39.3	46.2	26.3	27.9	33.7	39.2	34.7	35.6	40.1

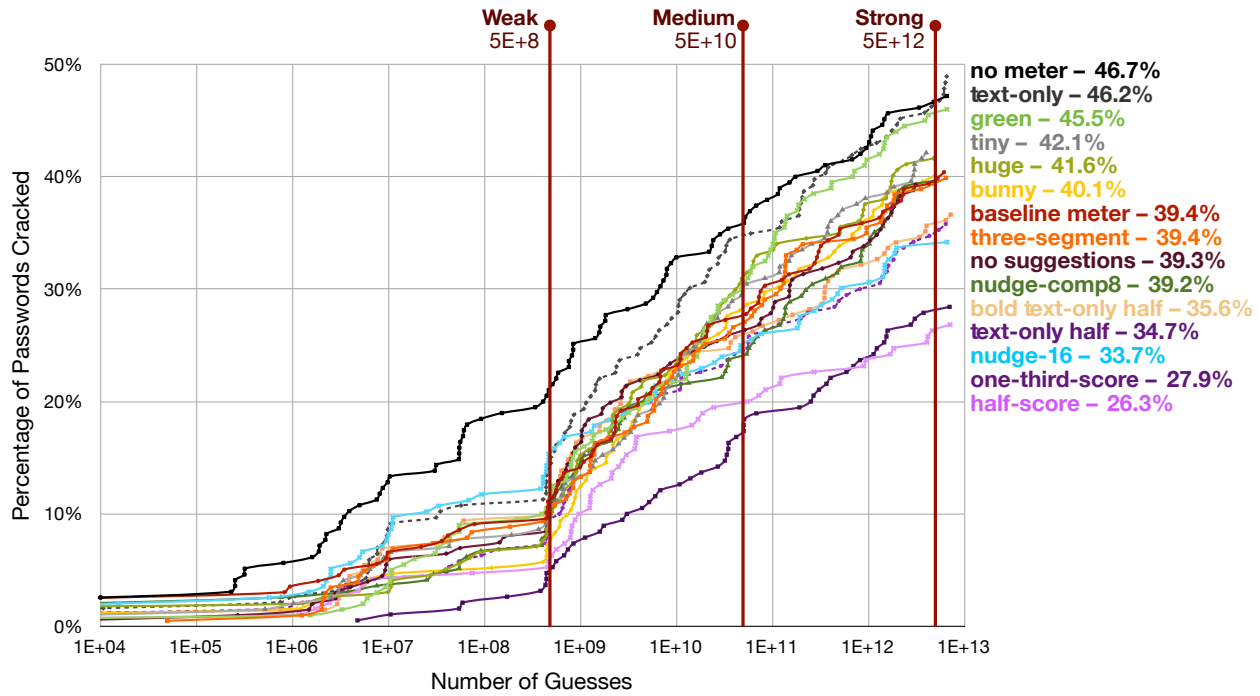


Figure 3: This graph contrasts the percentage of passwords that were cracked in each condition. The x-axis, which is logarithmically scaled, indicates the number of guesses made by an adversary, as described in Section 2.4. The y-axis indicates the percentage of passwords in that condition cracked by that particular guess number.

for *bold text-only half-score*, 38.5 seconds for *text-only half-score*, 33.1 seconds for *nudge-16*, and 30.4 seconds for *bunny*. Compared also to the *baseline meter* meter, where mean password creation time was 23.5 seconds, participants took significantly longer in the *half-score*, *one-third-score*, and *bold text-only half-score* conditions (HC χ^2 , $p < .008$). The mean time of password creation ranged from 21.0 to 26.6 seconds in all other conditions.

Password meters encouraged participants both to avoid passwords that the meter rated “bad” or “poor” and

to create passwords that filled the meter. Had there been a password meter, 24.1% of passwords created in *no meter* would have scored “bad” or “poor,” which was significantly higher than the 12.0% or fewer of passwords in all non-stringent conditions other than *no suggestions* and *nudge-16* rated “bad” or “poor” (HC χ^2 , $p \leq 0.035$). Had *no meter* contained a password meter, 25.1% of passwords created would have filled the meter. A larger proportion of passwords in all non-stringent conditions other than *no suggestions* and *nudge-16* filled the meter (HC

Table 3: A comparison across conditions of password creation: the percentage of participants who completely *filled the password meter* or equivalently scored “excellent” in text-only conditions, the percentage of participants whose password received a score of “*bad*” or “*poor*”, the *time* of password creation (first to last keystroke), the number of *deletions* (characters deleted after being entered) in the password creation process, the percentage of participants who *changed their password* (initially entering a valid password containing at least 8 characters before completely deleting it and entering a different password), and the *edit distance* between the initial password entered and the final password saved, normalized by the length of the final password. Conditions differing significantly from *no meter* are indicated with an asterisk (*), while those differing significantly from *baseline meter* are marked with a dagger (†).

Metric	no meter (*)	baseline meter (†)	three-segment	green	tiny	huge	no suggestion	text-only	half-score	one-third-score	nudge-16	nudge-comp8	text-only half	bold text-only half	bunny
Filled Meter		*	*	*	*	*	*	*	*,†	*,†	†	*	*,†	*,†	*
% of participants	(25.1)	48.5	53.2	42.5	48.2	52.8	37.3	46.2	9.0	1.6	24.5	46.9	3.2	5.0	48.4
“Bad” or “Poor”		*	*	*	*	*	*	*	*,†	*,†	†	*	*,†	*,†	*
% of participants	(24.1)	9.1	10.3	12.0	9.6	8.1	7.5	13.4	58.4	93.7	37.2	9.8	76.3	67.8	8.3
Time (seconds)									*,†	*,†	*		*	*,†	*
Mean	19.9	23.5	22.7	21.0	21.5	25.8	24.7	24.8	60.8	59.8	33.1	26.6	38.5	57.1	30.4
SD	28.4	22.7	23.6	22.2	23.2	28.9	36.6	29.4	75.7	84.9	33.2	30.2	49.8	150.0	36.9
Median	10.6	15.6	14.0	13.7	13.1	14.7	13.0	14.0	39.1	34.2	23.2	13.8	23.5	32.8	19.8
Deletions									*,†	*,†	*,†		*,†	*,†	*
Mean	5.3	6.2	7.5	5.8	6.2	7.8	5.5	7.8	23.8	22.9	12.1	8.1	14.6	23.1	10.7
SD	10.7	10.2	13.7	12.4	10.8	11.3	8.4	11.9	29.0	26.6	16.2	13.3	19.3	26.9	17.2
Median	0	0	0	0	1	2	0	0	13.5	13	8	1	8	13.5	5
Changed PW									*,†	*,†	*,†		*,†	*,†	*,†
% of participants	14.4	18.7	25.6	16.5	23.9	23.4	25.9	25.8	52.6	52.6	40.3	24.7	35.8	51.0	34.9
Norm. Edit Dist.									*,†	*,†	*,†		*,†	*,†	*,†
Mean	0.10	0.09	0.47	0.09	0.14	0.12	0.15	0.17	0.37	0.45	0.27	0.15	0.27	0.35	0.28
SD	0.29	0.23	4.84	0.28	0.30	0.31	0.37	0.36	0.42	1.22	0.38	0.36	0.43	0.47	0.70
Median	0	0	0	0	0	0	0	0	0.15	0.11	0	0	0	0.08	0

χ^2 , $p \leq 0.006$). In each of these conditions, 42.5% or more of the passwords filled the meter. While the proportion of passwords in *nudge-16* and the four stringent conditions reaching these thresholds was significantly lower than *baseline meter*, the proportions would have been higher than *baseline meter* were the *baseline meter* scoring algorithm used in those conditions.

During the password creation process, participants in all four stringent conditions, as well as in *nudge-16*, made more changes to their password than in *no meter* or *baseline meter*. We considered the number of deletions a participant made, which we defined as the number of characters that were inserted into the password and then later deleted. In the four stringent conditions and in *nudge-16*, the mean number of deletions by each participant ranged from 12.1 to 23.8 characters. In contrast, significantly fewer deletions were made in *no meter*, with a mean of 5.3 deletions, and *baseline meter*, with a mean of 6.2 deletions (HC MWU, $p < 0.001$). The *bunny* condition, with a mean of 10.7, also had significantly more deletions than *no meter* (HC MWU, $p = 0.004$).

We further analyzed the proportion of participants who changed their password, finding significantly more changes occurring in the stringent conditions, as well as in *nudge-16* and *bunny*. Some participants entered a password containing eight or more characters, meeting the stated requirements, and then completely erased the password creation box to start over. We define the *initial password* to be the longest such password containing eight or more characters that a participant created before starting over. Similarly, we define the *final password* to be the password the participant eventually saved. We considered participants to have changed their password if they created an initial password, completely erased the password field, and saved a final password that differed by one edit or more from their initial password.

More than half of the participants in *half-score*, *one-third-score*, and *bold text-only half-score* changed their password during creation. Similarly, between 34.9% and 40.3% of *nudge-16*, *text-only half-score*, and *bunny* participants changed their password. The proportion of participants in these six conditions who changed their pass-

word was greater than the 14.4% of *no meter* participants and 18.7% of *baseline meter* participants who did so (HC χ^2 , $p \leq .010$). Across all conditions, only 7.7% of final passwords consisted of the initial password with additional characters added to the end; in a particular condition, this percentage never exceeded 16%.

These changes in the password participants were creating resulted in final passwords that differed considerably from the initial password. We assigned an edit distance of 0 to all participants who did not change their password. For all other participants, we computed the Levenshtein distance between the initial and final password, normalized by the length of the final password. The mean normalized edit distance between initial and final passwords ranged from 0.27 to 0.45 in the six aforementioned conditions, significantly greater than *no meter*, with a mean of 0.10, and *baseline meter*, with a mean of 0.09 (HC MWU, $p < .003$).

We also compared the guessability of the initial and final passwords for participants whose initial password, final password, or both were guessed by the strong adversary. 86.1% of the 43 such changes in *half-score* resulted in a password that would take longer to guess, as did 83.8% of 37 such changes in *text-only half-score*. In contrast, 50% of 18 such changes in *baseline meter* and between 56.7% and 76.7% such changes in all other conditions resulted in passwords that would take longer to guess. However, these differences were not statistically significant.

5.5 Participant Demographics

Participants ranged in age from 18 to 74 years old, and 63% percent reported being male and 37% female.⁵ 40% percent reported majoring in or having a degree or job in computer science, computer engineering, information technology, or a related field; 55% said they did not. Participants lived in 96 different countries, with most from India (42%) and the United States (32%). Because many of our password meters used a color scheme that includes red and green, we asked about color-blindness; 3% of participants reported being red-green color-blind, while 92% said they were not, consistent with the general population [30].

The number of subjects in each condition ranged from 184 to 202, since conditions were not reassigned if a participant did not complete the study. There were no statistically significant differences in the distribution of participants' gender, age, technology background, or country of residence across experimental conditions.

However, participants who lived in different countries created different types of passwords. We separated par-

⁵We offered the option not to answer demographic questions; when percentages sum to less than 100, non-answers make up the remainder.

ticipants into three groups based on location: United States, India, and "the rest of the world." Indian subjects' passwords had mean length 12.2, U.S. subjects' passwords had mean length 11.9, and all other subjects' passwords had mean length 12.1 (HC K-W, $p = 0.002$). Furthermore, Indian subjects' passwords had a mean of 0.9 uppercase letters, and both U.S. subjects' and all other subjects' passwords had a mean of 1.0 uppercase letters (HC K-W, $p < 0.001$). While the percentage of passwords cracked by a *weak* or *medium* attacker did not differ significantly between the three groups, a lower percentage of the passwords created by Indian participants than those created by American participants was cracked by a *strong* adversary (HC χ^2 , $p = .032$). 42.3% of passwords created by subjects from the U.S., 35.5% of passwords created by subjects from India, and 38.8% of passwords created by subjects from neither country were cracked by a *strong* adversary. However, the guessing algorithm was trained on sets of leaked passwords from sites based in the U.S., which may have biased its guesses.

6 Participants' Attitudes and Perceptions

We asked participants to rate their agreement on a Likert scale with fourteen statements about the password creation process, such as whether it was fun or annoying, as well as their beliefs about the password meter they saw. We also asked participants to respond to an open-ended prompt about how the password meter did or did not help. We begin by reporting participants' survey responses, which reveal annoyance among participants in the stringent conditions. The *one-third-score* condition and *text-only stringent* conditions also led participants to believe the meter gave an incorrect score and to place less importance on the meter's rating. The distribution of responses to select survey questions is shown in Figure 4. We then present participants' open-ended responses, which illuminate strategies for receiving high scores from the meter.

6.1 Attitudes Toward Password Meters

In a survey immediately following password creation, a higher percentage of participants in the stringent conditions found password creation to be annoying or difficult than those in *baseline meter*. A larger proportion of subjects in the four stringent conditions than in either the *no meter* or *baseline meter* conditions agreed that creating a password in this study was annoying (HC χ^2 , $p \leq .022$). Similarly, a higher percentage of subjects in the *half-score* and *bold text-only half-score* found creating a password difficult than in either the *no meter* or *baseline meter* conditions (HC χ^2 , $p \leq .012$). Creating a password was also considered difficult by a higher percentage of

subjects in *one-third-score* and *text-only half-score* than in *baseline meter* (HC χ^2 , $p \leq .003$), although these conditions did not differ significantly from *no meter*.

Participants in the stringent conditions also found the password meter itself to be annoying at a higher rate. A higher percentage of subjects in all four stringent conditions than in *baseline meter* agreed that the password-strength meter was annoying (HC χ^2 , $p \leq .007$). Between 27% and 40% of participants in the four stringent conditions, compared with 13% of *baseline meter* participants, found the meter annoying.

Participants in the two stringent conditions without a visual bar felt that they did not understand how the meter rated their password. 38% of *text-only half-score* and 39% of *bold text-only half-score* participants agreed with the statement, “I do not understand how the password strength meter rates my password,” which was significantly greater than the 22% of participants in *baseline meter* who felt similarly (HC χ^2 , $p \leq .015$). 32% of *half-score* participants and 34% of *one-third-score* participants also agreed, although these conditions were not statistically significantly different than *baseline meter*.

The *one-third-score* condition and both text-only stringent conditions led participants to place less importance on the meter. A smaller proportion of *one-third-score*, *text-only half-score*, and *bold text-only half-score* participants than *baseline meter* subjects agreed, “It’s important to me that the password-strength meter gives my password a high score” (HC χ^2 , $p \leq .021$). 72% of *baseline meter* participants, yet only between 49% and 56% of participants in those three conditions, agreed. In all other conditions, between 64% and 78% of participants agreed. Among these conditions was *half-score*, in which 68% of participants agreed, significantly more than in *one-third-score* (HC χ^2 , $p = .005$).

More participants in those same three conditions felt the meter’s score was incorrect. 42-47% of *one-third-score*, *text-only half-score*, and *bold text-only half-score* participants felt the meter gave their password an incorrect score, significantly more than the 21% of *baseline meter* participants who felt similarly (HC χ^2 , $p \leq .001$). Between 12% and 33% of participants in all other conditions, including *half-score*, agreed; these conditions did not differ significantly from *baseline meter*.

6.2 Participant Motivations

Participants’ open-ended responses to the prompt, “Please explain how the password strength meter helped you create a better password, or explain why it was not helpful,” allowed some participants to explain their thought process in reaction to the meter, while others discussed their impressions of what makes a good password.

6.2.1 Reactions to the Password Meter

Some participants noted that they changed their behavior in response to the meter, most commonly adding a different character class to the end of the password. One participant said the meter “motivated [him] to use symbols,” while another “just started adding numbers and letters to the end of it until the high score was reached.” Participants also said that the meter encouraged or reminded them to use a more secure password. One representative participant explained, “It kept me from being lazy when creating my password. [I] probably would not have capitalized any letters if not for the meter.”

Other participants chose a password before seeing the meter, yet expressed comfort in receiving validation. For instance, one representative participant noted, “The password I ultimately used was decided on before hand. However, whilst I was typing and I saw the strength of my password increase and in turn felt reassured.”

However, a substantial minority of participants explained that they ignore password meters, often because they believe these meters discourage passwords they can remember. One representative participant said, “No matter what the meter says, I will just use the password I chose because it’s the password I can remember. I do not want to get a high score for the meter and in the end have to lose or change my password.” Some participants expressed frustration with meters for not understanding this behavior. For instance, one participant explained, “I have certain passwords that I use because I can remember them easily. I hate when the meter says my password is not good enough— it’s good enough for me!”

Participants also reported embarrassment at poor scores, fear of the consequences of having a weak password, or simply a desire to succeed at all tasks. One participant who exemplifies the final approach said, “I wanted to make my password better than just ‘fair,’ so I began to add more numbers until the password-strength meter displayed that my password was ‘good.’ I wanted to create a strong password because I’m a highly competitive perfectionist who enjoys positive feedback.” In contrast, another participant stated, “Seeing a password strength meter telling me my password is weak is scary.”

6.2.2 Impressions of Password Strength

Participants noted impressions of password strength that were often based on past experiences. However, the stringent conditions seemed to violate their expectations.

Most commonly, subjects identified a password containing different character classes as strong. One representative participant said, “I am pretty familiar with password strength meters, so I knew that creating a password with at least 1 number/symbol and a mixture of upper

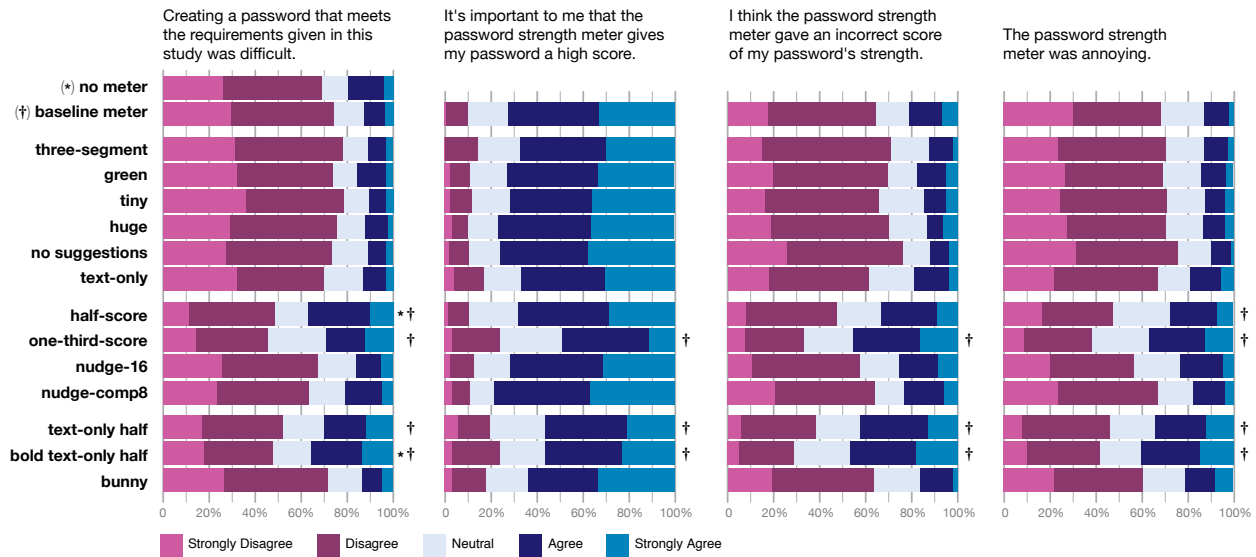


Figure 4: These charts depict participants’ agreement or disagreement with the statement above each chart. Each color represents the proportion of participants in that condition who expressed a particular level of agreement or disagreement with the statement. Conditions in which the proportion of participants agreeing with a statement differed significantly from *no meter* are indicated with an asterisk (*), while those that differed significantly from *baseline meter* are marked with a dagger (†). Participants in *no meter* did not respond to questions about password meters.

and lower case letters would be considered strong.” Participants also had expectations for the detailed algorithm with which passwords were scored, as exemplified by a participant who thought the meter “includes only English words as predictable; I could have used the Croatian for ‘password123’ if I wanted.”

The stringent conditions elicited complaints from participants who disagreed with the meter. For example, one participant was unsure how to receive a good score, saying, “No matter what I typed, i.e. how long or what characters, it still told me it was poor or fair.” Another participant lamented, “Nothing was good enough for it!” Some participants questioned the veracity of the stringent meters. For instance, a *one-third-score* participant said, “I have numbers, upper/lower case, and several symbols. It’s 13 characters long. It still said it was poor. No way that it’s poor.” Other participants reused passwords that had received high scores from meters in the wild, noting surprise at the stringent meters’ low scores. Some participants became frustrated, including one who said the *one-third-score* meter “was extremely annoying and made me want to punch my computer.”

The *bunny* received mixed feedback from participants. Some respondents thought that it sufficed as a feedback mechanism for passwords. For instance, one subject said, “I think it was just as helpful as any other method I have seen for judging a password’s strength...I do think the dancing bunny is much more light-hearted and fun.” However, other participants found the more traditional

bar to be more appropriate, including one who said *bunny* “was annoying, I am not five [years old].”

6.2.3 Goals for the Password Meter

Participants stated two primary goals they adopted while using the password meter. Some participants aimed to fill the bar, while others hoped simply to reach a point the meter considered not to be poor. Those participants who aimed to fill the bar noted that they continued to modify their password until the bar was full, citing as motivation the validation of having completed their goal or their belief that a full bar indicated high security.

Participants employing the latter strategy increased the complexity of their password until the text “poor” disappeared. One participant noted, “It gave me a fair score, so I went ahead with the password, but if it would have given me a low score I would not have used this password.” A number of participants noted that they didn’t want to receive a poor rating. One representative participant said, “I didn’t want to have poor strength, while I didn’t feel I needed something crazy.”

Some participants also identified the bar’s color as a factor in determining when a password was good enough. Some participants hoped to reach a green color, while others simply wanted the display not to be red. One participant aiming towards a green color said, “I already chose a fairly long password, but I changed a letter in it to an uppercase one to make it turn green.” Another

participant expressed, “I knew that I didn’t want to be in the red, but being in the yellow I thought was ok.”

7 Discussion

We discuss our major findings relating to the design of effective password meters. We also address our study’s ethical considerations, limitations, and future work.

7.1 Effective Password Meters

At a high level, we found that users do change their behavior in the presence of a password-strength meter. Seeing a password meter, even one consisting of a dancing bunny, led users to create passwords that were longer. Although the differences were generally not statistically significant, passwords created in all 14 conditions with password meters were cracked at a lower rate by adversarial models of different strengths.

However, the most substantial changes in user behavior were elicited by stringent meters. These meters led users to add additional character classes and make their password longer, leading to significantly increased resistance to a guessing attack. Furthermore, more users who saw stringent meters changed the password they were creating, erasing a valid password they had typed and replacing it with one that was usually harder to crack.

Unfortunately, the scoring systems of meters we observed in the wild were most similar to our non-stringent meters. This result suggests that meters currently in use on popular websites are not aggressive enough in encouraging users to create strong passwords. However, if all meters a user encountered were stringent, he or she might habituate to receiving low scores and ignore the meter, negating any potential security benefits.

There seems to be a limit to the stringency that a user will tolerate. In particular, the *one-third-score* meter seemed to push users too hard; *one-third-score* participants found the meter important at a lower rate and thought the meter to be incorrect at a higher rate, yet their passwords were comparable in complexity and cracking-resistance to those made by *half-score* participants. Were meters too stringent, users might just give up.

Tweaks to the password meter’s visual display did not lead to significant differences in password composition or user sentiment. Whether the meter was tiny, monochromatic, or a dancing bunny did not seem to matter. However, an important factor seemed to be the combination of text and a visual indicator, rather than only having text or only having a visual bar. Conditions containing text without visual indicators, run as part of our experiment, and conditions containing a visual bar without text, run subsequently to the experiment we focus on

here, were cracked at a higher rate and led to less favorable user sentiment than conditions containing a combination of text and a visual indicator.

In the presence of password-strength meters, participants changed the way they created a password. For instance, the majority of participants in the stringent conditions changed their password during creation. Meters seemed to encourage participants to create a password that filled the meter. If that goal seemed impossible, participants seemed content to avoid passwords that were rated “bad” or “poor.” In essence, the password meter functions as a progress meter, and participants’ behavior echoed prior results on the effects progress meters had on survey completion [8]. Meters whose estimates of password strength mirrored participants’ expectations seemed to encourage the creation of secure passwords, whereas very stringent meters whose scores diverged from expectations led to less favorable user sentiment and an increased likelihood that a participant would abandon the task of creating a strong password.

We also found many users to have beliefs regarding how to compose a strong password, such as including different character classes. Because users’ understanding of password strength appears at least partially based on experience with real-world password-strength meters and password-composition policies, our results suggest that wide-scale deployment of more stringent meters may train users to create stronger passwords routinely.

7.2 Ethical Considerations

We calculated our guessability results by training a guess-number calculator on sets of passwords that are publicly and widely available, but that were originally gathered through illegal cracking and phishing attacks. It can be argued that data acquired illegally should not be used at all by researchers, and so we want to address the ethical implications of our work. We use the passwords alone, excluding usernames and email addresses. We neither further propagate the data, nor does our work call significantly greater attention to the data sets, which have been used in several scientific studies [4, 9, 18, 38, 39]. As a result, we believe our work causes no additional harm to the victims, while offering potential benefits to researchers and system administrators.

7.3 Limitations

One potential limitation of our study is its ecological validity. Subjects created passwords for an online study, and they were not actually protecting anything valuable with those passwords. Furthermore, one of the primary motivations for part of the MTurk population is financial compensation [17], which differs from real-world moti-

variations for password creation. Outside of a study, users would create passwords on web pages with the logos and insignia of companies they might trust, perhaps making them more likely to heed a password meter's suggestions. On the other hand, subjects who realize they are participating in a password study may be more likely to think carefully about their passwords and pay closer attention to the password meter than they otherwise would. We did ask participants to imagine that they were creating passwords for their real email accounts, which prior work has shown to result in stronger passwords [21]. Because our results are based on comparing passwords between conditions, we believe our findings about how meters compare to one another can be applied outside our study.

Our study used a password-cracking algorithm developed by Weir et al. [39] in a guess-number calculator implemented by Kelley et al. [18] to determine a password's guessability. We did not experiment with a wide variety of cracking algorithms since prior work [18, 38, 42] has found that this algorithm outperformed alternatives including John the Ripper. Nevertheless, the relative resistance to cracking of the passwords we collected may differ depending on the choice of cracking algorithm.

Furthermore, the data we used to train our cracking algorithm was not optimized to crack passwords of particular provenance. For instance, passwords created by participants from India were the most difficult to crack. The data with which we trained our guessing algorithm was not optimized for participants creating passwords in languages other than English, which may have led to fewer of these passwords being cracked; prior work by Kelley et al. [18] found that the training set has a substantial effect on the success of the guessing algorithm we used.

7.4 Future Work

Further research in password-strength meters may involve continued examination of the structure and composition of passwords created with meters. The presence of a meter caused changes in users' behavior, with over 50% of participants in three of the four stringent meter conditions erasing a valid 8-character password they had already entered and entering a new, different password. The strategies users employed both initially and after this shift deserve further investigation, both to suggest directions for user feedback and to uncover patterns that can improve techniques for cracking passwords.

In addition, we have certainly not exhausted the space of possible password-strength meters. Although we have found that the score conveyed to the user is a more important factor than the visual display, it is possible that either subtle or substantial variations to the scoring algorithm (e.g., representing a password's likelihood [7]) or to the textual feedback provided to users may increase

the usability and security of the resulting passwords. Furthermore, there seems to be a limit to how stringent a meter can be. Alternate scoring algorithms, improved text feedback, and the degree of stringency that leads to the best tradeoff between usability and security for passwords thus appear to be fertile ground for future work.

8 Conclusion

We have conducted the first large-scale study of password-strength meters, finding that meters did affect user behavior and security. Meters led users to create longer passwords. However, unless the meter scored passwords stringently, the resulting passwords were only marginally more resistant to password cracking attacks.

Meters that rated passwords stringently led users to make significantly longer passwords that included more digits, symbols, and uppercase letters. These passwords were not observed to be less memorable or usable, yet they were cracked at a lower rate by simulated adversaries making 500 million, 50 billion, and 5 trillion guesses. The most stringent meter annoyed users, yet did not provide security benefits beyond those provided by slightly less stringent meters. The combination of a visual indicator and text outperformed either in isolation. However, the visual indicator's appearance did not appear to have a substantial impact.

Despite the added strength that these more stringent meters convey, we observed many more lenient meters deployed in practice. Our findings suggest that, so long as they are not overly onerous, employing more rigorous meters would increase security.

9 Acknowledgments

This research was supported by NSF grants DGE-0903659 and CNS-1116776, by CyLab at Carnegie Mellon under grants DAAD19-02-1-0389 and W911NF-09-1-0273 from the Army Research Office, by Air Force Research Lab Award No. FA87501220139, and by a gift from Microsoft Research.

References

- [1] ADAMS, A., SASSE, M. A., AND LUNT, P. Making passwords secure and usable. In *Proc. HCI on People and Computers XII* (1997).
- [2] BISHOP, M., AND KLEIN, D. V. Improving system security via proactive password checking. *Computers & Security* 14, 3 (1995), 233–249.
- [3] BONNEAU, J. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In *Proc. IEEE Symposium on Security and Privacy* (2012).

- [4] BONNEAU, J., JUST, M., AND MATTHEWS, G. What's in a name? Evaluating statistical attacks on personal knowledge questions. In *Proc. Financial Crypto* (2010).
- [5] BUHRMESTER, M., KWANG, T., AND GOSLING, S. D. Amazon's Mechanical Turk: A new source of inexpensive, yet high-quality, data? *Perspectives on Psychological Science* 6, 1 (2011), 3–5.
- [6] BURR, W. E., DODSON, D. F., AND POLK, W. T. Electronic authentication guideline. Tech. rep., NIST, 2006.
- [7] CASTELLUCCIA, C., DÜRMUTH, M., AND PERITO, D. Adaptive password-strength meters from Markov models. In *Proc. NDSS* (2012).
- [8] CONRAD, F. G., COUPER, M. P., TOURANGEAU, R., AND PEYTCHEV, A. The impact of progress indicators on task completion. *Interacting with computers* 22, 5 (2010), 417–427.
- [9] DELL'AMICO, M., MICHIARDI, P., AND ROUDIER, Y. Password strength: An empirical analysis. In *Proc. INFOCOM* (2010).
- [10] DOWNS, J. S., HOLBROOK, M. B., SHENG, S., AND CRANOR, L. F. Are your participants gaming the system? Screening Mechanical Turk workers. In *Proc. CHI* (2010).
- [11] FEW, S. *Information Dashboard Design: The Effective Visual Communication of Data*. O'Reilly Media, Inc., 2006.
- [12] FLORÊNCIO, D., AND HERLEY, C. A large-scale study of web password habits. In *Proc. WWW* (2007).
- [13] FORGET, A., CHIASSON, S., VAN OORSCHOT, P., AND BIDDLE, R. Improving text passwords through persuasion. In *Proc. SOUPS* (2008).
- [14] HERLEY, C. So long, and no thanks for the externalities: The rational rejection of security advice by users. In *Proc. NSPW* (2009).
- [15] HERLEY, C., AND VAN OORSCHOT, P. A research agenda acknowledging the persistence of passwords. *IEEE Security & Privacy*, 99 (2011).
- [16] INGLESANT, P., AND SASSE, M. A. The true cost of unusable password policies: Password use in the wild. In *Proc. CHI* (2010).
- [17] IPEIROTIS, P. G. Demographics of Mechanical Turk. Tech. Rep. CeDER-10-01, New York University, March 2010.
- [18] KELLEY, P. G., KOMANDURI, S., MAZUREK, M. L., SHAY, R., VIDAS, T., BAUER, L., CHRISTIN, N., CRANOR, L. F., AND LOPEZ, J. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *Proc. IEEE Symposium on Security and Privacy* (2012).
- [19] KESSLER, D. A., MANDE, J. R., SCARBROUGH, F. E., SCHAPIRO, R., AND FEIDEN, K. Developing the “nutrition facts” food label. *Harvard Health Policy Review* 4, 2 (2003), 13–24.
- [20] KITTUR, A., CHI, E. H., AND SUH, B. Crowdsourcing user studies with Mechanical Turk. In *Proc. CHI* (2008).
- [21] KOMANDURI, S., SHAY, R., KELLEY, P. G., MAZUREK, M. L., BAUER, L., CHRISTIN, N., CRANOR, L. F., AND EGELMAN, S. Of passwords and people: Measuring the effect of password-composition policies. In *Proc. CHI* (2011).
- [22] KOTADIA, M. Gates predicts death of the password, Feb. 2004. <http://news.cnet.com/2100-1029-5164733.html>.
- [23] LEYDEN, J. Office workers give away passwords for a cheap pen, Apr. 2003. http://www.theregister.co.uk/2003/04/18/office_workers_give_away_passwords/.
- [24] LOEWENSTEIN, G. F., AND HAISLEY, E. C. The economist as therapist: Methodological ramifications of ‘light’ paternalism. In *The Foundations of Positive and Normative Economics*. Oxford University Press, 2008.
- [25] MILMAN, D. A. Death to passwords, Dec. 2010. http://blogs.computerworld.com/17543/death_to_passwords.
- [26] PROCTOR, R. W., LIEN, M.-C., VU, K.-P. L., SCHULTZ, E. E., AND SALVENDY, G. Improving computer security for authentication of users: Influence of proactive password restrictions. *Behavior Research Methods, Instruments, & Computers* 34, 2 (2002), 163–169.
- [27] SCHNEIER, B. Myspace passwords aren't so dumb, Dec. 2006. <http://www.wired.com/politics/security/commentary/securitymatters/2006/12/72300>.
- [28] SHAY, R., KELLEY, P. G., KOMANDURI, S., MAZUREK, M. L., UR, B., VIDAS, T., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. Correct horse battery staple: Exploring the usability of system-assigned passphrases. In *Proc. SOUPS* (2012).
- [29] SHAY, R., KOMANDURI, S., KELLEY, P. G., LEON, P. G., MAZUREK, M. L., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. Encountering stronger password requirements: User attitudes and behaviors. In *Proc. SOUPS* (2010).
- [30] SHEVELL, S. K., Ed. *The Science of Color*. Elsevier, 2003.
- [31] SOTIRAKOPOULOS, A., MUSLUKOV, I., BEZNOV, K., HERLEY, C., AND EGELMAN, S. Motivating users to choose better passwords through peer pressure. In *Proc. SOUPS (Poster Abstract)* (2011).
- [32] STANTON, J. M., STAM, K. R., MASTRANGELO, P., AND JOLTON, J. Analysis of end user security behaviors. *Comp. & Security* 24, 2 (2005), 124–133.
- [33] SUMMERS, W. C., AND BOSWORTH, E. Password policy: The good, the bad, and the ugly. In *Proc. WISICT* (2004).
- [34] THALER, R., AND SUNSTEIN, C. *Nudge: Improving decisions about health, wealth, and happiness*. Yale University Press, 2008.
- [35] TOOMIM, M., KRIPLEAN, T., PÖRTNER, C., AND LANDAY, J. Utility of human-computer interactions: Toward a science of preference measurement. In *Proc. CHI* (2011).
- [36] VANCE, A. If your password is 123456, just make it hackme. *New York Times* (New York edition), Jan. 21, 2010.
- [37] VU, K.-P. L., PROCTOR, R. W., BHARGAV-SPANTZEL, A., TAI, B.-L. B., AND COOK, J. Improving password security and memorability to protect personal and organizational information. *Int. J. of Human-Comp. Studies* 65, 8 (2007), 744–757.
- [38] WEIR, M., AGGARWAL, S., COLLINS, M., AND STERN, H. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *Proc. CCS* (2010).
- [39] WEIR, M., AGGARWAL, S., DE MEDEIROS, B., AND GLODEK, B. Password cracking using probabilistic context-free grammars. In *Proc. IEEE Symposium on Security and Privacy* (2009).
- [40] WOGALTER, M., AND VIGILANTE, JR., W. Effects of label format on knowledge acquisition and perceived readability by younger and older adults. *Ergonomics* 46, 4 (2003), 327–344.
- [41] YAN, J. J. A note on proactive password checking. In *Proc. NSPW* (2001).
- [42] ZHANG, Y., MONROSE, F., AND REITER, M. K. The security of modern password expiration: An algorithmic framework and empirical analysis. In *Proc. CCS* (2010).

I Forgot Your Password: Randomness Attacks Against PHP Applications*

George Argyros
*Dept. of Informatics & Telecom.,
University of Athens,*
argyros.george@gmail.com

Aggelos Kiayias
*Dept. of Informatics & Telecom.,
University of Athens,*
aggelos@di.uoa.gr
& *Computer Science and Engineering,
University of Connecticut, Storrs, USA.*

Abstract

We provide a number of practical techniques and algorithms for exploiting randomness vulnerabilities in PHP applications. We focus on the predictability of password reset tokens and demonstrate how an attacker can take over user accounts in a web application via predicting or algorithmically derandomizing the PHP core randomness generators. While our techniques are designed for the PHP language, the principles behind our techniques and our algorithms are independent of PHP and can readily apply to any system that utilizes weak randomness generators or low entropy sources. Our results include: algorithms that reduce the entropy of time variables, identifying and exploiting vulnerabilities of the PHP system that enable the recovery or reconstruction of PRNG seeds, an experimental analysis of the Håstad-Shamir framework for breaking truncated linear variables, an optimized online Gaussian solver for large sparse linear systems, and an algorithm for recovering the state of the Mersenne twister generator from any level of truncation. We demonstrate the gravity of our attacks via a number of case studies. Specifically, we show that a number of current widely used web applications can be broken using our techniques including Mediawiki, Joomla, Gallery, osCommerce and others.

1 Introduction

Modern web applications employ a number of ways for generating randomness, a feature which is critical for their security. From session identifiers and password reset tokens, to random filenames and password salts, almost every web application is relying on the unpredictability of these values for ensuring secure operation. However, usually programmers fail to understand the importance of using cryptographically secure pseudorandom number generators (PRNG) something that opens the potential for attacks. Even worse, the same trend holds for whole programming languages;

PHP for example lacks a built-in cryptographically secure PRNG in its core and until recently, version 5.3, it totally lacked a cryptographically secure randomness generation function.

This left PHP programmers with two options: They will either implement their own PRNG from scratch or they will employ whatever functions are offered by the API in a “homebrew” and ad-hoc fashion. In addition, backwards compatibility and other issues (cf. section 2), often push the developers away even from the newly added randomness functions, making their use very limited. As we will demonstrate and heavily exploit in this work, this approach does not produce secure web applications.

Observe that using a low entropy source or a cryptographically weak PRNG to produce randomness does not necessarily imply that an attack is feasible against a system. Indeed, so far there have been a very limited number of published attacks based on the insecure usage of PRNG functions in PHP, while popular exploit databases¹ contain nearly zero exploits for such vulnerabilities (and this may partially explain the delay in the PHP community adopting secure randomness generation functions). Showing that such attacks are in fact very practical is the objective of our work.

In this paper we develop generic techniques and algorithms to exploit randomness vulnerabilities in PHP applications. We describe implementation issues that allow one to either predict or completely recover the initial seed of the PRNGs used in most web applications. We also give algorithms for recovering the internal state of the PRNGs used by the PHP system, including the Mersenne twister generator and the glibc LFSR based generator, even when their output is truncated. These algorithms could be used in order to attack hardened PHP installations even when strong seeding is employed, as it is done by the Suhosin extension for PHP and they may be of independent interest.

We also conducted an extensive audit of several popular PHP applications. We focused on the security of password reset implementations. Using our attack

*Research partly supported by ERC Project CODAMODA.

¹e.g. <http://www.exploit-db.com>

framework we were able to mount attacks that take over arbitrary user accounts with practical complexity. A number of widely used PHP applications are affected (see Figure 7), while we believe that the impact is even larger in less known applications.

Our results suggest that randomness attacks should be considered practical for PHP applications and existing systems should be audited for these vulnerabilities. Weak randomness is a grave vulnerability in any secure system as it was also recently demonstrated in the widely publicized discovery of common primes in RSA public-keys by Lenstra et al. [14]. We finally stress that our techniques apply in any setting beyond PHP, whenever the same PRNG functions are used and the attack vector relies on predicting a system defined random object.

This is only an extended abstract, a full version can be found in [1].

1.1 Attack model

In Figure 1 we present our general attack template. An attacker is trying to predict the password reset token in order to gain another user's privileges (say an administrator's). Each time the attacker makes a request to the web server, his request is handled by a web application instance, usually represented by a specific operating system process, which contains some process specific state. The web application uses a number of application objects with values depending on its internal state, with some of these objects leaking to the attacker through the web server responses. Examples of such objects are session identifiers and outputs of PRNG functions. Although our focus is in password reset functions, the principles that we use and the techniques that we develop can be readily applied in other contexts when the application relies on the generation of random values for security applications. Examples of such applications are CAPTCHA's and the production of random filenames.

Attack complexity. Since we present explicit practical attacks, we define next the complexity under which an attack should be consider practical. There are two measure of complexity of interest. The first is the time complexity and the second is the query or communication complexity. For some of our attacks the main computational operation is the calculation of an MD5 hash. With current GPU technologies an attacker can perform up to 2^{30} MD5 calculations per second with a \$250 GPU, while with an additional \$500 can reach up to 2^{32} calculations [9]. These figures suggest that attacks that require up to 2^{40} MD5 calculations can be easilty mounted. In terms of communication complexity, most of our attacks have a query complexity of a few thousand requests at most, while some have as little as a few tens of requests. Our most communication intensive attacks (section 5) require less than

$35K(\approx 2^{15})$ requests. Sample benchmarks that we performed in various applications and server installations show that on average one can perform up to 2^{22} requests in the course of a day.

2 PHP System

We will now describe functionalities of the PHP system that are relevant to our attacks. We first describe the different modes in which PHP might be running, and then we will do a description of the randomness generation functions in PHP. We focus our analysis in the Apache web server, the most popular web server at the time of this writing, however our attacks are easily ported to any webserver that meets the configuration requirements that we describe for each attack.

2.1 Process management

There are different ways in which a PHP script is executed. These ways affect its internal states, and thus the state of its PRNGs. We will focus on the case when PHP is running as an Apache module, which is the default installation in most Linux distributions and is also very popular in Windows installations.

mod_php: Under this installation the Apache web server is responsible for the process management. When the server is started a number of child processes are created and each time the number of occupied processes passes a certain threshold a new process is created. Conversely, if the idle processes are too many, some processes are killed. One can specify a maximum number of requests for each process although this is not enabled by default. Under this setting each PHP script runs in the context of one of the child processes, so its state is preserved under multiple connections unless the process is killed by the web server process manager. The configuration is similar in the case the web server uses threads instead of processes.

Keep-Alive requests. The HTTP protocol offers a request header, called Keep-Alive. When this header is set in an HTTP request, the web server is instructed to keep the connection alive after the request is served. Under mod_php installations this means that any subsequent request will be handled from the same process. This is a very important fact, that we will use in our attacks. However in order to avoid having a process hang from one connection for infinite time, most web servers specify an upper bound on the number of consequent keep-alive requests. The default value for this bound in the Apache web server is 100.

2.2 Randomness Generation

In order to satisfy the need for generating randomness in a web application, PHP offers a number of different

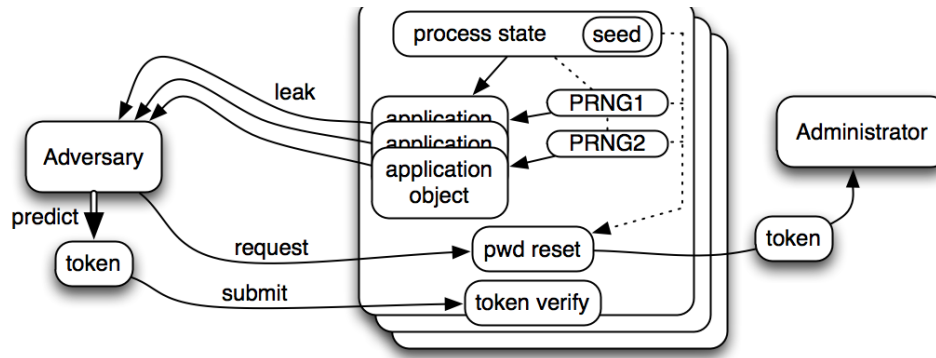


Figure 1: Attack template.

randomness functions. We briefly describe each function below.

- `php_combined_lcg()/lcg_value()`: the `php_combined_lcg()` function is used internally by the PHP system, while `lcg_value()` is its public interface. This function is used in order to create sessions, as well as in the `uniqid` function described below to add extra entropy. It uses two linear congruential generators (LCGs) which it combines in order to get better quality numbers. The output of this function is 64 bits.
- `uniqid(prefix, extra_entropy)`: This function returns a string concatenation of the seconds and microseconds of the server time converted in hexadecimal. When given an additional argument it will prefix the output string with the prefix given. If the second argument is set to true, the function will suffix the output string with an output from the `php_combined_lcg()` function. This makes the total output to have length up to 15 bytes without the prefix.
- `microtime(), time()`: The function `microtime()` returns a string concatenation of the current microseconds divided by 10^6 with the seconds obtained from the server clock. The `time()` function returns the number of seconds since Unix Epoch.
- `mt_srand(seed)/mt_rand(min, max)`: `mt_rand` is the interface for the Mersenne Twister (MT) generator [15] in the PHP system. In order to be compatible with the 31 bit output of `rand()`, the LSB of the MT function is discarded. The function takes two optional arguments which map the 31 bit number to the `[min,max]` range. The `mt_srand()` function is used to seed the MT generator with the 32 bit value `seed`; if no seed is provided then the seed is provided by the PHP system.
- `srand(seed)/rand(min, max)`: `rand` is the interface function of the PHP system to the `rand()` function provided by `libc`. In unix, `rand()` ad-

ditive feedback generator (resembling a Linear Feedback Shift Register (LFSR)), while in Windows it is an LCG. The numbers generated by `rand()` are in the range $[0, 2^{31} - 1]$ but like before the two optional arguments give the ability to map the random number to the range `[min,max]`. Like before the `srand()` function seeds the generator similarly to the `mt_srand()` function.

- `openssl_random_pseudo_bytes(length, strong)`: This function is the only function available in order to obtain cryptographically secure random bytes. It was introduced in version 5.3 of PHP and its availability depends on the availability of the `openssl` library in the system. In addition, until version 5.3.4 of PHP this function had performance problems [2] running in Windows operating systems. The `strong` parameter, if provided, is set to true if the function returned cryptographically strong bytes and false otherwise. For these reasons, and for backward compatibility, its use is still very limited in PHP applications.

In addition the application can utilize an operating system PRNG (such as `/dev/urandom`). However, this does not produce portable code since `/dev/urandom` is unavailable in Windows OS.

3 The entropy of time measurements

Although ill-advised (e.g., [5]) many web applications use time measurements as an entropy source. In PHP, time is accessed through the `time()` and `microtime()` functions. Consider the following problem. At some point a script executing a request made by the attacker makes a time measurement and use the results to, say, generate a password reset token. The attacker's goal it to predict the output of the measurement made by the PHP script. The `time()` function has no entropy at all from an attacker point of view, since the server reveals its time in the HTTP response header as dictated by the HTTP protocol. On the other

hand, microtime ranges from 0 to 10^6 giving a maximum entropy of about 20 bits. We develop two distinct attacks to reduce the entropy of `microtime()` that have different advantages and mostly target two different scenarios. The first one, Adversarial Time Synchronization, aims to predict the output of a specific time measurement when there is no access to other such measurements. The second, Request Twins, exploits the fact that the script may enable the attacker to generate a correlated leak to the target measurement.

Adversarial Time Synchronization (ATS). As we mentioned above, in each HTTP response the web server includes a header containing the full date of the server including hour, minutes and seconds. The basic observation is that although we get no leak regarding the microseconds from the HTTP date header we know that when a second changes the microseconds are zeroed. We use this observation to narrow down their value.

The algorithm proceeds as follows: We connect to the web server and issue pairs of HTTP requests $R1$ and $R2$ in corresponding times $T1$ and $T2$ until a pair is found in which the date HTTP header of the corresponding responses is different. At that point we know that between the processing of the two HTTP requests the microseconds of the server were zeroed. We proceed to approximate the time of this event S in localtime, denoted by the timestamp D , by calculating the average RTT of the two requests and offsetting the middle point between $T2$ and $T1$ by this value divided by two.

In the Apache web server the date HTTP header is set after processing the request of the user. If the attacker requests a non existent file, then the point the header is set is approximately the point that a valid request will start executing the PHP script. It follows that if the attacker uses ATS with HTTP requests to not existent files then he will synchronize approximately with the beginning of the script's execution. Given a steady network where each request takes $\frac{RTT}{2}$ time to reach the target server, our algorithm deviation depends only on the rate that the attacker can send HTTP requests. In practice, we find that the algorithm's main source of error is the network distance between the attacker's system and the server cf. Figure 3. The above implementation we described is a proof-of-concept and various optimizations can be applied to improve its accuracy.

Request Twins. Consider the following setting: an application uses `microtime()` to generate a password token for any user of the system. The attacker has access to a user account of the application and tries to take over the account of another user. This allows the attacker to obtain password reset tokens for his account and thus outputs of the `microtime()` function. The key observation is that if the attacker performs in rapid

succession two password reset requests, one for his account and one for the target user's account, then these requests will be processed by the application with a very small time difference and thus the conditional entropy of the target user's password reset token given the attacker's token will be small. Thus, the attacker can generate a token for an account he owns and in fast succession a token for the target account. Then the `microtime()` used for generating the token of his account can be used to approximate the `microtime()` output that was used for the token of the target account.

Experiments. We conducted a series of experiments for both our algorithms using the following setup. We created a PHP "time" script that prints out the current seconds and microseconds of the server. To evaluate the ATS algorithm we first performed synchronization between a client and the server and afterwards we sent a request to the time script and tried to predict the value it would return. To evaluate the Request Twins algorithm we submitted two requests to the time script in fast succession and measured the difference between the output of the two responses.

In Figure 3 we show the time difference between the server's time and our client's calculation for four servers with different CPU's and RTT parameters. Our experiments suggest that both algorithms significantly reduce the entropy of microseconds (up to an average of 11 bits with ATS and 14 bits with Request Twins) having different advantages each. Specifically, the ATS algorithm seems to be affected by large RTT values while it is less affected by differences in the CPU speed. The situation is reversed for Request Twins where the algorithm is immune to changes in the RTT however, it is less effective in old systems with low processing speed.

4 Seed Attacks

In this section we describe attacks that allow either the recovery or the reconstruction of the seeds used for the PHP system's PRNGs. This allows the attacker to predict all future iterations of these functions and hence reduces the entropy of functions `rand()`, `mt_rand()`, `lcg_value()` as well as the extra entropy argument of `uniqid()` to zero bits. We exploit two properties of the seeds used in these functions. The first one is the reuse of entropy sources between different seeds. This enables us to reconstruct a seed without any access to outputs of the respective PRNG. The second is the small entropy of certain seeds that allows one to recover its value by bruteforce.

We present three distinct attacks. The first attack allows one to recover the seed of the internal LCG seed used by the PHP system using a session identifier. Using that seed our second attack reconstructs the seed of `rand()` and `mt_rand()` functions from the elements of the LCG seed without any access to outputs of these

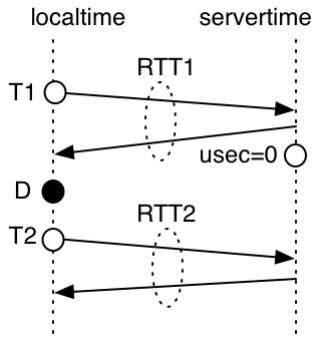


Figure 2: ATS.

Configuration		ATS			Req. Twins		
CPU(GHz)	RTT(ms)	min	max	avg	min	max	avg
1 × 3.2	1.1	0	4300	410	0	1485	47
4 × 2.3	8.2	5	76693	4135	565	1669	1153
1 × 0.3	9	53	39266	2724	1420	23022	4849
2 × 2.6	135	73	140886	83573	2	1890	299

Figure 3: Effectiveness of our time entropy lowering techniques against four servers of different computational power and RTT. Time measurements are in microseconds.

functions. Finally, we exploit the fact that the seed used in these functions is small enough for someone with access to the output of these functions to recover its value by bruteforce.

Generating fresh processes. Our attacks on this section rely on the ability of the attacker to connect to a process with a newly initialized state. We describe a generic technique against `mod_php` in order to achieve a connection to a fresh process. Recall that in `mod_php` when the number of occupied processes passes a certain threshold new processes are created to handle the new connections. This gives the attacker a way to force the creation of fresh processes: The attacker creates a large number of connections to the target server with the keep-alive HTTP header set. Having occupied a large number of processes the web server will create a number of new processes to handle subsequent requests. The attacker, keeping the previous connections open, makes a new one which, given that the attacker created enough connections, will be handled by a fresh process.

4.1 Recovering the LCG seed from Session ID's

In this section we present a technique to recover the `php_combined_lcg()` seed using a PHP session identifier. In PHP, when a new session is created using the respective PHP function (`session_start()`), a pseudorandom string is returned to the user in a cookie, in order to identify that particular session. That string is generated using a conjunction of user specific and process specific information, and then is hashed using a hash function which is by default MD5, however there is an option to use other hash functions such as SHA-1. The values contained in the hash are:

- Client IP address (32 bits).
- A time measurement: Unix epoch and microseconds (32 + 20 bits).
- A value generated by `php_combined_lcg()` (64 bits).

Notice now that in the context of our attack model the attacker controls each request thus he knows ex-

actly most of the values. Specifically, the client IP address is the attacker's IP address and the Unix Epoch can be determined through the date HTTP header. In addition, if `php_combined_lcg()` is not initialized at the time the session is created, as it happens when a fresh process is spawned, then it is seeded. The state of the `php_combined_lcg()` is two registers s_1 , s_2 of size 32 bits each, which are initialized as follows. Let T_1 and T_2 be two subsequent time measurements. Then we have that

$$s_1 = T_1.sec \oplus (T_1.usec \ll 11) \text{ and } s_2 = pid \oplus (T_2.usec \ll 11)$$

where pid denotes the current process id, or if threads are used the current thread id².

Process id's have a range of 2^{15} values in Linux systems. In Windows systems the process id's (resp. threads) are also at most 2^{15} unless there are more than 2^{15} active processes (resp. threads) in the system which is a very unlikely occurrence.

Observe now that the session calculation involves three time measurements T_0 , T_1 and T_2 . Given that these three measurements are conducted successively it is advantageous to estimate their entropy by examining the random variables $\Delta_1 = T_1 - T_0$, $\Delta_2 = T_2 - T_1$. We conducted experiments in different systems to estimate the range of values for Δ_1 and Δ_2 . Our experiments suggest that $\Delta_1 \in [1, 4]$ while $\Delta_2 \in [0, 3]$. We also found a positive linear correlation in the values of the two pairs. This enables a cutdown of the possible valid pairs. These results suggest that the additionally entropy introduced by the two Δ variables is at most 5 bits.

To summarize, the total remaining entropy of the session identifier hash is the sum of the microseconds entropy from T_0 (≈ 20 bits) the two Δ variables (≈ 5 bits) and the process identifier (15 bits). These give a total of 40 bits which is tractable cf. section 1.1. Furthermore the following improvements can be made: (1) Using the ATS algorithm the microseconds entropy can be reduced as much as 11 bits on average. (2) The attacker can make several connections to fresh processes instead of one, in rapid succession, obtaining

²In PHP versions before 5.3.2 the seed used only one time measurement which made it even weaker.

session identifiers from each of the processes. Because the requests were made in a small time interval the preimages of the hashes obtained belong into the same search space, thus improving the probability of inverting one of the preimages proportionally to the number of session identifiers obtained. Our experiments with the request twins technique suggest that at least 4 session identifiers can be obtained from within the same search space thus offering a reduction of at least two bits. Adding these improvements reduces the search time up to 2^{27} MD5 computations.

4.2 Reconstructing the PRNG Seed from Session ID's

In this section we exploit the fact that the PHP system reuses entropy sources between different generators, in order to reconstruct the PRNG seed used by `rand()` and `mt_rand()` functions from a PHP session identifier. In order to predict the seed we only need to find a preimage for the session id, using the methods described in the previous section. One advantage of this attack is that it requires no outputs from the affected functions.

When a new process is created the internal state of the functions `rand()` and `mt_rand()` is uninitialized. Thus, when these functions are called for the first time within the script a seed is constructed as follows:

$$seed = (epoch \times pid) \oplus (10^6 \times php_combined_lcg())$$

where *epoch* denotes the seconds since epoch and *pid* denotes the process id of the process handling the request. It is easy to notice, that an attacker with access to a session id preimage has all the information needed in order to calculate the seed used to initialize the PRNGs since:

- *epoch* is obtained through the HTTP Date header.
- *pid* is known from the seed of the `php_combined_lcg()` obtained through the preimage of the session id from section 4.1.
- `php_combined_lcg()` is also known, since the attacker has access to its seed, he can easily predict the next iteration after the initial value.

In summary the technique of this section allows the reconstruction of the seed of the `mt_rand()` and `rand()` functions given access to a PHP session id of a fresh process. The time complexity of the attack is the same as the one described in section 4.1 while the query complexity is one request, given that the attacker spawned a fresh process (which itself requires only a handful of requests).

4.3 Recovering the Seed from Application leaks

In contrast to the technique presented in the previous section, the attack presented here recovers the seed of

the PRNG functions `rand()` and `mt_rand()` when the attacker has access to the output of these functions. We exploit the fact that the seed used by the PHP system is only 32 bits. Thus, an attacker who connects to a fresh process and obtains a number of outputs from these functions can bruteforce the 32 bit seed that produces the same output.

We emphasize that this attack works even if the outputs are truncated or passed through transformations like hash functions. The requirements of the attack is that the attacker can define a function from the set of all seeds to a sufficiently large range and can obtain a sample of this function evaluated on the seed that the attacker tries to recover. Additionally for the attack to work this function should behave as a random map.

Consider the following example. The attacker has access to a user account of an application which generates a password reset token as 6 symbols where each symbol is defined as $g(mt_rand())$ where g is a table lookup function for a table with 60 entries containing alphanumeric characters. The attacker defines the function f to be the concatenation of two password reset tokens generated just after the PRNG is initialized. The attacker samples the function by connecting to a fresh process and resetting his password two times. Since the table of function g contains 60 entries, the attacker obtains 6 bits per token symbol, giving a total range to the function f of 72 bits.

The time complexity of the attack is 2^{32} calculations of f however, we can reduce the online complexity of the attack using a time-space tradeoff. In this case the online complexity of the attack can be as little as 2^{16} . The query complexity of the attack depends on the number of requests needed to obtain a sample of f . In the example given above the query complexity is two requests.

5 State recovery attacks

One can argue that randomness attacks can be easily thwarted by increasing the entropy of the seeding for the PRNG functions used by the PHP system. For example, the suhosin PHP hardening extension replaces the `rand()` function with a Mersenne Twister generator with separate state from `mt_rand()` and offers a larger seed for both generators getting entropy from the operating system³.

We show that this is not the case. We exploit the algebraic structure of the PRNGs used in order to recover their internal state after a sufficient number of past outputs (leaks) have been observed by the attacker. Any such attack has to overcome two challenges. First, web applications usually need only a small range of

³The suhosin patch installed in some Unix operating systems by default does not include the randomness patches, rather than it mainly offers protection from memory corruption attacks. The full extension is usually installed separately from the PHP packages.

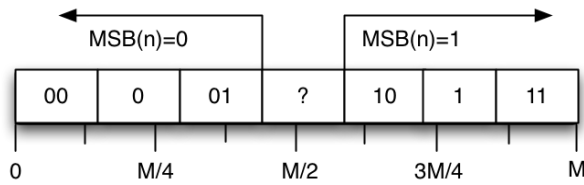


Figure 4: Mapping a random number $n \in [M]$ to 7 buckets and the respective bits of n that are revealed given each bucket.

random numbers, for example to sample a random entry from an array. To achieve that, the PHP system maps the output of the PRNG to the given range, an action that may break the linearity of the generators. Second, in order to collect the necessary leaks the attacker may need to reconnect to the same process many times to collect the leaks from the same generator instance. Since, there could be many PHP processes running in the system, this poses another challenge for the attacker.

In this section we present state recovery algorithms for the truncated PRNG functions `rand()` and `mt_rand()`. The algorithm for the latter function is novel, while regarding the former we implement and evaluate the Håstad-Shamir cryptanalytic framework [8] for truncated linearly related variables. We begin by discussing the way truncation takes place in the PHP system. Afterwards, we tackle the problem of reconnecting into the same server process. Finally we present the two algorithms against the generators.

5.1 Truncating PRNG sequences in the PHP system

As mentioned in section 2.2 the `rand()` and `mt_rand()` functions can map their output to a user defined range. This has the effect of truncating the functions' output. Here we discuss the process of truncating the output and its implications for the attacker.

Let $n \in [M] = \{0, \dots, M-1\}$ be a random number generated by `rand()` or `mt_rand()`, where $M = 2^{31}$ in the PHP system. In order to map that number in the range $[a, b]$ where $a < b$ the PHP system maps n to a number $l \in [a, b]$ in the following way:

$$l = a + \frac{n \cdot (b - a + 1)}{M}$$

We can view the process above as a mapping from the set of numbers in the range $[M]$ to $b - a + 1$ "buckets." Our goal is to recover as many bits as possible of the original number n . Observe that given l it is possible to recover immediately up to $\lfloor \log(b - a + 1) \rfloor$ most significant bits (MSB) of the original number n as follows:

Given that n belongs to bucket l we obtain the fol-

lowing range for possible values for n :

$$\lfloor \frac{(l-a) \cdot M}{b-a+1} \rfloor \leq n \leq \lfloor \frac{(l-a+1) \cdot M}{b-a+1} \rfloor - 1$$

Therefore, given a bucket number l we are able to find an upper and lower bound for the original number denoted respectively by L_l and U_l . In order to recover a part of the original number n one can simply find the number of most significant bits of L_l and U_l that are equal and observe that these bits would be the same also in the number n . Therefore, given a bucket l we can compare the MSBs of both numbers and set the MSBs of n to the largest sequence of common most significant bits of L_l, U_l .

Notice that in some cases even the most significant bit of the two numbers are different, thus we are unable to infer any bit of the original number n with absolute certainty. For example, in Figure 4 given that a number falls in bucket 3 we have that $920350134 \leq n \leq 1227133512$. Because $920350134 < 2^{30}$ and $1227133512 > 2^{30}$ we are unable to infer any bit of the original number n .

Another important observation is that this specific truncation algorithm allows the recovery of a fragment of the MSBs of the original number. Therefore, in the following sections we will assume that the truncation occurs in the MSBs and we will describe our algorithms based on MSB truncated numbers. However, all algorithms described work for any kind of truncation.

5.2 Process distinguisher

As we mentioned in section 2.1, if one wants to receive a number of leaks from the same PHP process one can use keep-alive requests. However, there is an upper bound that limits the number of such requests (by default 100). Therefore, if the attacker needs to observe more outputs beyond the keep-alive limit the connection will drop and when the attacker connects back to the server he may be served from a different process with a different internal state. Therefore, in order to apply state recovery attacks (which typically require more than 100 requests), we must be able to submit all the necessary requests to the same process. In this section, we will describe a generic technique that finds the same process over and over using the PHP session leaks described in section 4.1.

While we cannot avoid disconnecting from a process after we have submitted the maximum number of keep-alive requests, we can start reconnecting back to the server until we hit the process we were connected before and continue to submit requests. The problem in applying this approach is that it is not apparent to distinguish whether the process we are currently connected to is the one that was serving us in the previous connection. To distinguish between different processes, we can use the preimage from a session identifier. Recall that the session id contains a value from

the `php_combined_lcg()` function, which in turn uses process specific state variables. Thus, if the session is produced from the same process as before then the `php_combined_lcg()` will contain the next state from the one it was before. This gives us a way to find the correct process among all the server processes running in the server. In summary the algorithm will proceed as follows:

1. The attacker obtains a session identifier and a preimage for that id using the techniques discussed in section 4.1.
2. The attacker submits the necessary requests to obtain leaks from the PRNG, using the keep-alive HTTP header until the maximum number of requests is reached.
3. The attacker initiates connections to the server requesting session identifiers. He attempts to obtain a preimage for every session identifier using the next value of the `php_combined_lcg()` from the one used before or, if the server has high traffic, the next few iterations. If a preimage is obtained the attacker repeats step 2, until all necessary leaks are obtained.

Notice that obtaining a preimage after disconnecting requires to bruteforce a maximum number of 20 bits (the microseconds), and thus testing for the correct session id is an efficient procedure. Even if the application is not using PHP sessions, or if a preimage cannot be obtained, there are other, application specific, techniques in order to find the correct process.

A generic technique for Windows. In the case of Windows systems the attacker can employ another technique to collect the necessary leaks from the same process in case the server has low traffic. In unix servers with apache preforked server + `mod_php` all idle processes are in a queue waiting to handle an incoming client. The first process in the queue handles a client and then the process goes to the back of the queue. Thus, if an attacker wants to reconnect to the same process without using some process distinguisher he will need to know exactly the number of processes in the system and if there are any intermediate requests by other clients while the attacker tries to reconnect to the same process. However, in Windows prethreaded server with `mod_php` things are slightly better for an attacker. Threads are in a priority queue and when a thread in the first place of the queue handles a request from a client it returns again in that first place and handles the first subsequent incoming request. Thus, an attacker which manages to connect to that first thread of the server, can rapidly close and reopen the connections thus leaving a very small window in which that thread could be occupied by another client. Of course, in high traffic servers the attacker would have a difficulty connecting in a time when the server is idle in

the first place. Nevertheless, techniques exist [16] to remotely determine the traffic of a server and thus allow the attacker to find an appropriate time window within which he will attempt this attack.

Based on the above, in the following sections we will assume that the attacker is able to collect the necessary number of leaks from the targeted function.

5.3 State recovery for `mt_rand()`

The `mt_rand()` function uses the Mersenne Twister generator in order to produce its output. In this section we give a description of the Mersenne Twister generator and present an algorithm that allows the recovery of the internal state of the generator even when the output is truncated. Our algorithm also works in the presence of non consecutive outputs as in the case resulting from the buckets truncation algorithm of the PHP system (cf. section 5.1).

Mersenne Twister. Mersenne Twister, and specifically the widely used MT19937 variant, is a linear PRNG with a 624 32-bit word state. The MT algorithm is based on the following recursion: for all k ,

$$x_{k+n} = x_{k+m} \oplus ((x_k \wedge 0x80000000) | (x_{k+1} \wedge 0x7fffffff))A$$

where $n = 624$ and $m = 397$. The logical AND operation with `0x80000000` discards all but the most significant bit of x_k while the logical AND with `0x7fffffff` discards only the MSB of x_{k+1} . A is a 32×32 matrix for which multiplication by a vector x is defined as follows:

$$xA = \begin{cases} (x \gg 1) & \text{if } x^{31} = 0 \\ (x \gg 1) \oplus a & \text{if } x^{31} = 1 \end{cases}$$

Here $a = (a^0, a^1, \dots, a^{31}) = 0x9908B0DF$ is a constant 32-bit vector (note that we use x^{31} to denote the LSB of a vector x). The output of this recurrence is finally multiplied by a 32×32 non singular matrix T , called the tempering matrix, in order to produce the final output $z = xT$.

State recovery. Since the tempering matrix T is non singular, given 624 outputs of the MT generator one can easily compute the original state by multiplying the output z with the inverse matrix T^{-1} thus obtaining the state variable used as $x_i = z_i T^{-1}$. After recovering 624 state variables one can predict all future iterations. However, when the output of the generator is truncated, predicting future iterations is not as straightforward as before because it is not possible to locally recover all needed bits of the state variables given the truncated output.

The key observation in recovering the internal state is that due to the fact that the generator is in $GF(2)$ the truncation does not introduce non linearity even though there are missing bits from the respective equations.

Thus, we can express the output of the generator as a set of linear equations in GF(2) which, when solved, yield the initial state that produced the observed sequence. From the basic recurrence of MT we can derive the following equations for each individual bit:

Lemma 5.1. *Let x_0, x_1, \dots be an MT sequence and $j > 0$. Then the following equations hold for any $k \geq 0$:*

1. $x_{jn+k}^0 = x_{(j-1)n+k+m}^0 \oplus (x_{(j-1)n+k+1}^{31} \wedge a^0)$
2. $x_{jn+k}^1 = x_{(j-1)n+k+m}^1 \oplus x_{(j-1)n+k}^0 \oplus (x_{(j-1)n+k+1}^{31} \wedge a^1)$
3. $\forall i, 2 \leq i \leq 31 : x_{jn+k}^i = x_{(j-1)n+k+m}^i \oplus x_{(j-1)n+k+1}^{i-1} \oplus (x_{(j-1)n+k+1}^{31} \wedge a^i)$

Proof. The equations follow directly from the basic recurrence.

In addition since the tempering matrix is only a linear transformation of the bits of the state variable x_i , we can similarly express each bit of the final output of MT as a linear equation of the bits of the respective state variable.

To recover the initial state of MT, we generate all equations over the state bit variables $x_0, x_1, \dots, x_{19936}$. To map any position in the MT sequence in an equation over this set of variables, we apply the equations of the lemma above recursively until all variables in the right hand side have index below 19937.

Depending on the positions observed in the MT sequence the resulting linear system will be different. The question that remains is whether that system is solvable. Regarding the case of the 31-bit truncation, i.e. only the MSB of the output word is revealed, we can use known properties of the generator in order to easily prove the following:

Lemma 5.2. *Suppose we obtain the MSB of 19937 consecutive words from the MT generator. Then the resulting linear system is uniquely solvable.*

Proof. It is known that the MT sequence is 19937-distributed to 1-bit accuracy⁴. The linear system is uniquely solvable iff the rows are linearly independent. Suppose that a set $k \leq 19937$ of rows are linearly dependent. Then the last row of the set k obtained is computable from the other members of the k -set something that contradicts the order of equidistribution of MT. \square

The above result is optimal in the sense that this is the minimum number of observed outputs needed for the system to become fully determined. In the case we obtain non consecutive outputs due to truncation

⁴Suppose that a sequence is k -distributed to u -bit accuracy. Then knowledge of the u most significant bits of l words does not allow one to make any prediction for the u bits of the next word when $l < k$. This is the cryptographic interpretation of the “order of equidistribution” whose exact definition can be found in [15].

or application behavior, linear dependencies may arise between the resulting equations and therefore we may need a larger number of observed outputs.

Because we cannot know in advance when the system will become solvable or the equations that will be included, we employ an online version of Gaussian elimination in order to form and solve the resulting system. In this way, the attacker can begin collecting leaks and gradually feed them to our Gaussian solver until he is notified that a sufficient number of independent equations have been collected. Note that regular Gaussian elimination uses both elementary row and elementary column operations. However, because we do not have in advance the entire linear system we cannot use elementary column operations. Instead we make Gaussian elimination using only elementary row operations and utilize a bookkeeping system to enter equations in their place as they are produced by the leaks supplied to the solver. Our solver employs a sparse vector representation and is capable of solving overdetermined sparse systems of tens of thousands of equations in a few minutes.

We ran a sequence of experiments to determine the solvability of the system when a different number of bits is truncated from the output. In addition we ran experiments when the outputs of the MT generator is passed through the PHP truncation algorithm, with different user defined ranges. All experiments were conducted in a 4×2.3 GHz machine with 4 GB of RAM.

In Figure 5 we present the number of equations needed when the PHP truncation algorithm is used. In the x-axis we have the logarithm of the number of buckets. We also show the standard deviation appearing as vertical bars. It can be seen that the number of equations needed is much higher than the theoretical lower bound of 19937 and fluctuates between 27000 and 33000. Nevertheless, the number of leaks required is decreasing linearly to the number of buckets we have. The reason is that although we have more linearly dependent equations, the total number of equations we obtain due to the larger number of buckets is bigger.

Implementation error in the PHP system. The PHP system up to current version, 5.3.10, has an error in the implementation of the Mersenne Twister generator (we discovered this during the testing of our solver). Specifically the following basic recurrence is effectively used in the PHP system due to a programming error:

$$x_{k+n} = x_{k+m} \oplus ((x_k \wedge 0x80000000) | (x_{k+1} \wedge 0x7ffffffe) | (x_k \wedge 0x1))A$$

As a result the PHP system uses a different generator which, as it turns out, has slightly more linear dependencies than the MT generator. This means that probably the randomness properties of the PHP generator are poorer compared to the original MT generator.

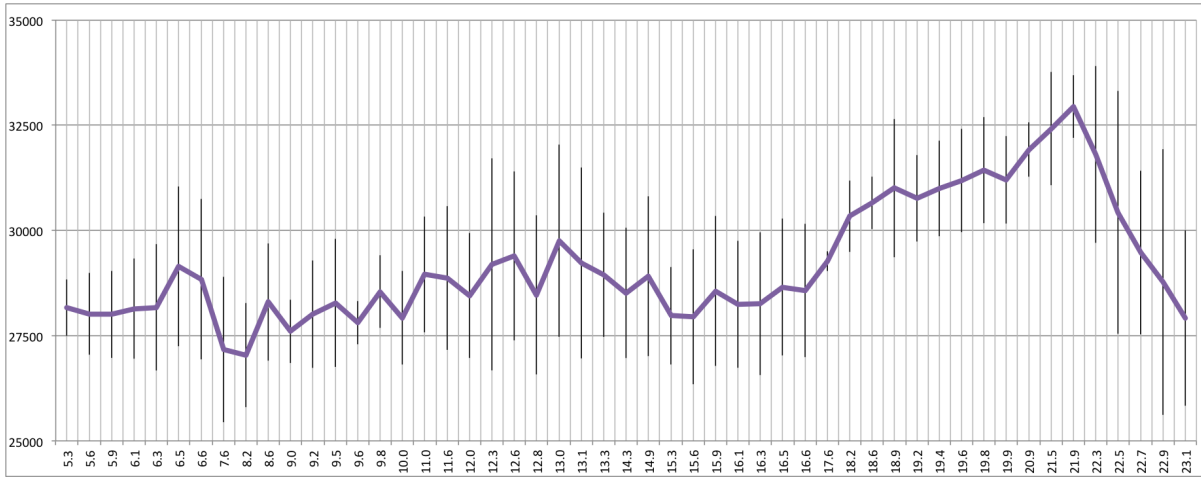


Figure 5: Solving MT; y-axis: number of equations; x-axis: number of buckets (logarithm). Standard deviation shown as vertical bars.

5.4 State recovery for rand()

We turn now to the problem of recovering the state of `rand()` given a sequence of leaks from this generator. While `mt_rand()` is implemented within the PHP source code and thus is unchanged across different environments, the `rand()` function uses the respective function defined from the standard library of the operating system. This results in different implementations across different operating systems. There are mainly two different implementations of `rand()` one from the `glibc` and one from the Windows library.

Windows `rand()`. The `rand()` function defined in Windows is a Linear Congruential Generator (LCG). An LCG is defined by a recurrence of the form

$$X_{n+1} = (aX_n + c) \bmod m$$

Although LCGs are fast and require a small memory footprint there are many problems which make them insufficient for many uses, including of course cryptographic purposes. The parameters used by the Windows LCG are $a = 214013, c = 2531011, m = 2^{32}$. In addition, the output is truncated by default and only the top 15 bits are returned. If PHP is running in a threaded server in Windows then the parameters of the LCG used are $a = 1103515245, c = 12345, m = 2^{15}$.

Glibc `rand()`. In the past, `glibc` also used an LCG for the `rand()` function. Subsequently an LFSR-like “additive feedback” design was adopted. The generator has a state of 31 words (of 32 bits each), over which it is defined by the following recurrence:

$$r_i = (r_{i-3} + r_{i-31}) \bmod 2^{32}$$

In addition the LSB of each word is discarded and the output returned to the user is $o_i = r_i \gg 1$. An interesting note is that the man page of `rand()` states that `rand`

is a non-linear generator. Nevertheless, the non linearity introduced by the truncation of the LSB is negligible and one can easily recover the initial values given enough outputs of the generator.

State recovery. Notice that if the generators used have a small state such as the Windows LCGs then state recovery is easy, by applying the attack from section 4 to bruteforce the entire state of the generator. However, on the `Glibc` generator, which has a state of 992 bits, these attacks are infeasible assuming that the state is random. Although LCGs and the `Glibc` generators are different, they both fall into the same cryptanalytic framework introduced by Håstad and Shamir in 1985 for recovering values of truncated linear variables. This framework allows one to uniquely solve an underdefined system of linear equations when the values of the variables are partially known. In this section we will discuss our experiences with applying this technique in the two aforementioned generators: The LCG and the additive-feedback generator of `glibc`. We will briefly describe the algorithm for recovering the truncated variables in order to discuss our experiments and results. The interested reader can find more information about the algorithm in the original paper [8].

Suppose we are given a system with l linear equations on k variables modulo m denoted by x_1, x_2, \dots, x_k ,

$$a_1^1 x_1 + a_2^1 x_2 + \dots + a_k^1 x_k = 0 \bmod m$$

$$a_1^2 x_1 + a_2^2 x_2 + \dots + a_k^2 x_k = 0 \bmod m$$

...

$$a_1^l x_1 + a_2^l x_2 + \dots + a_k^l x_k = 0 \bmod m$$

where $l < k$ and each variable x_i is partially known. We want to solve the system uniquely by utilizing the partial information of the k variables x_i .

We use the coefficients of the l equations to create a set of l vectors, where each vector is of the form $\mathbf{v}_i =$

(a_1, \dots, a_k) . In addition we add to this set the k vectors $m \cdot \mathbf{e}_i, 0 < i \leq k$. The cryptanalytic framework exploits properties of the lattice L that is defined as the linear span of these vectors. Observe that the dimension of L is k and in addition for every vector $\mathbf{v} \in L$ we have that $\sum_{i=1}^k v_i x_i = 0 \pmod{m}$.

Given the above the attack works as follows: first a lattice is defined using the recurrence that defines the linear generator; then, a lattice basis reduction algorithm is employed to create a set of linearly independent equations modulo m with small coefficients; finally, using the partially known values for each variable, we convert this set of equations to equations over the integers which can be solved uniquely. Specifically, we use the LLL [13] algorithm in order to obtain a reduced basis B for the lattice L . Now because $B = \{\mathbf{w}_j\}$ is a basis, the vectors of B are linearly independent. The key observation is that the lattice definition implies that $\mathbf{w}_j \cdot \mathbf{x} = \mathbf{w}_j \cdot (\mathbf{x}_{\text{unknown}} + \mathbf{x}_{\text{known}}) = d_j \cdot m$ for some unknown d_j . Now as long as $\mathbf{x}_{\text{unknown}} \cdot \mathbf{w}_j < m/2$ (this is the critical condition for solvability) we can solve for d_j and hence recover k equations for $\mathbf{x}_{\text{unknown}}$ which will uniquely determine it.

The original paper provided a relation between the size of $\mathbf{x}_{\text{known}}$ and the number of leaks required from the generator so that the upper bound of $m/2$ is ensured given the level of basis reduction achieved by LLL. In the case of LCGs the paper demanded the modulo m to be squarefree. However, as shown above, in the generators used it holds that $m = 2^{32}$ and thus their arguments do not apply. In addition, the lattice of the additive generator of glibc is different than the one generated by an LCG and thus needs a different analysis.

We conducted a thorough experimental analysis of the framework focusing on the two types of generators above. In each case we tested the maximum possible value of $\mathbf{x}_{\text{known}}$ to see if the $m/2$ bound holds for the reduced LLL basis. In the following paragraphs we will briefly discuss the results of these experiments for these types of generators.

In Figure 6 we show the relationship between the number of leaks required for recovering the state with the lattice-attack and the number of leaks that are truncated for four LCGs: the Windows LCG, the glibc LCG (which are both 32 bits), the Visual Basic LCG (which is 24 bits) and an LCG used in the MMIX of Knuth (which is 64 bits). It is seen that the number of leaks required is very small but increases sharply as more bits are truncated. In all cases the attack stops being useful once the number of truncated bits leaves none but the $\log w - 1$ most significant bits where w is the size of the LCG state. The logarithm barrier seems to be uniformly present and hints that the MSB's of a truncated LCG sequence may be hard to predict (at least using the techniques considered here). A similar logarithmic barrier was also found in the experimental analysis that was conducted by Contini and Shparlinski [3] when they were investigating Stern's attack [17]

against truncated LCG's with secret parameters.

Applying the attack in the glibc additive feedback generator we found that the LLL algorithm became a bottleneck in the algorithm running time; due to its large state the algorithm required a large number of leaks to recover even small truncation levels therefore increasing the lattice dimension that was given to the LLL algorithm. Our testing system (a 3.2GHz cpu with 2GB memory) ran out of memory when 7 bits were truncated. The version of LLL we employed (SageMath 4.8) has time complexity $O(k^5)$ where k is the dimension of the lattice (which represents roughly the number of leaks). The best time-complexity known is $O(k^3 \log k)$ derived from [12]; this may enable much higher truncation levels to be recovered for the glibc generator, however we were not able to test this experimentally as no implementation of this algorithm is publicly available.

We conclude that truncated LCG type of generators can be broken (in the sense of entirely recovering their internal state) for all but extremely high levels of truncation (e.g. in the case of 32-bit state LCG's modulo 2^{32} when they are truncated to 16 buckets or less). For additive feedback type of generators, such as the one in glibc, the situation is similar, however higher recursion depths require more leaks (with a linear relationship) that in turn affect the lattice dimension resulting in longer running times. Comparing the results between the LCGs and the additive feedback generators one may find some justification for the adoption of the latter in recent versions of glibc: it appears that - at least as far as lattice-based attacks are concerned - it is harder to predict truncated glibc sequences (compared to say, Windows LCG's) due to the higher running times of LLL reduction (note though that this does not mean that these are cryptographically secure).

6 Experimental results and Case studies

In order to evaluate the impact of our attacks on real applications we conducted an audit to the password reset function implementations of popular PHP applications. Figure 7 shows the results from our audit. In each case successfully exploiting the application resulted in takeover of arbitrary user accounts⁵ and in some cases, when the administrator interface was affected, of the entire application. In addition to identifying these vulnerabilities we wrote sample exploits for some types of attack we presented, each on one affected application.

⁵The only exception to that is the HotCRP application where passwords were stored in cleartext thus there was no password reset functionality. However, in this case we were able to spoof registrations for arbitrary email accounts.

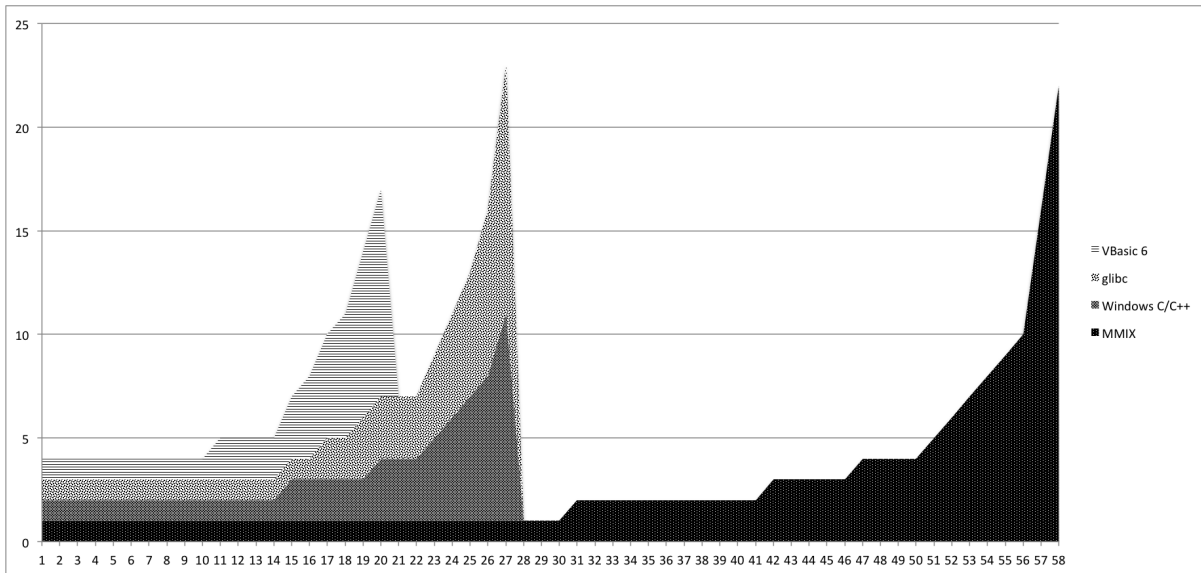


Figure 6: Solving LCGs with LLL; y-axis: number of leaks; x-axis: number of bits truncated.

Application	Attack				Application	Attack			
mediawiki	4.2	4.3	5.3	•	Joomla	4.3			•
Open eClass	4.2	4.3	5.4	•	MyBB	ATS ^c	4.1 ^c	5.3 ^c	○
taskfreak	4.2	4.3	5.3	•	IpBoard	ATS ^c	4.1 ^c	4.2 ^c	•
zen-cart	ATS	RT		•	phorum	4.2	4.3	5.3	•
osCommerce 2.x	ATS	RT		•	HotCRP	4.2	4.3	5.3	•
osCommerce 3.x	4.2	4.3	5.4	•	gazelle	4.3	5.3		•
elgg	ATS ^c	4.2	4.3	•	tikiWiki	4.2	4.3	5.4	•
Gallery	RT ^c	4.1 ^c	4.2 ^c	•	SMF	ATS ^c	4.3 ^c		○

Figure 7: Summary of audit results. The ^c superscript denotes that the attack need to be used in combination with other attacks with the same superscript. The • denotes a full attack while ○ denotes a weakness for which the practical exploitation is either unverified or requires very specific configurations. The number denotes the section in which the applied attack is described in the paper.

6.1 Selected Audit Results

Many applications we audited were trivially vulnerable to our attacks since they used the affected PRNG functions in a straightforward manner, thus making it pretty easy for an attacker to apply our techniques and exploit them. However some applications attempted to defend against randomness attacks by creating custom token generators. We will describe some attacks that resulted from using our framework against custom generators.

Gallery. PHP Gallery is a very popular web based photo album organizer. In order for a user to reset his password he has to click to a link, which contains the security token. The function that generates the token is the following:

```
function hash($entropy="") {
    return md5($entropy . uniqid(mt_rand(), true));
}
```

The token is generated using three entropy sources,

namely a time measurement from `uniqid()`, an output from the MT generator and an output from the `php_combined_lcg()` through the extra argument in the `uniqid()` function. In addition the output is passed through the MD5 hash function so its infeasible to recover the initial values given the output of this function. Since we do not have access to the output of the function, the state reconstruction attack seems an appropriate choice for attacking this token generation algorithm. Indeed, the Gallery application uses PHP sessions thus an attacker can use them to predict the `php_combined_lcg()` and `mt_rand()` outputs. In addition by utilizing the request twins technique from section 3 the attacker can further reduce the search space he has to cover to a few thousand requests.

Joomla. Joomla is one of the most popular CMS applications, and it also have a long history of weaknesses in its generation of password reset tokens [4, 11]. Until recently, the code for the random token generation was the following:

```

function genRandomPassword( $length=8 ) {
    $salt = abc...xyzABC...XYZ0123456789 ;
    $len = strlen ( $salt );
    $makepass = '';
- $stat = @stat ( FILE ) ;
- if (empty($stat) || !isarray($stat))
-     $stat=array(phpuname());
- mt_srand(crc32(microtime().implode(|,$stat)));
  for($i=0;$i<$length;$i++){
      $makepass .= $salt[mt_rand(0,$len1)];
  }
  return $makepass;
}

```

In addition the output of this function is hashed using MD5 along a secret, 16 bytes, key (`config.secret`) which is created at installation using the function above. The `config.secret` value was also used to create a “remember me” cookie in the following way:

```
cookie = md5(config.secret+'JLOGIN REMEMBER')
```

Since the second part of the string is constant and the `config.secret` is generated through the `genRandomPassword` function which has only 2^{32} possible values for each length, one could bruteforce all possible values and recover `config.secret`. All that was left was the prediction of the output of the `genRandomPassword()` function in order to predict the security token used to reset a password. One then observes that although the contents of the `$stat` variable in the `genRandomPassword()` function are sufficiently random, the fact that `crc` is used to convert this value to a 4 byte seed allows one to predict the seed generated and thus the token. This attack was reported in 2010 in [11] and a year after, Joomla released a patch for this vulnerability which removed the custom seeding (dashed lines) from the token generation function. The idea was that because the generator is rolling constantly without reseeding one will be unable to recover the `config.secret` and thus the generator will be secure due to its secret state. Unfortunately, this may not be the case. If at the installation time the process handling the installation script is fresh, a fact quite probable if we consider dedicated servers that do not run other PHP applications, then the search space of the `config.secret` will be again 2^{32} and thus an attacker can use the same technique as before to recover it. After the `config.secret` is recovered, exploitation of the password reset implementation is straightforward using our seed recovery attack from section 4.3. A similar attack also holds when `mod_cgi` is used for script execution as each request will be handled by a fresh process again reducing the search space for `config.secret` in 2^{32} values.

However, the low entropy of the `config.secret` key is not the only problem of this implementation. Even if the key had enough entropy to be totally unpredictable, the generator would still be vulnerable. Notice that in case the `genRandomPassword()` is called

with a newly initialized MT generator then there are at most 2^{32} possible tokens, independently of the entropy of `config.secret`. This gives an interesting attack vector: We generate two tokens from a fresh process sequentially for a user account that we control. Then we start to connect to a fresh process and request a token for our account. If the token matches the token generated before then we can submit a second request for the target user’s account which, since the first token matched the token we own, will match the second token that we requested before (recall that the tokens are not bound to users). Observe that if we generate only one pair of tokens this attack is expected to succeed after 2^{32} requests, assuming that the seed is random. Nevertheless, we can request more than one pair of tokens thus increasing our success probability. Specifically, if we have n pairs of tokens then at the second phase the attack is expected to succeed after $2^{32}/n$ requests. Therefore, if we denote by $r(n)$ the expected requests that the attack needs to hit a “good” token given n initial token pairs, then we have that $r(n) = 2n + 2^{32}/n$. Our goal is to minimize the function $r(n)$; this function obtains a positive minimum at $n = 2^{31/2}$, for which we have that $r(2^{31/2}) \approx 185000$. A simple bruteforcing framework that we wrote was able to achieve around 2500 requests per minute, a rate at which an attacker can compromise the application in a little more than one hour. To be fair, we have to add the requests that are required to spawn new processes but even if we go as far as to double the needed requests (and this is grossly overestimating) we still have a highly practical attack.

Gazelle. Gazelle is a torrent tracker application, which includes a frontend for building torrent sharing communities. It’s been under active development for the last couple of years and its gaining increasing popularity. The interesting characteristic of the application’s password reset implementation is that it uses two generators of the PHP system (namely `rand()` and `mt_rand()`). The code that generates a token is this:

```

function make_secret($Length = 32) {
    $Secret = '';
    $Chars='abcdefghijklmnopqrstuvwxyz0123456789';
    for($i=0; $i<$Length; $i++) {
        $Rand = mt_rand(0, strlen($Chars)-1);
        $Secret .= substr($Chars, $Rand, 1);
    }
    return str_shuffle($Secret);
}

```

The code generates a random string using `mt_rand()` and then shuffles the string using the `str_shuffle()` function which internally uses the `rand()` function. If we try to apply directly the seed recovery attack, i.e. try to ask a question of the form “which seed produces this token” then we will run into problems because we have to take into account two seeds, and a total search space of 64 bits which

is infeasible. The normal action would be to follow the same path as we did in the Gallery application where we had a similar problem and utilize the seed reconstruction attack which does not require an output of the PRNGs. However, the Gazelle application uses custom sessions (which are generated using the same function), and thus we cannot apply that attack either. The solution lies into slightly modifying the seed recovery attack. Instead of asking the question “which seed produces this `mt_rand()` sequence”, which is shuffled and thus affected by the second PRNG, we instead ask which seed produces the *unsorted* set which contains the characters of our string. This set is not affected by the shuffling and thus we can effectively bruteforce the `mt_rand()` seed independently. After recovering the `mt_rand()` seed we know the initial sequence that was produced and we can subsequently recover the seed of `rand()` using the same attack.

6.2 Attacks Implementation

In addition to auditing the applications, we implemented a number of our attacks targeting selected applications. In particular, we implemented a seed recovery attack against Mediawiki, a state reconstruction attack against the Phorum application and the request twins technique against Zen-cart. In the following sections we will briefly describe each vulnerability and the results of our attacks implementation.

Mediawiki. Mediawiki is a very popular wiki application used, among others, by Wikipedia. Mediawiki uses `mt_rand()` in order to generate a new password when the user requests a password reset. In order to predict the generated password we use the seed recovery attack of section 4.3. The function f that we sample is the one used to generate a CSRF token which is the following:

```
function generateToken( $salt = '' ) {
    $token = dehex(mt_rand()).dehex(mt_rand());
    return md5( $token . $salt );
}
```

Our function f given a seed s first seeds the `mt_rand()` generator and then uses that generator to produce a token as the function above. To fully evaluate the practicality of the attack we implemented the attack online, without any time-space tradeoff. Our implementation was able to cover around 1300000 seed evaluations of f per second in a dual-core laptop with two 2.3 GHz processors. This allowed us to cover the full 2^{32} range in about 70 minutes. Of course, using a time-space tradeoff the search time could be further reduced to a few minutes.

Zen cart. Zen-Cart is a popular eCommerce application. At the time of this writing, a sample database which shops enter voluntarily numbers about 2500 active e-shops ⁶. In order to reset a user’s password

zen-cart first seeds the `mt_rand()` generator with the `microtime()` function and then uses the `mt_rand()` function to produce a new password for the user. Thus, there at most 10^6 possible passwords which could be produced. Our exploit used the request twins technique to reset both our password and the target user’s password. Afterwards, we bruteforced the generated password for our account to recover the `microtime()` value that produced it. This takes at most a few seconds on any modern laptop. Then, our exploit bruteforces the passwords generated by `microtime()` values close to the one that generated our own new password. We ran our exploit in a network with RTT around 9 ms, and Zen-Cart was installed in a 4×2.3 GHz server. The average difference of the two passwords was about 3600 microseconds, and the exploit needed at most two times that requests since we don’t know which password was produced first. With the rate of 2500 requests per minute that our implementation achieves, the attack is completed in a few minutes.

Phorum. Phorum is a classic bulletin board application. It was used, among others, by the eStream competition as an online discussion platform. In order for a user to reset his password the following function is used:

```
function phorum_gen_password($charpart=4, $numpart=3)
{
    $vowels = ... //[char array];
    $cons = ... //[char array];
    $num_vowels = count($vowels);
    $num_cons = count($cons);
    $password="";

    for($i = 0; $i < $charpart; $i++){
        $password .= $cons[mt_rand(0, $num_cons - 1)]
            . $vowels[mt_rand(0, $num_vowels - 1)];
    }
    $password = substr($password, 0, $charpart);
    if($numpart){
        $max=(int)str_pad("", $numpart, "9");
        $min=(int)str_pad("1", $numpart, "0");
        $num=(string)mt_rand($min, $max);
    }
    return strtolower($password.$num);
}
```

What makes this function interesting in the context of state recovery is that at if called with no arguments (as it is in the application), at least four `mt_rand()` leaks are discarded in each call. We implemented the attack having the application installed in a Windows server with the Apache web server and we used our generic technique for Windows in order to reconnect to the same process. On average, the attack required around 1100 requests and 11 reconnections of our client. The running time was about 30 minutes, and the main source of overhead was the system solving. This fact is mainly explained from the small number of buckets and the lost leaks of each iteration. Nevertheless, the attack remained highly practical, as we

⁶www.zen-cart.com/index.php?main_page=showcase

were able to compromise any user account (including the administrator) within half an hour.

7 Defending against the Attacks

We believe that a major shortcoming of the PHP core is that it does not provide a native cryptographically secure PRNG and token generator. In fact, a pseudorandom function (PRF) would be the most suitable cryptographic primitive for generating random tokens based on program defined labels; PRF's can be constructed by PRNG's [7]. We feel that this is a shortcoming since developers tend to prefer functions from the core as they are compatible with every different environment PHP is running in. A possible solution would be to introduce a secure PRNG in the PHP core (as a new function). We proposed this solution to the PHP development team which informed us that the development overhead would be too big for supporting such a function and the solution of using `openssl_random_pseudo_bytes()` (which requires OpenSSL) is their recommendation.

On the other hand, administrators can take a number of precautions to defend against randomness attacks using current PHP versions. The Suhosin extension provides a secure seed in the `mt_rand()` and `rand()` functions. The seed exploits the fact that the Mersenne Twister has a large state and fills that state using a hash function. Because `rand()` may have a small state and is dependent from the operating system, the Suhosin extension replaces `rand()` with a Mersenne twister generator with a different state from `mt_rand()`. The hashed values of the seed used are a concatenation of predictable values such as process identifiers and timestamps, along with, potentially, unpredictable ones such as memory addresses of variables and input from `/dev/urandom`. Because the addresses in any modern operating system are randomized through ASLR, as a security precaution, using them as a seed should provide enough additional entropy to make the two seed attacks (sections 4.2, 4.3) infeasible (assuming ASLR addresses are unpredictable). In addition, the suhosin extension ignores the calls to the seeding functions `mt_srand()`, `srand()` in order to defend against weak seeding from the application. Although this may introduce a state recovery vulnerability, in the majority of our case studies, custom seeding was pretty weak and this measure (of securely seeding once and ignoring application based reseeding) increases security. We strongly believe that securely seeding the generators, when possible, is a very useful exploit mitigation for the attacks we presented. Although state recovery attacks would still be possible, these attacks are more complex than the seed attacks which require a handful of requests and commodity hardware to compromise the applications. Furthermore, creating a secure seed from such sources has a negligible performance overhead. There-

fore, such measures should be employed by the PHP system as safeguards for applications that misuse the PHP core PRNGs.

Our session preimage attack (section 4.1) can be mitigated by utilizing an option (disabled by default) of PHP to add extra entropy, from a file, in the session identifier. By specifying `/dev/urandom` as the entropy file, a user can increase the entropy of a session arbitrarily thus making it infeasible for an attacker to obtain a preimage. In Windows, because `/dev/urandom` is not available this option gathers entropy using the same algorithm as in the `openssl_random_pseudo_bytes()` function. The PHP development team informed us that the above option will be enabled by default in the upcoming version, PHP 5.4.

The above workarounds, if employed, will kill our seed attacks and the generic process distinguisher we devised. However, state recovery attacks would still be possible either through some application specific leak, or using the generic technique described for Windows operating systems (section 5.2). In addition, we find the possibility of the existence of other process distinguishers very probable; after all, the process identifier is not considered a cryptographic secret and could be leaked either through the application or the web server or even the operating system itself. Therefore, we feel that even using these workarounds, one should consider state recovery attacks practical.

With the present state of the PHP system, developers should avoid using directly the PRNGs of the PHP core for security purposes. Any application that requires a security token should employ a custom generator, that will either use the functions from the PHP extensions such as the `openssl_random_pseudo_bytes()`, if available, or it will use other entropy sources. We give an example of one such function in [1].

8 Related Work

The first randomness attack in PHP that we are aware of appeared in a blog post by Stefan Esser [5, 6], where he described basic system properties such as keep-alive connection handling by web server processes, and described how misusing `mt_srand()` could result in security vulnerabilities that he demonstrated in some popular applications. Shortly after, the same author released an update of the Suhosin extension which included the randomness features for strong seeding mentioned above. Our preimage attack on PHP sessions was inspired by an attack introduced by Samy Kamkar [10], in which he described some cases where an adversary would be able to guess a PHP session. However these attacks assumed a side-channel of server information. Finally Gregor Kopf [11] described, along other attacks, the vulnerability in the password reset implementation of Joomla. This work describes some type of seed recovery attacks but only

for the case that a fresh seeding occurs within the PHP script executed.

9 Conclusions

We find the fact that the most popular programming language in a domain that has a clear need for cryptographically strong randomness does not have such a generator within its core system to be a security hazard. Still, even if such a generator existed in the language, the misuse of other functions would not disappear immediately as API misuse is a very common security problem in modern systems. Therefore, we believe that research in the practical exploitation of such insecure functions should be continued and extended to other environments even if they do offer better security features in their API than PHP. In this paper we explored the case of PHP installed in the Apache web server along with `mod_php`. We also showed the applicability of some of our attacks in `cgi` mode where each request is handled by a new process. However, the case of `fastcgi` needs further investigation as its behavior depends highly on its configuration. In addition, it would be interesting to check other languages and web servers, such as PHP on an IIS web server, or Python and Ruby on Rails web applications in Apache. A problem that is also of theoretical interest is the development of faster algorithms for recovering truncated linear variables and finding an explanation for the logarithmic barrier we encountered when experimenting with the Håstad-Shamir framework. To conclude, despite the fact that linear generators are cryptographically insecure, the fact that developers misuse them for security critical features makes the analysis of their practical security within a certain application context an interesting research question which we believe needs further attention and awareness.

References

- [1] George Argyros and Aggelos Kiayias. I forgot your password: randomness attacks against php applications. http://crypto.di.uoa.gr/CRYPTO.SEC/Randomness_Attacks.html, 2012.
- [2] Unknown Author. `openssl_random_pseudo_bytes()` painfully slow. PHP Bug # 51636, <https://bugs.php.net/bug.php?id=51636>, 2010.
- [3] Scott Contini and Igor Shparlinski. On stern's attack against secret truncated linear congruential generators. In Colin Boyd and Juan Manuel González Nieto, editors, *ACISP*, volume 3574 of *Lecture Notes in Computer Science*, pages 52–60. Springer, 2005.
- [4] Stefan Esser. Joomla weak random password reset token vulnerability. SektionEins GmbH, Security Advisory 2008/09/11, 2008.
- [5] Stefan Esser. Lesser known security problems in php applications. In *Zend Conference*, 2008.
- [6] Stefan Esser. `mt_srand` and not so random numbers. http://www.suspekt.org/2008/08/17/mt_srand-and-not-so-random-numbers/, 2008.
- [7] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986.
- [8] Johan Håstad and Adi Shamir. The cryptographic security of truncated linearly related variables. In Robert Sedgewick, editor, *STOC*, pages 356–362. ACM, 1985.
- [9] Robert "Hackajar" Imhoff-Dousharm. Economics of password cracking in the gpu era. In *DEFCON 19*, 2011.
- [10] Samy Kamkar. `phpwn`: Attacking sessions and pseudo-random numbers in php. In *Blackhat USA, Las Vegas, NV 2010*, 2010.
- [11] Gregor Kopf. Non-obvious bugs by example. In *27th Chaos Communication Congress CCC*, 2010.
- [12] Henrik Koy and Claus-Peter Schnorr. Segment III-reduction of lattice bases. In Joseph H. Silverman, editor, *CaLC*, volume 2146 of *Lecture Notes in Computer Science*, pages 67–80. Springer, 2001.
- [13] A.K. Lenstra, H.W.jun. Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261:515–534, 1982.
- [14] Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, and Christophe Wachter. Ron was wrong, whit is right. *IACR Cryptology ePrint Archive*, 2012:064, 2012.
- [15] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.
- [16] HD Moore and Val Smith. Tactical exploitation. In *DEFCON 15*, 2007.
- [17] Jacques Stern. Secret linear congruential generators are not cryptographically secure. In *FOCS*, pages 421–426. IEEE Computer Society, 1987.

An Evaluation of the Google Chrome Extension Security Architecture

Nicholas Carlini, Adrienne Porter Felt, and David Wagner

University of California, Berkeley

nicholas.carlini@berkeley.edu, apf@cs.berkeley.edu, daw@cs.berkeley.edu

Abstract

Vulnerabilities in browser extensions put users at risk by providing a way for website and network attackers to gain access to users' private data and credentials. Extensions can also introduce vulnerabilities into the websites that they modify. In 2009, Google Chrome introduced a new extension platform with several features intended to prevent and mitigate extension vulnerabilities: strong isolation between websites and extensions, privilege separation within an extension, and an extension permission system. We performed a security review of 100 Chrome extensions and found 70 vulnerabilities across 40 extensions. Given these vulnerabilities, we evaluate how well each of the security mechanisms defends against extension vulnerabilities. We find that the mechanisms mostly succeed at preventing direct web attacks on extensions, but new security mechanisms are needed to protect users from network attacks on extensions, website metadata attacks on extensions, and vulnerabilities that extensions add to websites. We propose and evaluate additional defenses, and we conclude that banning HTTP scripts and inline scripts would prevent 47 of the 50 most severe vulnerabilities with only modest impact on developers.

1 Introduction

Browser extensions can introduce serious security vulnerabilities into users' browsers or the websites that extensions interact with [20, 32]. In 2009, Google Chrome introduced a new extension platform with several security mechanisms intended to prevent and mitigate extension vulnerabilities. Safari and Mozilla Firefox have since adopted some of these mechanisms for their own extension platforms. In this paper, we evaluate the security of the widely-deployed Google Chrome extension platform with the goal of understanding the practical successes and failures of its security mechanisms.

Most extensions are written by well-meaning developers who are not security experts. These non-expert

developers need to build extensions that are robust to attacks originating from malicious websites and the network. Extensions can read and manipulate content from websites, make unfettered network requests, and access browser userdata like bookmarks and geolocation. In the hands of a web or network attacker, these privileges can be abused to collect users' private information and authentication credentials.

Google Chrome employs three mechanisms to prevent and mitigate extension vulnerabilities:

- *Privilege separation.* Chrome extensions adhere to a privilege-separated architecture [23]. Extensions are built from two types of components, which are isolated from each other: *content scripts* and *core extensions*. Content scripts interact with websites and execute with no privileges. Core extensions do not directly interact with websites and execute with the extension's full privileges.
- *Isolated worlds.* Content scripts can read and modify website content, but content scripts and websites have separate program heaps so that websites cannot access content scripts' functions or variables.
- *Permissions.* Each extension comes packaged with a list of permissions, which govern access to the browser APIs and web domains. If an extension has a core extension vulnerability, the attacker will only gain access to the permissions that the vulnerable extension already has.

In this work, we provide an empirical analysis of these security mechanisms, which together comprise a state-of-the-art least privilege system. We analyze 100 Chrome extensions, including the 50 most popular extensions, to determine whether Chrome's security mechanisms successfully prevent or mitigate extension vulnerabilities. We find that 40 extensions contain at least one type of vulnerability. Twenty-seven extensions contain core extension vulnerabilities, which give an attacker full control over the extension.

Based on this set of vulnerabilities, we evaluate the effectiveness of each of the three security mechanisms. Our primary findings are:

- The isolated worlds mechanism is highly successful at preventing content script vulnerabilities.
- The success of the isolated worlds mechanism renders privilege separation unnecessary. However, privilege separation would protect 62% of extensions if isolated worlds were to fail. In the remaining 38% of extensions, developers either intentionally or accidentally negate the benefits of privilege separation. This highlights that forcing developers to divide their software into components does not automatically achieve security on its own.
- Permissions significantly reduce the severity of half of the core extension vulnerabilities, which demonstrates that permissions are effective at mitigating vulnerabilities in practice. Additionally, dangerous permissions do not correlate with vulnerabilities: developers who write vulnerable extensions use permissions the same way as other developers.

Although these mechanisms reduce the rate and scope of several classes of attacks, a large number of high-privilege vulnerabilities remain.

We propose and evaluate four additional defenses. Our extension review demonstrates that many developers do not follow security best practices if they are optional, so we propose four mandatory bans on unsafe coding practices. We quantify the security benefits and functionality costs of these restrictions on extension behavior. Our evaluation shows that banning inline scripts and HTTP scripts would prevent 67% of the overall vulnerabilities and 94% of the most dangerous vulnerabilities at a relatively low cost for most extensions. In concurrent work, Google Chrome implemented Content Security Policy (CSP) for extensions to optionally restrict their own behavior. Motivated in part by our study [5], future versions of Chrome will use CSP to enforce some of the mandatory bans that we proposed and evaluated.

Contributions. We contribute the following:

- We establish the rate at which extensions contain different types of vulnerabilities, which should direct future extension security research efforts.
- We perform the first large-scale study of the effectiveness of privilege separation when developers who are not security experts are required to use it.
- Although it has been assumed that permissions mitigate vulnerabilities [12, 14, 10], we are the first to evaluate the extent to which this is true in practice.
- We propose and evaluate new defenses. This study partially motivated Chrome’s adoption of a new mandatory security mechanism.

2 Extension Security Background

2.1 Threat Model

In this paper, we focus on non-malicious extensions that are vulnerable to external attacks. Most extensions are written by well-meaning developers who are not security experts. We do not consider malicious extensions; preventing malicious extensions requires completely different tactics, such as warnings, user education, security scans of the market, and feedback and rating systems. Benign-but-buggy extensions face two types of attacks:

- *Network attackers.* People who use insecure networks (e.g., public WiFi hotspots) may encounter network attackers [26, 21]. A network attacker’s goal is to obtain personal information or credentials from a target user. To achieve this goal, a network attacker will read and alter HTTP traffic to mount man-in-the-middle attacks. (Assuming that TLS works as intended, a network attacker cannot compromise HTTPS traffic.) Consequently, data and scripts loaded over HTTP may be compromised.

If an extension adds an *HTTP script* – a JavaScript file loaded over HTTP – to itself, a network attacker can run arbitrary JavaScript within the extension’s context. If an extension adds an HTTP script to an HTTPS website, then the website will no longer benefit from the confidentiality, integrity, and authentication guarantees of HTTPS. Similarly, inserting HTTP data into an HTTPS website or extension can lead to vulnerabilities if the untrusted data is allowed to execute as code.

- *Web attackers.* Users may visit websites that host malicious content (e.g., advertisements or user comments). A website can launch a cross-site scripting attack on an extension if the extension treats the website’s data or functions as trusted. The goal of a web attacker is to gain access to browser userdata (e.g., history) or violate website isolation (e.g., read another site’s password).

Extensions are primarily written in JavaScript and HTML, and JavaScript provides several methods for converting strings to code, such as `eval` and `setTimeout`. If used improperly, these methods can introduce code injection vulnerabilities that compromise the extension. Data can also execute if it is written to a page as HTML instead of as text, e.g., through the use of `document.write` or `document.body.innerHTML`. Extension developers need to be careful to avoid passing unsanitized, untrusted data to these execution sinks.

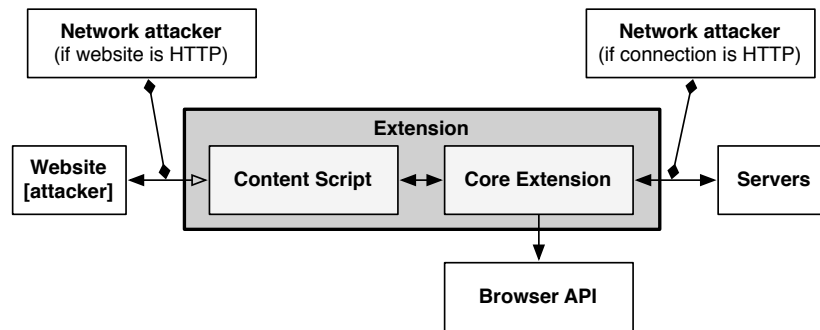


Figure 1: The architecture of a Google Chrome extension.

2.2 Chrome Extension Security Model

Many Firefox extensions have publicly suffered from vulnerabilities [20, 32]. To prevent this, the Google Chrome extension platform was designed to protect users from vulnerabilities in benign-but-buggy extensions [4]. It features three primary security mechanisms:

- *Privilege separation.* Every Chrome extension is composed of two types of components: zero or more *content scripts* and zero or one *core extension*. Content scripts read and modify websites as needed. The core extension implements features that do not directly involve websites, including browser UI elements, long-running background jobs, an options page, etc. Content scripts and core extensions run in separate processes, and they communicate by sending structured clones over an authenticated channel. Each website receives its own separate, isolated instance of a given content script. Core extensions can access Chrome’s extension API, but content scripts cannot. Figure 1 illustrates the relationship between components in a Chrome extension.

The purpose of this architecture is to shield the privileged part of an extension (i.e., the core extension) from attackers. Content scripts are at the highest risk of attack because they directly interact with websites, so they are low-privilege. The sheltered core extension is higher-privilege. As such, an attack that only compromises a content script does not pose a significant threat to the user unless the attack can be extended across the message-passing channel to the higher-privilege core extension.

1.4% of extensions also include binary plugins in addition to content scripts and core extensions [12]. Binary plugins are native executables and are not protected by any of these security mechanisms. We do not discuss the security of binary plugins in this paper because they are infrequently used and must undergo a manual security review before they can be posted in the Chrome Web Store.

- *Isolated worlds.* The isolated worlds mechanism is intended to protect content scripts from web attackers. A content script can read or modify a website’s DOM, but the content script and website have separate JavaScript heaps with their own DOM objects. Consequently, content scripts and websites never exchange pointers. This should make it more difficult for websites to tamper with content scripts.¹
- *Permissions.* By default, extensions cannot use parts of the browser API that impact users’ privacy or security. In order to gain access to these APIs, a developer must specify the desired permissions in a file that is packaged with the extension. For example, an extension must request the bookmarks permission to read or alter the user’s bookmarks. Permissions also restrict extensions’ use of cross-origin XMLHttpRequests; an extension needs to specify the domains that it wants to interact with. Only the core extension can use permissions. Content scripts cannot invoke browser APIs or make cross-origin XHRs.² A content script has only two privileges: it can access the website it is running on, and send messages to its core extension.

Permissions are intended to mitigate core extension vulnerabilities.³ An extension is limited to the permissions that its developer requested, so an attacker cannot request new permissions for a compromised extension. Consequently, the severity of a vulnerability in an extension is limited to the API calls and domains that the permissions allow.

¹Although isolated worlds separates websites from content scripts, it not a form of privilege separation; privilege separation refers to techniques that isolate parts of the same application from each other.

²In newer versions of Chrome, content scripts can make cross-origin XHRs. However, this was not permitted at the time of our study.

³Extension permissions are shown to users during installation, so they may also have a role in helping users avoid malicious extensions; however, we focus on benign-but-buggy extensions in this work.

Google Chrome was the first browser to implement privilege separation, isolated worlds, and permissions for an extension system. These security mechanisms were intended to make Google Chrome extensions safer than Mozilla Firefox extensions or Internet Explorer browser helper objects [4]. Subsequently, Safari adopted an identical extension platform, and Mozilla Firefox's new Add-on SDK (Jetpack) privilege-separates extension modules. All of our study findings are directly applicable to Safari's extension platform, and the privilege separation evaluation likely translates to Firefox's Add-on SDK.

Contemporaneously with our extension review, the Google Chrome extension team began to implement a fourth security mechanism: *Content Security Policy (CSP)* for extensions. CSP is a client-side HTML policy system that allows website developers to restrict what types of scripts can run on a page [29]. It is intended to prevent cross-site scripting attacks by blocking the execution of scripts that have been inserted into pages. By default, CSP disables inline scripts: JavaScript will not run if it is in a link, between `<script>` tags, or in an event handler. The page's policy can specify a set of trusted servers, and only scripts from these servers will execute. Consequently, any attacker that were to gain control of a page would only be able to add code from the trusted servers (which should not lead to harm). CSP can also restrict the use of `eval`, `XHR`, and `iframes`. In Chrome, CSP applies to extensions' HTML pages [28].

3 Extension Security Review

We reviewed 100 Google Chrome extensions from the official directory. This set is comprised of the 50 most popular extensions and 50 randomly-selected extensions from June 2011.⁴ Section 3.1 presents our extension review methodology. Our security review found that 40% of the extensions contain vulnerabilities, and Section 3.2 describes the vulnerabilities. Section 3.3 presents our observation that 31% of developers do not follow even the simplest security best practices. We notified most of the authors of vulnerable extensions (Section 3.4).

3.1 Methodology

We manually reviewed the 100 selected extensions, using a three-step security review process:

1. *Black-box testing.* We exercised each extension's user interface and monitored its network traffic to observe inputs and behavior. We looked for instances of network data being inserted into the

DOM of a page. After observing an extension, we inserted malicious data into its network traffic (including the websites it interacts with) to test potential vulnerabilities.

2. *Source code analysis.* We examined extensions' source code to determine whether data from an untrusted source could flow to an execution sink. After manually reviewing the source code, we used `grep` to search for any additional sources or sinks that we might have missed. For sources, we looked for static and dynamic script insertion, `XMLHttpRequests`, cookies, bookmarks, and reading websites' DOMs. For sinks, we looked for uses of `eval`, `setTimeout`, `document.write`, `innerHTML`, etc. We then manually traced the call graph to find additional vulnerabilities.
3. *Holistic testing.* We matched extensions' source code to behaviors we identified during black-box testing. With our combined knowledge of an extension's source code, network traffic, and user interface, we attempted to identify any additional behavior that we had previously missed.

We then verified that all of the vulnerabilities could occur in practice by building attacks. Our goal was to find all vulnerabilities in every extension.

During our review, we looked for three types of vulnerabilities: vulnerabilities that extensions add to websites (e.g., HTTP scripts on HTTPS websites), vulnerabilities in content scripts, and vulnerabilities in core extensions. Some content script vulnerabilities may also be core extension vulnerabilities, depending on the extensions' architectures. Core extension vulnerabilities are the most severe because the core is the most privileged extension component. We do not report vulnerabilities if the potential attacker is a trusted website (e.g., `https://mail.google.com`) and the potentially malicious data is not user-generated; we do not believe that well-known websites are likely to launch web attacks.

After our manual review, we applied a well-known commercial static analysis tool to six extensions, with custom rules. However, our manual review identified significantly more vulnerabilities, and the static analysis tool did not find any additional vulnerabilities because of limitations in its ability to track strings. Prior research has similarly found that a manual review by experts uncovers more bugs than static analysis tools [30]. Our other alternative, VEX [3], was not built to handle several of the types of attacks that we reviewed. Consequently, we did not pursue static analysis further.

⁴We excluded four extensions because they included binary plugins; they were replaced with the next popular or random extensions. The directory's popularity metric is primarily based on the number of users.

Vulnerable Component	Web Attacker	Network Attacker
Core extension	5	50
Content script	3	1
Website	6	14

Table 1: 70 vulnerabilities, by location and threat model.

Vulnerable Component	Popular	Random	Total
Core extension	12	15	27
Content script	1	2	3
Website	11	6	17
Any	22	18	40

Table 2: The number of extensions with vulnerabilities, of 50 popular and 50 randomly-selected extensions.

3.2 Vulnerabilities

We found 70 vulnerabilities across 40 extensions. The appendix identifies the vulnerable extensions. Table 1 categorizes the vulnerabilities by the location of the vulnerability and the type of attacker that could exploit it. More of the vulnerabilities can be leveraged by a network attacker than by a web attacker, which reflects the fact that two of the Chrome extension platform’s security measures were primarily designed to prevent web attacks. A bug may be vulnerable to both web and network attacks; we count it as a single vulnerability but list it in both categories in Table 1 for illustrative purposes.

The vulnerabilities are evenly distributed between popular and randomly-selected extensions. Table 2 shows the distribution. Although popular extensions are more likely to be professionally written, this does not result in a lower vulnerability rate in the set of popular extensions that we examined. We hypothesize that popular extensions have more complex communication with websites and servers, which increases their attack surface and neutralizes the security benefits of having been professionally developed. The most popular vulnerable extension had 768,154 users in June 2011.

3.3 Developer Security Effort

Most extension developers are not security experts. However, there are two best practices that a security-conscious extension developer can follow without any expertise. First, developers can use HTTPS instead of HTTP when it is available, to prevent a network attacker from inserting data or code into an extension. Second, developers can use `innerText` instead of `innerHTML` when adding untrusted, non-HTML data to a page; `innerText` does not allow inline scripts to execute. We evaluate developers’ use of these best practices in order to determine how security-conscious they are.

We find that 31 extensions contain at least one vulnerability that was caused by not following these two simple best practices. This demonstrates that a substantial fraction of developers do not make use of optional security mechanisms, even if the security mechanisms are very simple to understand and use. As such, we advocate mandatory security mechanisms that force developers to follow best security practices (Section 7).

3.4 Author Notification

We disclosed the extensions’ vulnerabilities to all of the developers that we were able to contact. We found contact information for 80% of the vulnerable extensions.⁵ Developers were contacted between June and September 2011, depending on when we completed each review. We sent developers follow-up e-mails if they did not respond to our initial vulnerability disclosure within a month.

Of the 32 developers that we contacted, 19 acknowledged and fixed the vulnerabilities in their extensions, and 7 acknowledged the vulnerabilities but have not completely fixed them as of February 7, 2012. Two of the un-patched extensions are official Google extensions. As requested, we provided guidance on how the security bugs could be fixed. None of the developers disputed the legitimacy of the vulnerabilities, although one developer argued that a vulnerability was too difficult to fix.

The appendix identifies the extensions that have been fixed. However, the “fixed” extensions are not necessarily secure despite our review. While checking on the status of vulnerabilities, we discovered that developers of several extensions have introduced new security vulnerabilities that were not present during our initial review. We do not discuss the new vulnerabilities in this paper.

4 Evaluation of Isolated Worlds

The isolated worlds mechanism is intended to protect content scripts from malicious websites, including otherwise-benign websites that have been altered by a network attacker. We evaluate whether the isolated worlds mechanism is sufficient to protect content scripts from websites. Our security review indicates that isolated worlds largely succeeds: only 3 of the 100 extensions have content script vulnerabilities, and only 2 of the vulnerabilities allow arbitrary code execution.

Developers face four main security challenges when writing extensions that interact with websites. We discuss whether and how well the isolated worlds mechanism helps prevent these vulnerability classes.

⁵For the remaining 20%, contact information was unavailable, the extension had been removed from the directory, or we were unable to contact the developer in a language spoken by the developer.

Data as HTML. One potential web development mistake is to insert untrusted data as HTML into a page, thereby allowing untrusted data to run as code. The isolated worlds mechanism mitigates this type of error in content scripts. When a content script inserts data as HTML into a website, any scripts in the data are executed within the website's isolated world instead of the extension's. This means that an extension can read data from a website's DOM, edit it, and then re-insert it into the page without introducing a content script vulnerability. Alternately, an extension can copy data from one website into another website. In this case, the extension will have introduced a vulnerability into the edited website, but the content script itself will be unaffected.

We expect that content scripts would exhibit a higher vulnerability rate if the isolated worlds mechanism did not mitigate data-as-HTML bugs. Six extensions' content scripts contained data-as-HTML errors that resulted in web site vulnerabilities, instead of the more-dangerous content script vulnerabilities. Furthermore, we found that 20 of the 50 (40%) core extension vulnerabilities are caused by inserting untrusted data into HTML; core extensions do not have the benefit of the isolated worlds mechanism to ameliorate this class of error. Since it is unlikely that developers exercise greater caution when writing content scripts than when writing core extensions, we conclude that the isolated worlds mechanism reduces the rate of content script vulnerabilities by mitigating data-as-HTML errors.

Eval. Developers can introduce vulnerabilities into their extensions by using `eval` to execute untrusted data. If an extension reads data from a website's DOM and `evals` the data in a content script, the resulting code will run in the content script's isolated world. As such, the isolated worlds mechanism does not prevent or mitigate vulnerabilities due to the use of `eval` in a content script.

We find that relatively few developers use `eval`, possibly because its use has been responsible for well-known security problems in the past [8, 27]. Only 14 extensions use `eval` or equivalent constructs to convert strings to code in their content scripts, and most of those use it only once in a library function. However, we did find two content script vulnerabilities that arise because of an extension's use of `eval` in its content script. For example, the *Blank Canvas Script Handler* extension can be customized with supplemental scripts, which the extension downloads from a website and `evals` in a content script. Although the developer is intentionally running data from the website as code, the integrity of the HTTP website that hosts the supplemental scripts could be compromised by a network attacker.

Click Injection. Extensions can register event handlers for DOM elements on websites. For example, an extension might register a handler for a button's `onClick` event. However, extensions cannot differentiate between events that are triggered by the user and events that are generated by a malicious web site. A website can launch a click injection attack by invoking an extension's event handler, thereby tricking the extension into performing an action that was not requested by the user. Although this attack does not allow the attacker to run arbitrary code in the vulnerable content script, it does allow the website to control the content script's behavior.

The isolated worlds mechanism does not prevent or mitigate click injection attacks at all. However, the attack surface is small because relatively few extensions register event handlers for websites' DOM elements. Of the 17 extensions that register event handlers, most are for simple buttons that toggle UI state. We observed only one click injection vulnerability, in the *Google Voice* extension. The extension changes phone numbers on websites into links. When a user clicks a phone number link, Google Voice inserts a confirmation dialog onto the DOM of the website to ensure that the user wants to place a phone call. Google Voice will place the call following the user's confirmation. However, a malicious website could fire the extension's event handlers on the link and confirmation dialog, thereby placing a phone call from the user's Google Voice account without user consent.

Prototypes and Capabilities. In the past, many vulnerabilities due to prototype poisoning and capability leaks have been observed in bookmarklets and Firefox extensions [20, 32, 2]. The isolated worlds mechanism provides heap separation, which prevents both of these types of attacks. Regardless of developer behavior, these attacks are not possible in Chrome extensions as long as the isolation mechanism works correctly.

Based on our security review, the isolated worlds mechanism is highly effective at shielding content scripts from malicious websites. It mitigates data-as-HTML errors, which we found were very common in the Chrome extensions that we reviewed. Heap separation also prevents prototype poisoning and capability leaks, which are common errors in bookmarklets and Firefox extensions. Although the isolated worlds mechanism does not prevent click injection or `eval`-based attacks, we find that developers rarely make these mistakes. We acknowledge that our manual review could have missed some content script vulnerabilities. However, we find it unlikely that we could have missed many, given our success at finding the same types of vulnerabilities in core extensions. We therefore conclude that the isolated worlds mechanism is effective, and other extension platforms should implement it if they have not yet done so.

5 Evaluation of Privilege Separation

Privilege separation is intended to shield the privileged core extension from attacks. The isolated worlds mechanism serves as the first line of defense against malicious websites, and privilege separation is supposed to protect the core extension when isolated worlds fails. We evaluate the effectiveness of extension privilege separation and find that, although it is unneeded, it would be partially successful at accomplishing its purpose if the isolated worlds mechanism were to fail.

5.1 Cross-Component Vulnerabilities

Some developers give content scripts access to core extension permissions, which removes the defense-in-depth benefits of privilege separation. We evaluate the impact of developer behavior on the effectiveness of extension privilege separation.

Vulnerable Content Scripts. The purpose of privilege separation is to limit the impact of content script vulnerabilities. Even if a content script is vulnerable, privilege separation should prevent an attacker from executing code with the extension's permissions. We identified two extensions with content script vulnerabilities that permit arbitrary code execution; these two extensions could benefit from privilege separation.

Despite privilege separation, both of the vulnerabilities yield access to some core extension privileges. The vulnerable content scripts can send messages to their respective core extensions, requesting that the core extensions exercise their privileges. In both extensions, the core extension makes arbitrary XHRs on behalf of the content script and returns the result to the content script. This means that the two vulnerable content scripts could trigger arbitrary HTTP XHRs even though content scripts should not have access to a cross-origin XMLHttpRequest object. These vulnerable extensions represent a partial success for privilege separation because the attacker cannot gain full privileges, but also a partial failure because the attacker can gain the ability to make cross-origin XHRs.

Hypothetical Vulnerabilities. Due to the success of the isolated worlds mechanism, our set of vulnerabilities only includes two extensions that need privilege separation as a second line of defense. To expand the scope of our evaluation of privilege separation, we explore a hypothetical scenario: if the currently-secure extensions' content scripts had vulnerabilities, would privilege separation mitigate these vulnerabilities?

Of the 98 extensions that do not have content script vulnerabilities, 61 have content scripts. We reviewed the message passing boundary between these content scripts

Permissions	Number of Scripts
All of the extension's permissions	4
Partial: Cross-origin XHRs ²	9
Partial: Tab control	5
Partial: Other	5

Table 3: 61 extensions have content scripts that do not have code injection vulnerabilities. If an attacker were hypothetically able to compromise the content scripts, these are the permissions that the attacker could gain access to via the message-passing channel with the cores.

and their core extensions. We determined that 38% of content scripts can leverage communication with their core extensions to abuse some core extension privileges: 4 extensions' content scripts can use all of their cores' permissions, and 19 can use some of their cores' permissions. Table 3 shows which permissions attackers would be able to obtain via messages if they were able to compromise the content scripts. This demonstrates that privilege separation could be a relatively effective layer of defense, if needed: we can expect that privilege separation would be effective at limiting the damage of a content script vulnerability 62% of the time.

Example. The *AdBlock* extension allows its content script to execute a set of pre-defined functions in the core extension. To do this, the content script sends a message to the core extension. A string in the message is used to index the window object, allowing the content script to select a pre-defined function to run. Unfortunately, this also permits arbitrary code execution because the window object provides access to `eval`. As such, a compromised content script would have unfettered access to the core extension's permissions.

Example. A bug in the *Web Developer* extension unintentionally grants its content script full privileges. Its content script can post small notices to the popup page, which is part of the core extension. The notices are inserted using `innerHTML`. The notices are supposed to be text, but a compromised content script could send a notice with an inline script that would execute in the popup page with full core extension permissions.

5.2 Web Site Metadata Vulnerabilities

The Chrome extension platform applies privilege separation with the expectation that malicious website data will first enter an extension via a vulnerable content script. However, it is possible for a website to attack a core extension without crossing the privilege separation boundary. Website-controlled metadata such as titles and URLs can be accessed by the core extension through browser

Type	Vulnerabilities
Website content	2
Website metadata	5
HTTP XHR	16
HTTP script	28
Total	50

Table 4: The types of core extension vulnerabilities.

managers (e.g., the history, bookmark, and tab managers). This metadata may include inline scripts, and mishandled metadata can lead to a core extension vulnerability. Website metadata does not flow through content scripts, so privilege separation does not impede it. We identified five vulnerabilities from metadata that would allow an attacker to circumvent privilege separation.

Example. The *Speeddial* extension replicates Chrome’s built-in list of recently closed pages. Speeddial keeps track of the tabs opened using the tabs manager and does not sanitize the titles of these pages before adding them to the HTML of one of its core extension pages. If a title were to contain an inline script, it would execute with the core extension’s permissions.

5.3 Direct Network Attacks

Privilege separation is intended to protect the core extension from web attackers and HTTP websites that have been compromised by network attackers. However, the core extension may also be subject to direct network attacks. Nothing separates a core extension from code in HTTP scripts or data in HTTP `XMLHttpRequests`. HTTP scripts in the core extension give a network attacker the ability to execute code with the extension’s full permissions, and HTTP XHRs cause vulnerabilities when extensions allow the HTTP data to execute.

Direct network attacks comprise the largest class of core extension vulnerabilities, as Table 4 illustrates. Of the 50 core extension vulnerabilities, 44 vulnerabilities (88%) stem from HTTP scripts or HTTP `XMLHttpRequests`, as opposed to website data. For example, many extensions put the HTTP version of the Google Analytics script in the core extension to track which of the extensions’ features are used.

Example. Google Dictionary allows a user to look up definitions of words by double clicking on a word. The desired definition is fetched by making a HTTP request to `google.com` servers. The response is inserted into one of the core extension’s pages using `innerHTML`. A network attacker could modify the response to contain malicious inline scripts, which would then execute as part of the privileged core extension page.

5.4 Implications

The isolated worlds mechanism is so effective at protecting content scripts from websites that privilege separation is rarely needed. As such, privilege separation is used to address a threat that almost does not exist, at the cost of increasing the complexity and performance overhead of extensions. (Privilege separation requires an extra process for each extension, and communication between content scripts and core extensions is IPC.) We find that network attackers are the real threat to core extension security, but privilege separation does not mitigate or prevent these attacks. This shows that although privilege separation can be a powerful security mechanism [23], its placement within an overall system is an important determining factor of its usefulness.

Our study also has implications for the use of privilege separation in other contexts. All Chrome extension developers are required to privilege separate their extensions, which allows us to evaluate how well developers who are not security experts use privilege separation. We find that privilege separation would be fairly effective at preventing web attacks in the absence of isolated worlds: privilege separation would fully protect 62% of core extensions. However, in more than a third of extensions, developers created message passing channels that allow low-privilege code to exploit high-privilege code. This demonstrates that forcing developers to privilege separate their software will improve security in most cases, but a significant fraction of developers will accidentally or intentionally negate the benefits of privilege separation. Mandatory privilege separation could be a valuable line of defense for another platform, but it should not be relied on as the only security mechanism; it should be coupled with other lines of defense.

6 Evaluation of the Permission System

The Chrome permission system is intended to reduce the severity of core extension vulnerabilities. If a website or network attacker were to successfully inject malicious code into a core extension, the severity of the attack would be limited by the extension’s permissions. However, permissions will not mitigate vulnerabilities in extensions that request many dangerous permissions. We evaluate the extent to which permissions mitigate the core extension vulnerabilities that we found.

Table 5 lists the permissions that the vulnerable extensions request. Ideally, each permission should be requested infrequently. We find that 70% of vulnerable extensions request the `tabs` permission; an attacker with access to the `tabs` API can collect a user’s browsing history or redirect pages that a user views. Fewer than half of extensions request each of the other permissions.

Permissions	Times Requested	Percentage
tabs (browsing history)	19	70%
all HTTP domains	12	44%
all HTTPS domains	12	44%
specific domains	10	37%
notifications	5	19%
bookmarks	4	15%
no permissions	4	15%
cookies	3	11%
geolocation	1	4%
context menus	1	4%
unlimited storage	1	4%

Table 5: The permissions that are requested by the 27 extensions with core extension vulnerabilities.

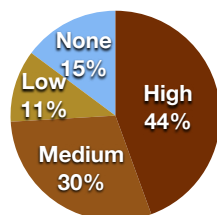


Figure 2: The 27 extensions with core vulnerabilities, categorized by the severity of their worst vulnerabilities.

To summarize the impact of permissions on extension vulnerabilities, we categorized all of the vulnerabilities by attack severity. We based our categorization on the Firefox Security Severity Ratings [1], which has been previously used to classify extension privileges [4]:

- *Critical*: Leaks the permission to run arbitrary code on the user’s system
- *High*: Leaks permissions for the DOM of all HTTP(S) websites
- *Medium*: Leaks permissions for private user data (e.g., history) or the DOM of specific websites that contain financial or important personal data (e.g., `https://*.google.com/*`)
- *Low*: Leaks permissions for the DOM of specific websites that do not contain sensitive data (e.g., `http://*.espn.com`) or permissions that can be used to annoy the user (e.g., fill up storage or make notifications)
- *None*: Does not leak any permissions

We did not find any critically-vulnerable extensions. This is a consequence of our extension selection methodology: we did not review any extensions with binary plugins, which are needed to obtain critical privileges.

Figure 2 categorizes the 27 vulnerable extensions by their most severe vulnerabilities. In the absence of a permission system, all of the vulnerabilities would give an

attacker access to all of the browser’s privileges (i.e., critical privileges). With the permission system, less than half of the vulnerable extensions yield access to high-severity permissions. As such, our study demonstrates that the permission system successfully limits the severity of most vulnerabilities.

We hypothesized that permissions would positively correlate with vulnerabilities. Past work has shown that many extensions are over-permissioned [12, 14], and we thought that developers who are unwilling to follow security best practices (e.g., use HTTPS) would be unwilling to take the time to specify the correct set of permissions. This would result in vulnerable extensions requesting dangerous permissions at a higher rate. However, we do not find any evidence of a positive correlation between vulnerabilities and permissions. The 27 extensions with core vulnerabilities requested permissions at a lower rate than the other 73 extensions, although the difference was not statistically significant. Our results show that developers of vulnerable extensions can use permissions well enough to reduce the privileges of their insecure extensions, even though they lack the expertise or motivation required to secure their extensions.

Permissions are not only used by the Google Chrome extension system. Android implements a similar permission system, and future HTML5 device APIs will likely be guarded with permissions. Although it has been assumed that permissions mitigate vulnerabilities [10, 12, 14], our study is the first to evaluate whether this is true for real-world vulnerabilities or measure quantitatively how much it helps mitigate these vulnerabilities in practice. Our findings indicate that permissions can have a significant positive impact on system security and are worth including in a new platform as a second line of defense against attacks. However, they are not effective enough to be relied on as the only defense mechanism.

7 Defenses

Despite Google Chrome’s security architecture, our security review identified 70 vulnerabilities in 40 extensions. Based on the nature of these vulnerabilities, we propose and evaluate four additional defenses. The defenses are bans on unsafe coding practices that lead to vulnerabilities. We advocate mandatory bans on unsafe coding practices because many developers do not follow security best practices when they are optional (Section 3.3). We quantify the security benefits and compatibility costs of each of these defenses to determine whether they should be adopted. Our main finding is that a combination of banning HTTP scripts and banning inline scripts would prevent 94% of the core extension vulnerabilities, with only a small amount of developer effort to maintain full functionality in most cases.

In concurrent work, Google Chrome implemented Content Security Policy (CSP) for extensions. CSP can be used to enforce all four of these defenses. Initially, the use of CSP was wholly optional for developers. As of Chrome 18, extensions that take advantage of new features will be subject to a mandatory policy; this change was partially motivated by our study [5].

7.1 Banning HTTP Scripts

Scripts fetched over HTTP are responsible for half of the vulnerabilities that we found. All of these vulnerabilities could be prevented by not allowing extensions to add HTTP scripts to their core extensions [15] or to HTTPS websites. Extensions that currently violate this restriction could be easily modified to comply by packaging the script with the extension or using a HTTPS URL. Only vulnerable extensions would be affected by the ban because any extension that uses HTTP scripts will be vulnerable to man-in-the-middle attacks.

Core Extension Vulnerabilities. Banning HTTP scripts from core extensions would remove 28 core extension vulnerabilities (56% of the total core extension vulnerabilities) from 15 extensions. These 15 extensions load HTTP scripts from 13 domains, 10 of which already offer the same script over HTTPS. The remaining 3 scripts are static files that could be downloaded once and packaged with the extensions.

Website Vulnerabilities. Preventing extensions from adding HTTP scripts to HTTPS websites would remove 8 website vulnerabilities from 8 extensions (46% of the total website vulnerabilities). These vulnerabilities allow a network attacker to circumvent the protection that HTTPS provides for websites. The extensions load HTTP scripts from 7 domains, 3 of which offer an HTTPS option. The remaining 4 scripts are static scripts that could be packaged with the extensions.

7.2 Banning Inline Scripts

Untrusted data should not be added to pages as HTML because it can contain inline scripts (e.g., inline event handlers, links with embedded JavaScript, and `<script>` tags). For example, untrusted data could contain an image tag with an inline event handler: ``. We find that 40% of the core extension vulnerabilities are caused by adding untrusted data to pages as HTML. These vulnerabilities could be prevented by not allowing any inline scripts to execute: the untrusted data will still be present as HTML, but it would be static. JavaScript will only run on a page if it is in a separate `.js` file that is stored locally or loaded from a trusted server that the developer has whitelisted.

Banning inline scripts from extension HTML would eliminate 20 vulnerabilities from 15 extensions. All of these vulnerabilities are core extension vulnerabilities. Content script vulnerabilities cannot be caused by inline scripts, and we cannot prevent extensions from adding inline scripts to HTTPS websites because existing enforcement mechanisms cannot differentiate between a website's own inline scripts and extension-added scripts.

However, banning inline scripts has costs. Developers use legitimate inline scripts for several reasons, such as to define event handlers. In order to maintain functionality despite the ban, all extensions would need to delete their inline scripts from HTML and move them to separate `.js` files. Inline event handlers (e.g., `onclick`) cannot simply be copied and pasted; they need to be rewritten as programmatically using the DOM API.

We reviewed the 100 extensions to determine what changes would be needed to comply with a ban on inline scripts. Applying this ban breaks 79% of the extensions. However, all of the extensions could be retrofitted to work without inline scripts without significant changes to the extension. Most of the compatibility costs pertain to moving the extensions' inline event handlers. The extensions contain an average of 7 event handlers, with a maximum of 98 and a minimum of 0 event handlers.

7.3 Banning Eval

Dynamic code generation converts strings to code, and its use can lead to vulnerabilities if the strings are untrusted data. Disallowing the use of dynamic code generation (e.g., `eval` and `setTimeout`) would eliminate three vulnerabilities: one core extension vulnerability, and two vulnerabilities that are both content script and core extension vulnerabilities.

We reviewed the 100 extensions and find that dynamic code generation is primarily used in three ways:

1. Developers sometimes pass static strings to `setTimeout` instead of functions. This coding pattern cannot be exploited. It would be easy to alter instances of this coding pattern to comply with a ban on dynamic code generation; the strings simply need to be replaced with equivalent functions.
2. Some developers use `eval` on data instead of `JSON.parse`. We identified one vulnerability that was caused by this practice. In the absence of dynamic code generation, developers could simply use the recommended `JSON.parse`.
3. Two extensions use `eval` to run user-specified scripts that extend the extensions. In both cases, their error is that they fetch the extra scripts over HTTP instead of HTTPS. For these two extensions, a ban on `eval` would prevent the vulnerabilities but irreparably break core features of the extensions.

Restriction	Security Benefit	Broken, But Fixable	Broken And Unfixable
No HTTP scripts in core	15%	15%	0%
No HTTP scripts on HTTPS websites	8%	8%	0%
No inline scripts	15%	79%	0%
No eval	3%	30%	2%
No HTTP XHRs	17%	29%	14%
All of the above	35%	86%	16%
No HTTP scripts and no inline scripts	32%	80%	0%
Chrome 18 policy	27%	85%	2%

Table 6: The percentage of the 100 extensions that would be affected by the restrictions. The “Security Benefit” column shows the number of extensions that would be fixed by the corresponding restriction.

Richards et al. present additional uses of `eval` in a large-scale study of web applications [24].

We find that 32 extensions would be broken by a ban on dynamic code generation. Most instances can easily be replaced, but 2 extensions would be permanently broken. Overall, a ban on `eval` would fix three vulnerabilities at the cost of fundamentally breaking two extensions.

7.4 Banning HTTP XHR

Network attacks can occur if untrusted data from an `HTTP XMLHttpRequest` is allowed to flow to a JavaScript execution sink. 30% of the 70 vulnerabilities are caused by allowing data from HTTP XHRs to execute. One potential defense is to disallow HTTP XHRs; all XHRs would have to use HTTPS. This ban would remove vulnerabilities from 17 extensions.

However, banning HTTP XHRs would have a high compatibility cost. The only way to comply with an HTTPS-only XHR policy is to ensure that the server supports HTTPS; unlike scripts, remote data cannot be packaged with extensions. Developers who do not control the servers that their extensions interact with will not be able to adapt their extensions. Extension developers who also control the domains may be able to add support for HTTPS, although this can be a prohibitively expensive and difficult process for a novice developer.

We reviewed the 100 extensions and found that 29% currently make HTTP XHRs. All of these would need to be changed to use HTTPS XHRs. However, not all of the domains offer HTTPS. Ten extensions request data from at least one HTTP-only domain. Additionally, four extensions make HTTP XHRs to an unlimited number of domains based on URLs provided by the user; these extensions would have permanently reduced functionality. For example, *Web Developer* lets users check whether a website is valid HTML. It fetches the user-specified website with an XHR and then validates it. Under a ban on HTTP XHRs, the extension would not be able to validate HTTP websites. In total, 14% of extensions would have some functionality permanently disabled by the ban.

7.5 Recommendations

Table 6 summarizes the benefits and costs of the defenses. If the set of 100 extensions were subject to all four bans, only 5 vulnerable extensions would remain, and 16 extensions would be permanently broken. Based on this evaluation, we conclude:

- We strongly recommend banning HTTP scripts and inline scripts; together, they would prevent 47 of the 50 core extension vulnerabilities, and no extension would be permanently broken. The developer effort required to comply with these restrictions is modest.
- Banning `eval` would have a neutral effect: neither the security benefits nor the costs are large. Consequently, we advise against banning `eval`.
- We do not recommend banning HTTP XHRs, given the number of extensions that would be permanently disabled by the ban. Of the 20 vulnerabilities that the ban on HTTP XHRs would prevent, 70% could also be prevented by banning inline scripts. We do not feel that the ban on HTTP XHRs adds enough value to justify breaking 14% of extensions.

Starting with Chrome 18, extensions will be subject to a CSP that enforces some of these bans [13]. Our study partially motivated their decision to adopt the bans [5], although the policy that they adopted is slightly stricter than our recommendations. The mandatory policy in Chrome 18 will ban HTTP scripts in core extensions, inline scripts, and dynamic code generation. Due to technical limitations, they are not adopting a ban on adding HTTP scripts to HTTPS websites. The policy will remove all of the core extension vulnerabilities that we found. The only extensions that the policy will permanently break are the two extensions that rely on `eval`.

8 Related Work

Extension vulnerabilities. To our knowledge, our work is the first to evaluate the efficacy of the Google Chrome extension platform, which is widely deployed and explicitly designed to prevent and mitigate extension vulnerabilities. Vulnerabilities in other extension platforms, such as Firefox, have been investigated by previous researchers [20, 3]. We found that 40% of Google Chrome extensions are vulnerable, which is in contrast to a previous study that found that 0.24% of Firefox extensions contain vulnerabilities [3]. This does not necessarily imply that Firefox extensions are more secure; rather, our scopes and methodologies differ. Unlike the previous study, we considered network attackers as well as web attackers. We find that 5% of Google Chrome extensions have the types of web vulnerabilities that the previous study covered. The remaining discrepancy could be accounted for by our methodology: we employed expert human reviewers whereas previous work relied on a static analysis tool that does not model dynamic code evaluation, data flow through the extension API, data flow through DOM APIs, or click injection attacks.

Privilege separation. Privilege separation is a fundamental software engineering principle proposed by Saltzer and Schroeder [25]. Numerous works have applied this concept to security, such as OpenSSH [23] and qmail [6]. Recently, researchers have built several tools and frameworks to help developers privilege separate their applications [7, 11, 17, 18, 22]. Studies have established that privilege separation has value in software projects that employ security experts (e.g., browsers [9]). However, we focus on the effectiveness of privilege separation in applications that are not written by security experts.

In concurrent and independent work, Karim et al. studied the effectiveness of privilege separation in Mozilla Jetpack extensions [16]. Like Chrome extensions, Jetpack extensions are split into multiple components with different permissions. They statically analyzed Jetpack extensions and found several capability leaks in modules. Although none of these capability leaks are tied to known vulnerabilities, the capability leaks demonstrate that developers can make errors in a privilege-separated environment. Their findings support the results of our analysis of privilege separation in Chrome extensions.

Extension permissions. Previous researchers have established that permissions can reduce the privileges of extensions without negatively impacting the extensions' functionality [4, 12]. Studies have also shown that some extensions request unnecessary permissions, which is undesirable because it unnecessarily increases the scope of a potential vulnerability [12, 14]. All of these past studies asserted that the correct usage of permissions

could reduce the severity of attacks on extensions. However, they did not study whether this is true in practice or quantify the benefit for deployed applications. To our knowledge, we are the first to test whether permissions mitigate vulnerabilities in practice.

CSP compatibility. Adapting websites to work with CSP can be a challenging undertaking for developers, primarily due to the complexities associated with server-side templating languages [31]. However, extensions do not use templating languages. Consequently, applying CSP to extensions is easier than applying it to websites in most cases. We expect that our CSP compatibility findings for extensions will translate to packaged JavaScript and packaged web applications.

Malicious extensions. Extension platforms can be used to build malware (e.g., FFsniff and Infostealer.Snifula [33]). Mozilla and Google employ several strategies to prevent malicious extensions, such as domain verification, fees, and security reviews. Liu et al. propose changes to Chrome to make malware easier to identify [19]. Research on extension malware is orthogonal to our work, which focuses on external attackers that leverage vulnerabilities in benign-but-buggy extensions.

9 Conclusion

We performed a security review on a set of 100 Google Chrome extensions, including the 50 most popular, and found that 40% have at least one vulnerability. Based on this set of vulnerabilities, we evaluated the effectiveness of Chrome's three extension security mechanisms: isolated worlds, privilege separation, and permissions.

We found that the isolated worlds mechanism is highly effective because it prevents common developer errors (i.e., data-as-HTML errors). The effectiveness of isolated worlds means that privilege separation is rarely needed. Privilege separation's infrequent usefulness may not justify the complexity and communication overhead that it adds to extensions. However, our study shows that privilege separation would improve security in the absence of isolated worlds. We also found that permissions can have a significant positive impact on system security; developers of vulnerable extensions can use permissions well enough to reduce the scope of their vulnerabilities.

Although we demonstrated that privilege separation and permissions can mitigate vulnerabilities, developers do not always use them optimally. We identified several instances in which developers accidentally negated the benefits of privilege separation or intentionally circumvented the privilege separation boundary to implement features. Similarly, extensions sometimes ask for more permissions than they need [12]. Automated tools for privilege separation and permission assignment could

help developers better use these security mechanisms, thereby rendering them even more effective.

Despite the successes of these security mechanisms, extensions are widely vulnerable. The vulnerabilities occur because the system was designed to address only one threat: websites that attack extensions through direct interaction. There are no security mechanisms to prevent direct network attacks on core extensions, website metadata attacks, or attacks on websites that have been altered by extensions. This finding should serve as a reminder that multiple threats should be considered when initially designing a system. We propose to prevent these additional threats by banning insecure coding practices that commonly lead to vulnerabilities; bans on HTTP scripts and inline scripts would remove 94% of the most serious attacks with a tractable developer cost.

Acknowledgements

We would like to thank Prateek Saxena and Adam Barth for their insightful comments. This material is based upon work supported by Facebook and National Science Foundation Graduate Research Fellowships. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of Facebook or the National Science Foundation. This work is also partially supported by National Science Foundation grant CCF-0424422, a gift from Google, and the Intel Science and Technology Center for Secure Computing.

References

- [1] L. Adamski. Security severity ratings. https://wiki.mozilla.org/Security_Severity_Ratings.
- [2] B. Adida, A. Barth, and C. Jackson. Rootkits for JavaScript Environments. In *Web 2.0 Security and Privacy (W2SP)*, 2009.
- [3] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. VEX: Vetting Browser Extensions For Security Vulnerabilities. In *USENIX Security*, 2010.
- [4] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting Browsers from Extension Vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*, 2010.
- [5] Adam Barth. More secure extensions, by default. <http://blog.chromium.org/2012/02/more-secure-extensions-by-default.html>, February 2012.
- [6] D. J. Bernstein. The gmail security guarantee. <http://cr.yp.to/gmail/guarantee.html>.
- [7] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: splitting applications into reduced-privilege compartments. In *USENIX Symposium on Networked Systems Design and Implementation*, 2008.
- [8] B. Chess, Y. T. O’Neil, and J. West. JavaScript Hijacking. Technical report, Fortify, 2007.
- [9] J. Drake, P. Mehta, C. Miller, S. Moyer, R. Smith, and C. Valasek. Browser Security Comparison: A Quantitative Approach. Technical report, Accuvant Labs, 2011.
- [10] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *ACM Conference on Computer and Communication Security (CCS)*, 2011.
- [11] A. P. Felt, M. Finifter, J. Weinberger, and D. Wagner. Diesel: Applying Privilege Separation to Database Access. In *ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2011.
- [12] A. P. Felt, K. Greenwood, and D. Wagner. The Effectiveness of Application Permissions. In *USENIX Conference on Web Application Development (WebApps)*, 2011.
- [13] Google Chrome Extensions. Content Security Policy (CSP). <http://code.google.com/chrome/extensions/trunk/contentSecurityPolicy.html>.
- [14] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *IEEE Symposium on Security and Privacy*, 2011.
- [15] C. Jackson. Block chrome-extension:// pages from importing script over non-https connections. <http://code.google.com/p/chromium/issues/detail?id=29112>.
- [16] Rezwana Karim, Mohan Dhawan, Vinod Ganapathy, and Chung chiech Shan. An Analysis of the Mozilla Jetpack Extension Framework. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP)*, 2012.
- [17] A. Krishnamurthy, A. Mettler, and D. Wagner. Fine-grained privilege separation for web applications. In *International Conference on World Wide Web (WWW)*, 2010.

- [18] M. Krohn, P. Efstathopoulos, C. Frey, F. Kaashoek, E. Kohler, D. Mazières, R. Morris, M. Osborne, S. VanDeBogart, and D. Ziegler. Make Least Privilege a Right (Not a Privilege). In *Conference on Hot Topics in Operating Systems*, 2005.
- [19] L. Liu, X. Zhang, G. Yan, and S. Chen. Chrome Extensions: Threat Analysis and Countermeasures. In *Network and Distributed System Security Symposium (NDSS)*, 2012.
- [20] R. S. Liverani and N. Freeman. Abusing Firefox Extensions. Defcon17.
- [21] A. Mikhailovsky, K. V. Gavrilenko, and A. Vladimirov. The Frame of Deception: Wireless Man-in-the-Middle Attacks and Rogue Access Points Deployment. <http://www.informit.com/articles/article.aspx?p=353735&seqNum=7>, 2004.
- [22] D. Murray and S. Hand. Privilege separation made easy: trusting small libraries not big processes. In *European Workshop on System Security (EU-ROSEC)*, 2008.
- [23] N. Provos, M. Friedl, and P. Honeyman. Preventing Privilege Escalation. In *USENIX Security Symposium*, 2003.
- [24] G. Richards, C. Hammer, B. Burg, and J. Vivek. The Eval that Men Do: A Large-scale Study of the Use of Eval in JavaScript Applications. In *European Conference on Object-Oriented Programming*, 2012.
- [25] J. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. In *IEEE 63*, 1975.
- [26] R. Saltzman and A. Sharabani. Active Man in the Middle Attacks: A Security Advisory. Technical report, IBM, 2009.
- [27] StackOverflow. Why is using JavaScript eval function a bad idea? <http://stackoverflow.com/questions/86513/why-is-using-javascript-eval-function-a-bad-idea>.
- [28] B. Sterne and A. Barth. Content security policy. <https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html>.
- [29] Brandon Sterne and Adam Barth. Content security policy 1.1. <https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html>, May 2012.
- [30] S. Wagner, J. Jurgens, C. Koller, and P. Trischberger. Comparing Bug Finding Tools with Reviews and Tests. *Lecture Notes in Computer Science*, 2005.
- [31] J. Weinberger, A. Barth, and D. Song. Towards Client-side HTML Security Policies. In *Workshop on Hot Topics on Security (HotSec)*, 2011.
- [32] S. Willison. Understanding the Greasemonkey vulnerability. <http://simonwillison.net/2005/Jul/20/vulnerability/>.
- [33] C. Wuest and E. Florio. Firefox and Malware: When Browsers Attack. Technical report, Symantec, 2009.

A. List of Extensions

We selected 100 extensions from the official Chrome extension directory. We have coded extensions as follows: vulnerable and fixed ([†]), vulnerable but not fixed ([‡]), and created by Google (*). We last checked whether extensions are still vulnerable on February 7, 2012.

Most Popular Extensions

The 50 most popular extensions (and versions) that we reviewed are as follows: Adblock 2.4.6, FB Photo Zoom 1.1105.7.2, FastestChrome - Browse Faster 4.0.6[†], Adblock Plus for Google Chrome? (Beta) 1.1.3[‡], Google Translate 1.2.3.1*[‡], Google Dictionary (by Google) 3.0.0*[†], Downloads 1, Turn Off the Lights 2.0.0.7, Google Chrome to Phone Extension 2.3.0*, Firebug Lite for Google Chrome 1.3.2.9761[†], Docs PDF/PowerPoint Viewer (by Google) 3.5*, RSS Subscription Extension (by Google) 2.1.3*[‡], Webpage Screenshot 5.2[†], Mail Checker Plus for Google Mail 1.2.3.3, Awesome Screenshot: Capture & Annotate 3.0.4[‡], Google Voice (by Google) 2.2.3.4*[†], Speed Dial 2.1[†], Smooth Gestures 0.15.2, Xmarks Bookmark Sync 1.0.14, Send from Gmail (by Google) 1.12*, SocialPlus! 2.5.4[‡], FlashBlock 0.9.31, AddThis - Share & Bookmark (new) 2.1[†], WOT 1.1, Add to Amazon Wish List 1.0.0.4[†], StumbleUpon 3.5.18.1[†], Google Calendar Checker (by Google) 1.2.1*, Clip to Evernote 5.0.14.9248, Google Quick Scroll 1.8*, Stylish 0.7, Silver Bird 1.9.7.9[†], SmoothScroll 1.0.1, Browser Button for Adblock 0.0.13, TV 2.0.5, Fast YouTube Search 1.2[‡], Slideshow 1.2.9[†], bit.ly — a simple URL shortener 1.2.1.9, Web Developer 0.3.1, LastPass 1.73.2, SmileyCentral 1.0.0.3[‡], Select To Get Maps 1.1.1[‡], TooManyTabs for Chrome 1.6.5, Blog This! (by Google) 0.1.1*, TinEye Reverse Image Search 1.1, ESPN Cricinfo 1.8.3[†], MegaUpload DownloadHelper 1.2, Forecastfox 2.0.10[‡], PanicButton

0.13.1[†], AutoPager Chrome 0.6.2.12, RapidShare DownloadHelper 1.1.1.

Randomly Selected Extensions

The 50 randomly selected extensions (and versions) that we reviewed are as follows: The Independent 1.7.0.3[†], Deposit Files Download Helper 1.2, The Huffington Post 1.0.5[‡], Bookmarks Menu 3.4.6, X-notifier (Gmail, Hotmail, Yahoo, AOL ...) 0.8.2[‡], SmartVideo For YouTube 0.94, PostRank Extension 0.1.7, Bookmark Sentry 1.6.5[†], Print Plus 1.0.5.0[‡], 4chan 4chrome 9001.47[‡], HootSuite Hootlet 1.5, Cortex 1.8.3, ScribeFire 1.7[‡], Chrome Dictionary Lite 0.2.6[†], Taberareloo 2.0.17, SEO Status Pagerank/Alexa Toolbar 1.6, ChatVibes Facebook Video Chat! 1.0.7[†], PHP Console 2.1.4, Blank Canvas Script Handler 0.0.17[‡], Reddit Reveal 0.2, Greplin 1.7.3, DropBox 1.1.5, Speedtest.or.th 1, Happy Status 1.0.1[‡], New Tab Favorites 0.1, Ricks Domain Cleaner for Chrome 1.1.1, Fazedr 1.6[†], LL Bonus Comics First! 2.2, Better Reddit 0.0.4, (non-English characters) 1, turl.im url shortener 1.1, Wooword Bounce 1.2, ntust Library 0.7, me2Mini 0.0.81[‡], Back to Top 1.1, Favstar Tally by @paul_shinn 1.0.0.0, ChronoMovie 0.1.0, AutoPagerize 0.3.1, Rlweb's Bitcoin Generator 0.1, Noooooo button 1[‡], The Bass Buttons 1.95, Buttons 1.4, OpenAttribute 0.6[†], Nu.nl TV gids 1.1.3[‡], Hide Sponsored Links in Gmail? 1.4, Short URL 4, Smart Photo Viewer on Facebook 1.3.0.1[‡], Airline Checkin (mobile) 1.2102, Democracy Now! 1.1[‡], Coworkr.net Chrome 0.9.

Establishing Browser Security Guarantees through Formal Shim Verification

Dongseok Jang
UC San Diego

Zachary Tatlock
UC San Diego

Sorin Lerner
UC San Diego

Abstract

Web browsers mediate access to valuable private data in domains ranging from health care to banking. Despite this critical role, attackers routinely exploit browser vulnerabilities to exfiltrate private data and take over the underlying system. We present QUARK, a browser whose kernel has been implemented and verified in Coq. We give a specification of our kernel, show that the implementation satisfies the specification, and finally show that the specification implies several security properties, including tab non-interference, cookie integrity and confidentiality, and address bar integrity.

1 Introduction

Web browsers increasingly dominate computer use as people turn to Web applications for everything from business productivity suites and educational software to social networking and personal banking. Consequently, browsers mediate access to highly valuable, private data. Given the browser's sensitive, essential role, it should be highly secure and robust in the face of adversarial attack.

Unfortunately, security experts consistently discover vulnerabilities in all popular browsers, leading to data loss and remote exploitation. In the annual Pwn2Own competition, part of the CanSecWest security conference [4], security experts demonstrate new attacks on *up-to-date* browsers, allowing them to subvert a user's machine through the click of a single link. These vulnerabilities represent realistic, zero-day exploits and thus are quickly patched by browser vendors. Exploits are also regularly found in the wild; Google maintains a Vulnerability Reward Program, publishing its most notorious bugs and rewarding the cash to their reporters [2].

Researchers have responded to the problems of browser security with a diverse range of techniques, from novel browser architectures [10, 42, 17, 41, 31] and defenses against specific attacks [26, 20, 22, 8, 36] to al-

ternative security policies [25, 40, 21, 8, 39, 5] and improved JavaScript safety [14, 23, 38, 6, 44]. While all these techniques improve browser security, the intricate subtleties of Web security make it very difficult to know with full certainty whether a given technique works as intended. Often, a solution only “works” until an attacker finds a bug in the technique or its implementation. Even in work that attempts to provide strong guarantees (for example [17, 13, 41, 12]) the guarantees come from analyzing a model of the browser, not the actual implementation. Reasoning about such a simplified model eases the verification burden by omitting the gritty details and corner cases present in real systems. Unfortunately, attackers exploit precisely such corner cases. Thus, these approaches still leave a *formality gap* between the theory and implementation of a technique.

There is one promising technique that could minimize this formality gap: *fully formal verification of the browser implementation*, carried out in the demanding and foundational context of a mechanical proof assistant. This severe discipline forces the programmer to specify precisely how their code should behave and then provides the tools to formally guarantee that it does, all in fully formal logic, building from basic axioms up. For their trouble, the programmer is rewarded with a *machine checkable proof* that the implementation satisfies the specification. With this proof in hand, we can avoid further reasoning about the large, complex implementation, and instead consider only the substantially smaller, simpler specification. In order to believe that such a browser truly satisfies its specification, one needs only trust a very small, extensively tested proof checker. By reasoning about the actual implementation directly, we can guarantee that any security properties implied by the specification will hold in every case, on every run of the actual browser.

Unfortunately, formal verification in a proof assistant is tremendously difficult. Often, those systems which we can formally verify are severely restricted, “toy” versions

of the programs we actually have in mind. Thus, many researchers still consider full formal verification of realistic, browser-scale systems an unrealistic fantasy. Fortunately, recent advances in fully formal verification allow us to begin challenging this pessimistic outlook.

In this paper we demonstrate how *formal shim verification* radically reduces the verification burden for large systems to the degree that we were able to formally verify the implementation of a modern Web browser, QUARK, within the demanding and foundational context of the mechanical proof assistant Coq.

At its core, formal shim verification addresses the challenge of formally verifying a large system by cleverly reducing the amount of code that must be considered; instead of formalizing and reasoning about gigantic system components, all components communicate through a small, lightweight shim which ensures the components are restricted to only exhibit allowed behaviors. Formal shim verification only requires one to reason about the shim, thus eliminating the tremendously expensive or infeasible task of verifying large, complex components in a proof assistant.

Our Web browser, QUARK, exploits formal shim verification and enables us to verify security properties for a *million* lines of code while reasoning about only a *few hundred*. To achieve this goal, QUARK is structured similarly to Google Chrome [10] or OP [17]. It consists of a small browser kernel which mediates access to system resources for all other browser components. These other components run in sandboxes which only allow the component to communicate with the kernel. In this way, QUARK is able to make strong guarantees about a million lines of code (*e.g.*, the renderer, JavaScript implementation, JPEG decoders, etc.) while only using a proof assistant to reason about a few hundred lines of code (the kernel). Because the underlying system is protected from QUARK's untrusted components (*i.e.*, everything other than the kernel) we were free to adopt state-of-the-art implementations and thus QUARK is able to run popular, complex Web sites like Facebook and Gmail.

By applying formal shim verification to only reason about a small core of the browser, we formally establish the following security properties in QUARK, all within a proof assistant:

1. **Tab Non-Interference:** no tab can ever affect how the kernel interacts with another tab
2. **Cookie Confidentiality and Integrity:** cookies for a domain can only be accessed/modified by tabs of that domain
3. **Address Bar Integrity and Correctness:** the address bar cannot be modified by a tab without the

user being involved, and always displays the correct address bar.

To summarize, our contributions are as follows:

- We demonstrate how formal shim verification enabled us to formally verify the implementation of a modern Web browser. We discuss the techniques, tools, and design decisions required to formally verify QUARK in detail.
- We identify and formally prove key security properties for a realistic Web browser.
- We provide a framework that can be used to further investigate and prove more complex policies within a working, formally verified browser.

The rest of the paper is organized as follows. Section 2 provides background on browser security techniques and formal verification. Section 3 presents an overview of the QUARK browser. Section 4 details the design of the QUARK kernel and its implementation. Section 5 explains the tools and techniques we used to formally verify the implementation of the QUARK kernel. Section 6 evaluates QUARK along several dimensions while Section 7 discusses lessons learned from our endeavor.

2 Background and Related Work

This section briefly discusses both previous efforts to improve browser security and verification techniques to ensure programs behave as specified.

Browser Security As mentioned in the Introduction, there is a rich literature on techniques to improve browser security [10, 42, 17, 41, 31, 13, 12]. We distinguish ourselves from all previous techniques by verifying the actual implementation of a modern Web browser and formally proving that it satisfies our security properties, all in the context of a mechanical proof assistant. Below, we survey the most closely related work.

Previous browsers like Google Chrome [10], Gazelle [42], and OP [17] have been designed using *privilege separation* [35], where the browser is divided into components which are then limited to only those privileges they absolutely require, thus minimizing the damage an attacker can cause by exploiting any one component. We follow this design strategy.

Chrome's design compromises the principles of privilege separation for the sake of performance and compatibility. Unfortunately, its design does not protect the user's data from a compromised tab which is free to leak all cookies for every domain. Gazelle [42] adopts a more principled approach, implementing the browser

as a multi-principal OS, where the kernel has exclusive control over resource management across various Web principals. This allows Gazelle to enforce richer policies than those found in Chrome. However, neither Chrome nor Gazelle apply any formal methods to make guarantees about their browser.

The OP [17] browser goes beyond privilege separation. Its authors additionally construct a model of their browser kernel and apply the Maude model checker to ensure that this model satisfies important security properties such as the same origin policy and address bar correctness. As such, the OP browser applies insight similar to our work, in that OP focuses its formal reasoning on a small kernel. However, unlike our work, OP does not make any formal guarantees about the actual browser implementation, which means there is still a formality gap between the model and the code that runs. Our formal shim verification closes this formality gap by conducting all proofs in full formal detail using a proof assistant.

Formal Verification Recently, researchers have begun using proof assistants to fully formally verify implementations for foundational software including Operating Systems [27], Compilers [28, 1], Database Management Systems [29], Web Servers [30], and Sandboxes [32]. Some of these results have even *experimentally* been shown to drastically improve software reliability: Yang et al. [43] show through random testing that the CompCert verified C compiler is substantially more robust and reliable than its non-verified competitors like GCC and LLVM.

As researchers verify more of the software stack, the frontier is being pushed toward higher level platforms like the browser. Unfortunately, previous verification results have only been achieved at staggering cost; in the case of seL4, verification took over 13 person years of effort. Based on these results, verifying a browser-scale platform seemed truly infeasible.

Our formal verification of QUARK was radically cheaper than previous efforts. Previous efforts were tremendously expensive because researchers proved nearly every line of code correct. We avoid these costs in QUARK by applying *formal shim verification*: we structure our browser so that all our target security properties can be ensured by a very small browser kernel and then reason only about that single, tiny component. Leveraging this technique enabled us to make strong guarantees about the behavior of a million of lines of code while reasoning about only a few hundred in the mechanical proof assistant Coq.

We use the Ynot library [34] extensively to reason about imperative programming features, *e.g.*, impure functions like `fopen`, which are otherwise unavailable in Coq's pure implementation language. Ynot also provides

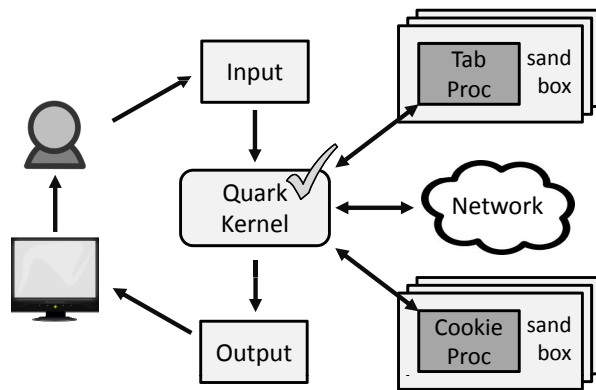


Figure 1: QUARK Architecture. This diagram shows how QUARK factors a modern browser into distinct components which run in separate processes; arrows indicate information flow. We guarantee our security properties by formally verifying the QUARK Kernel in the Coq proof assistant, which allows us to avoid reasoning about the intricate details of other components.

features which allow us to verify QUARK in a familiar style: invariants expressed as pre- and post-conditions over program states, essentially a variant of Hoare Type Theory [33]. Specifically, Ynot enables *trace-based verification*, used extensively in [30] to prove properties of servers. This technique entails reasoning about the sequence of externally visible actions a program may perform on any input, also known as *traces*. Essentially, our specification delineates which sequences of system calls the QUARK kernel can make and our verification consists of proving that the implementation is restricted to only making such sequences of system calls. We go on to formally prove that satisfying this specification implies higher level security properties like tab isolation, cookie integrity and confidentiality, and address bar integrity and correctness. Building QUARK with a different proof assistant like Isabelle/HOL would have required essentially the same approach for encoding imperative programming features, but we chose Coq since Ynot is available and has been well vetted.

Our approach is fundamentally different from previous verification tools like ESP [16], SLAM [7], BLAST [18] and Terminator [15], which work on existing code bases. In our approach, instead of trying to prove properties about a large existing code base expressed in difficult-to-reason-about languages like C or C++, we rewrite the browser inside of a theorem prover. This provides much stronger reasoning capabilities.

3 QUARK Architecture and Design

Figure 1 diagrams QUARK's architecture. Similar to Chrome [10] and OP [17], QUARK isolates complex and vulnerability-ridden components in sandboxes, forcing

them to access all sensitive resources through a small, simple browser kernel. Our kernel, written in Coq, runs in its own process and mediates access to resources including the keyboard, disk, and network. Each tab runs a modified version of WebKit in its own process. WebKit is the open source browser engine used in Chrome and Safari. It provides various callbacks for clients as Python bindings which we use to implement tabs. Since tab processes cannot directly access any system resources, we hook into these callbacks to re-route WebKit’s network, screen, and cookie access through our kernel written in Coq. QUARK also uses separate processes for displaying to the screen, storing and accessing cookies, as well reading input from the user.

Throughout the paper, we assume that an attacker can compromise any QUARK component which is exposed to content from the Internet, except for the kernel which we formally verified. This includes all tab processes, cookie processes, and the graphical output process. Thus, we provide strong formal guarantees about tab and cookie isolation, even when some processes have been completely taken over (*e.g.*, by a buffer overflow attack in the rendering or JavaScript engine of WebKit).

3.1 Graphical User Interface

The traditional GUI for Web browsers manages several key responsibilities: reading mouse and keyboard input, showing rendered graphical output, and displaying the current URL. Unfortunately, such a monolithic component cannot be made to satisfy our security goals. If compromised, such a GUI component could spoof the current URL or send arbitrary user inputs to the kernel, which, if coordinated with a compromised tab, would violate tab isolation. Thus QUARK must carefully separate GUI responsibilities to preserve our security guarantees while still providing a realistic browser.

QUARK divides GUI responsibilities into several components which the kernel orchestrates to provide a traditional GUI for the user. The most complex component displays rendered bitmaps on the screen. QUARK puts this component in a separate process to which the kernel directs rendered bitmaps from the currently selected tab. Because the kernel never reads input from this graphical output process, any vulnerabilities it may have cannot subvert the kernel or impact any other component in QUARK. Furthermore, treating the graphical output component as a separate process simplifies the kernel and proofs because it allows the kernel to employ a uniform mechanism for interacting with the outside world: messages over channels.

To formally reason about the address bar, we designed our kernel so that the current URL is written directly to the kernel’s `stdout`. This gives rise to a hybrid graphi-

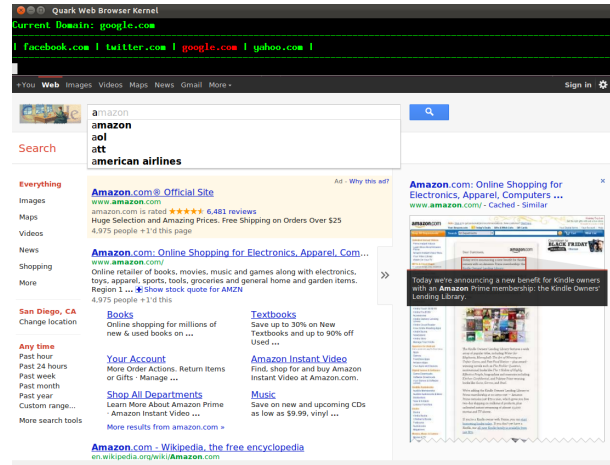


Figure 2: QUARK Screenshot. This screenshot shows QUARK running a Google search, including an interactive drop-down suggesting query completions and an initial set of search results from a JavaScript event handler dispatching an “instant search” as well as a page preview from a search result link. (Location blurred for double-blind review.)

cal/text output as shown in Figure 2 where the kernel has complete control over the address bar. With this design, the graphical output process is never able to spoof the address bar.

QUARK also uses a separate input process to support richer inputs, *e.g.*, the mouse. The input process is a simple Python script which grabs keyboard and mouse events from the user, encodes them as user input messages, and forwards them on to the kernel’s `stdin`. For keystrokes, the input process simply writes characters in ASCII format to the kernel’s `stdin`. We use several “unprintable” ASCII values (all smaller than 60 and all untypeable from the keyboard) to pass special information from the input process to the kernel. For example, the input process maps keys F1-F12 to such un-printable characters, which allows the kernel to use F11 for “new tab”, and F1-F10 for selecting tabs 1-10. Mouse clicks are also sent to the kernel through un-printable ASCII values. Because the input process only reads from the keyboard and mouse, and never from the kernel or any other QUARK components, it cannot be exposed any attacks originating from the network.

3.2 Example of Message Exchanges

To illustrate how the kernel orchestrates all the components in QUARK, we detail the steps from startup to a tab loading `http://www.google.com`. The user opens QUARK by starting the kernel which in turn starts three processes: the input process, the graphical output process, and a tab process. The kernel establishes a two-way communication channel with each process it starts. Next, the kernel then sends a

(Go "http://www.google.com") message to the tab indicating it should load the given URL (for now, assume this is normal behavior for all new tabs).

The tab process comprises our modified version of WebKit wrapped by a thin layer of Python to handle messaging with the kernel. After receiving the Go message, the Python wrapper tells WebKit to start processing http://www.google.com. Since the tab process is running in a sandbox, WebKit cannot directly access the network. When it attempts to, our Python wrapper intervenes and sends a GetURL request to the kernel. As long as the request is valid, the kernel responds with a ResDoc message containing the HTML document the tab requested.

Once the tab process has received the necessary resources from the kernel and rendered the Web pages, it sends a Display message to the kernel which contains a bitmap to display. When the kernel receives a Display message from the current tab, it forwards the message on to the graphical output process, which in turn displays the bitmap on the screen.

When the kernel reads a printable character *c* from standard input, it sends a (KeyPress *c*) message to the currently selected tab. Upon receiving such a message, the tab calls the appropriate input handler in WebKit. For example, if a user types "a" on Google, the "a" character is read by the kernel, passed to the tab, and then passed to WebKit, at which point WebKit adds the "a" character to Google's search box. This in turn causes WebKit's JavaScript engine to run an event handler that Google has installed on their search box. The event handler performs an "instant search", which initiates further communication with the QUARK kernel to access additional network resources, followed by another Display message to repaint the screen. Note that to ease verification, QUARK currently handles all requests synchronously.

3.3 Efficiency

With a few simple optimizations, we achieve performance comparable to WebKit on average (see Section 6 for measurements). Following Chrome, we adopt two optimizations critical for good graphics performance. First, QUARK uses shared memory to pass bitmaps from the tab process through the kernel to the output process, so that the Display message only passes a shared memory ID instead of a bitmap. This drastically reduces the communication cost of sending bitmaps. To prevent a malicious tab from accessing another tab's shared memory, we run each tab as a different user, and set access controls so that a tab's shared memory can only be accessed by the output process. Second, QUARK uses *rectangle-based* rendering: instead of sending a large bitmap of the entire screen each time the display changes,

the tab process determines which part of the display has changed, and sends bitmaps only for the rectangular regions that need to be updated. This drastically reduces the size of the bitmaps being transferred, and the amount of redrawing on the screen.

For I/O performance, the original Ynot library used single-character read/write routines, imposing significant overhead. We defined a new I/O library which uses size *n* reads/writes. This reduced reading an *n* byte message from *n* I/O calls to just three: reading a 1 byte tag, followed by a 4 byte payload size, and then a single read for the entire payload.

We also optimized socket connections in QUARK. Our original prototype opened a new TCP connection for each HTTP GET request, imposing significant overhead. Modern Web servers and browsers use *persistent connections* to improve the efficiency of page loading and the responsiveness of Web 2.0 applications. These connections are maintained anywhere from a few seconds to several minutes, allowing the client and server can exchange multiple request/responses on a single connection. Services like Google Chat make use of very long-lived HTTP connections to support responsive interaction with the user.

We support such persistent HTTP connections via Unix domain sockets which allow processes to send open file descriptors over channels using the sendmsg and recvmsg system calls. When a tab needs to open a socket, it sends a GetSoc message to the kernel with the host and port. If the request is valid, the kernel opens and connects the socket, and then sends an *open* socket file descriptor to the tab. Once the tab gets the socket file descriptor, it can read/write on the socket, but it cannot re-connect the socket to another host/port. In this way, the kernel controls all socket connections.

Even though we formally verify our browser kernel in a proof assistant, we were still able to implement and reason about these low-level optimizations.

3.4 Socket Security Policy

The GetSoc message brings up an interesting security issue. If the kernel satisfied all GetSoc requests, then a compromised tab could open sockets to any server and exchange arbitrary amounts of information. The kernel must prevent this scenario by restricting socket connections.

To implement this restriction, we introduce the idea of a *domain suffix* for a tab which the user enters when the tab starts. A tab's domain suffix controls several security features in QUARK, including which socket connections are allowed and how cookies are handled (see Section 3.5). In fact, our address bar, located at the very top of the browser (see Figure 2), displays the domain suffix, not just the tab's URL. We therefore refer to it as

the “domain bar”.

For simplicity, our current domain suffixes build on the notion of a *public suffix*, which is a top-level domain under which Internet users can directly register names, for example `.com`, `.co.uk`, or `.edu` – Mozilla maintains an exhaustive list of such suffixes [3]. In particular, we require the domain suffix for a tab to be exactly one level down from a public suffix, e.g., `google.com`, `amazon.com`, etc. In the current QUARK prototype the user provides a tab’s domain suffix separately from its initial URL, but one could easily compute the former from the later. Note that, once set, a tab’s domain suffix never changes. In particular, any frames a tab loads do not affect its domain suffix.

We considered using the tab’s origin (which includes the URL, scheme, and port) to restrict socket creation, but such a policy is too restrictive for many useful sites. For example, a single GMail tab uses frames from domains such as `static.google.com` and `mail.google.com`. However, our actual domain suffix checks are modularized within QUARK, which will allow us to experiment with finer grained policies in future work.

To enforce our current socket creation policy, we first define a subdomain relation \leq as follows: given domain d_1 and domain suffix d_2 , we use $d_1 \leq d_2$ to denote that d_1 is a subdomain of d_2 . For example `www.google.com` \leq `google.com`. If a tab with domain suffix t requests to open a connection to a host h , then the kernel allows the connection if $h \leq t$. To load URLs that are not a subdomain of the tab suffix, the tab must send a `GetURL` message to the kernel – in response, the kernel does not open a socket but, if the request is valid, may provide the content of the URL. Since the kernel does not attach any cookies to the HTTP request for a `GetURL` message, a tab can only access publicly available data using `GetURL`. In addition, `GetURL` requests only provide the response body, not HTTP headers.

Note that an exploited tab could leak cookies by encoding information within the URL parameter of `GetURL` requests, but only cookies for that tab’s domain could be leaked. Because we do not provide any access to HTTP headers with `GetURL`, we consider this use of `GetURL` to leak cookies analogous to leaking cookie data over timing channels.

Although we elide details in the current work, we also slightly enhanced our socket policy to improve performance. Sites with large data sets often use content distribution networks whose domains will not satisfy our domain suffix checks. For example `facebook.com` uses `fbcdn.net` to load much of its data. Unfortunately, the simple socket policy described above will force all this data to be loaded using slow `GetURL` requests through the kernel. To address this issue, we associate *whitelists* with the most popular sites so that tabs for those do-

main can open sockets to the associated content distribution network. The tab domain suffix remains a single string, e.g. `facebook.com`, but behind the scenes, it gets expanded into a list depending on the domain, e.g., [`facebook.com`, `fbcdn.net`]. When deciding whether to satisfy a given socket request, QUARK considers this list as a disjunction of allowed domain suffixes. Currently, we provide these whitelists manually.

3.5 Cookies and Cookie Policy

QUARK maintains a set of cookie processes to handle cookie accesses from tabs. This set of cookie processes will contain a cookie process for domain suffix S if S is the domain suffix of a running tab. By restricting messages to and from cookie processes, the QUARK kernel guarantees that browser components will only be able to access cookies appropriate for their domain.

The kernel receives cookie store/retrieve requests from tabs and directs the requests to the appropriate cookie process. If a tab with domain suffix t asks to store a cookie with domain c , then our kernel allows the operation if $c \leq t$, in which case it sends the store request to the cookie process for domain t . Similarly, if a tab with domain suffix t wants to retrieve a cookie for domain c , then our kernel allows the operation if $c \leq t$, in which case it sends the request to the cookie process for domain t and forwards any response to the requesting tab.

The above policy prevents cross-domain cookie reads from a compromised tab, and it prevents a compromised cookie process from leaking information about its cookies to another domain; yet it also allows different tabs with the same domain suffix (but different URLs) to communicate through cookies (for example, `mail.google.com` and `calendar.google.com`).

3.6 Security Properties of QUARK

We provide intuitive descriptions of the security properties we proved for QUARK’s kernel; formal definitions appear later in Section 4. A tab in the kernel is a pair, containing the tab’s domain suffix as a string and the tab’s communication channel as a file descriptor. A cookie process is also a pair, containing the domain suffix that this cookie process manages and its communication channel. We define the state of the kernel as the currently selected tab, the list of tabs, and the list of cookie processes. Note that the kernel state only contains strings and file descriptors.

We prove the following main theorems in Coq:

1. **Response Integrity:** The way the kernel responds to any request only depends on past user “control keys” (namely keys F1-F12). This ensures that one

browser component (*e.g.*, a tab or cookie process) can never influence how the kernel responds to another component, and that the kernel never allows untrusted input (*e.g.*, data from the web) to influence how the kernel responds to a request.

2. **Tab Non-Interference:** The kernel’s response to a tab’s request is the same no matter how other tabs interact with the kernel. This ensures that the kernel never provides a direct way for one tab to attack another tab or steal private information from another tab.
3. **No Cross-domain Socket Creation:** The kernel disallows any cross-domain socket creation (as described in Section 3.4).
4. **Cookie Integrity/Confidentiality:** The kernel disallows any cross-domain cookie stores or retrieves (as described in Section 3.5).
5. **Domain Bar Integrity and Correctness:** The domain bar cannot be compromised by a tab, and is always equal to the domain suffix of the currently selected tab.

4 Kernel Implementation in Coq

QUARK’s most distinguishing feature is its kernel, which is implemented and proved correct in Coq. In this section we present the implementation of the main kernel loop. In the next section we explain how we formally verified the kernel.

Coq enables users to write programs in a small, simple functional language and then reason formally about them using a powerful logic, the Calculus of Constructions. This language is essentially an effect-free (pure) subset of popular functional languages like ML or Haskell with the additional restriction that programs must always terminate. Unfortunately, these limitations make Coq’s default implementation language ill-suited for writing system programs like servers or browsers which must be effectful to perform I/O and by design may not terminate.

To address the limitations of Coq’s implementation language, we use Ynot [34]. Ynot is a Coq library which provides monadic types that allow us to write effectful, non-terminating programs in Coq while retaining the strong guarantees and reasoning capabilities Coq normally provides. Equipped with Ynot, we can write our browser kernel in a fairly straightforward style whose essence is shown in Figure 3.

Single Step of Kernel. QUARK’s kernel is essentially a loop that continuously responds to requests from the user or tabs. In each iteration, the kernel calls `kstep`

```

Definition kstep(ctab, ctabs) :=
  chan <- iselect(stdin, ctabs);
  match chan with
  | Stdin =>
    c <- read(stdin);
    match c with
    | "+" =>
      t <- mktab();
      write_msg(t, Render);
      return (t, t::ctabs)
    | ...
    end
  | Tab t =>
    msg <- read_msg(t);
    match msg with
    | GetSoc(host, port) =>
      if (safe_soc(host, domain_suffix(t))) then
        send_soc(t, host, port);
        return (ctab, ctabs)
      else
        write_msg(t, Error);
        return (ctab, ctabs)
    | ...
    end
  end
end

```

Figure 3: *Body for Main Kernel Loop.* This Coq code shows how our QUARK kernel receives and responds to requests from other browser components. It first uses a Unix-style select to choose a ready input channel, reads a request from that channel, and responds to the message appropriately. For example, if the user enters “+”, the kernel creates a new tab and sends it the `Render` message. In each case, the code returns the new kernel state resulting from handling this request.

which takes the current kernel state, handles a single request, and returns the new kernel state as shown in Figure 3. The kernel state is a tuple of the current tab (`ctab`), the list of tabs (`ctabs`), and a few other components which we omit here (*e.g.*, the list of cookie processes). For details regarding the loop and kernel initialization code please see [24].

`kstep` starts by calling `iselect` (the “i” stands for input) which performs a Unix-style select over `stdin` and all tab input channels, returning `Stdin` if `stdin` is ready for reading or `Tab t` if the input channel of tab `t` is ready. `iselect` is implemented in Coq using a `select` primitive which is ultimately just a thin wrapper over the Unix `select` system call. The Coq extraction process, which converts Coq into OCaml for execution, can be customized to link our Coq code with OCaml implementations of primitives like `select`. Thus `select` is exposed to Coq essentially as a primitive of the appropriate monadic type. We have similar primitives for reading/writing on channels, and opening sockets.

Request from User. If `stdin` is ready for reading, the kernel reads one character `c` using the `read` primitive, and then responds based on the value of `c`. If `c` is “+”, the kernel adds a new tab to the browser. To achieve this, it first calls `mktab` to start a tab process (another

primitive implemented in OCaml). `mktab` returns a tab object, which contains an input and output channels to communicate with the tab process. Once the tab `t` is created, the kernel sends it a `Render` message using the `write_msg` function – this tells `t` to render itself, which will later cause the tab to send a `Display` message to the kernel. Finally, we return an updated kernel state $(\tau, \tau::\text{tabs})$, which sets the newly created tab `t` as the current tab, and adds `t` to the list of tabs.

In addition to “+” the kernel handles several other cases for user input, which we omit in Figure 3. For example, when the kernel reads keys F1 through F10, it switches to tabs 1 through 10, respectively, if the tab exists. To switch tabs, the kernel updates the currently selected tab and sends it a `Render` message. The kernel also processes mouse events delivered by the input process to the kernel’s `stdin`. For now, we only handle mouse clicks, which are delivered by the input process using a single un-printable ASCII character (adding richer mouse events would not fundamentally change our kernel or proofs). The kernel in this case calls a primitive implemented in OCaml which gets the location of the mouse, and it sends a `MouseClicked` message using the returned coordinates to the currently selected tab. We use this two-step approach for mouse clicks (un-printable character from the input process, followed by primitive in OCaml), so that the kernel only needs to process a single character at a time from `stdin`, which simplifies the kernel and proofs.

Request from Tab. If a tab `t` is ready for reading, the kernel reads a message `m` from the tab using `read_msg`, and then sends a response which depends on the message. If the message is `GetSoc(host, port)`, then the tab is requesting that a socket be opened to the given `host/port`. We apply the socket policy described in Section 3.4, where `domain_suffix t` returns the domain suffix of a tab `t`, and `safe_soc(host, domsuf)` applies the policy (which basically checks that `host` is a sub-domain of `domsuf`). If the policy allows the socket to be opened, the kernel uses the `send_socket` to open a socket to the host, and send the socket over the channel to the tab (recall that we use Unix domain sockets to send open file descriptors from one process to another). Otherwise, it returns an `Error` message.

In addition to `GetSoc` the kernel handles several other cases for tab requests, which we omit in Figure 3. For example, the kernel responds to `GetURL` by retrieving a URL and returning the result. It responds to cookie store and retrieve messages by checking the security policy from Section 3.5 and forwarding the message to the appropriate cookie process (note that for simplicity, we did not show the cookie processes in Figure 3). The kernel also responds to cookie processes that are sending cookie results back to a tab, by forwarding the cookie results

to the appropriate tab. The kernel responds to `Display` messages by forwarding them to the output process.

Monads in Ynot. The code in Figure 3 shows how Ynot supports an imperative programming style in Coq. This is achieved via *monads* which allow one to encode effectful, non-terminating computations in pure languages like Haskell or Coq. Here we briefly show how monads enable this encoding. In the next section we extend our discussion to show how Ynot’s monads also enable reasoning about the kernel using pre- and post-conditions as in Hoare logic.

We use Ynot’s `ST` monad which is a parameterized type where `ST T` denotes computations which may perform some I/O and then return a value of type `T`. To use `ST`, Ynot provides a `bind` primitive which has the following dependent type:

```
bind : forall T1 T2,
      ST T1 -> (T1 -> ST T2) -> ST T2
```

This type indicates that, for any types `T1` and `T2`, `bind` will take two parameters: (1) a monad of type `ST T1` and (2) a function that takes a value of type `T1` and returns a monad of type `ST T2`; then `bind` will produce a value in the `ST T2` monad. The type parameters `T1` and `T2` are inferred automatically by Coq. Thus, the expression `bind X Y` returns a monad which represents the computation: run `X` to get a value `v`; run `(Y v)` to get a value `v'`; return `v'`.

To make using `bind` more convenient, Ynot also defines Haskell-style “do” syntactic sugar using Coq’s Notation mechanism, so that `x <- a; b` is translated to `bind a (fun x => b)`, and `a; b` is translated to `bind a (fun _ => b)`. Finally, the Ynot library provides a `return` primitive of type `forall T (v: T), ST T` (where again `T` is inferred by Coq). Given a value `v`, the monad `return v` represents the computation that does no I/O and simply returns `v`.

5 Kernel Verification

In this section we explain how we verified QUARK’s kernel. First, we specify correct behavior of the kernel in terms of *traces*. Second, we prove the kernel satisfies this specification using the full power of Ynot’s monads. Finally, we prove that our kernel specification implies our target security properties.

5.1 Actions and Traces

We verify our kernel by reasoning about the sequences of calls to primitives (*i.e.*, system calls) it can make. We call such a sequence a *trace*; our kernel specification (henceforth “spec”) defines which traces are allowed for a correct implementation as in [30].

```

Definition Trace := list Action.

Inductive Action :=
| ReadN   : chan -> positive -> list ascii -> Action
| WriteN  : chan -> positive -> list ascii -> Action
| MkTab   : tab -> Action
| SentSoc : tab -> list ascii -> list ascii -> Action
| ...

```

```

Definition Read c b :=
  ReadN c 1 [c]

```

Figure 4: *Traces and Actions*. This Coq code defines the type of externally visible actions our kernel can take. A *trace* is simply a list of such actions. We reason about our kernel by proving properties of the traces it can have. Traces are like other Coq values; in particular, we can write functions that return traces. `Read` is a helper function to construct a trace fragment corresponding to reading a single byte.

We use a list of *actions* to represent the trace the kernel produces by calling primitives. Each action in a trace corresponds to the kernel invoking a particular primitive. Figure 4 shows a partial definition of the `Action` datatype. For example: `ReadN f n l` is an `Action` indicating that the `n` bytes in list `l` were read from input channel `f`; `MkTab t` indicates that `tab t` was created; `SentSoc t host port` indicates a socket was connected to `host/port` and passed to `tab t`.

We can manipulate traces and `Actions` like any other values in Coq. For example, we can define a function `Read c b` to encode the special case that a single byte `b` was read on input channel `c`. Though not shown here, we also define similar helper functions to build up trace fragments which correspond to having read or written a particular message to a given component. For example, `ReadMsg t (GetSoc host port)` corresponds to the trace fragment that results from reading a `GetSoc` request from `tab t`.

5.2 Kernel Specification

Figure 5 shows a simplified snippet of our kernel spec. The spec is a predicate `tcorrect` over traces with two constructors, stating the two ways in which `tcorrect` can be established: (1) `tcorrect_nil` states that the empty trace satisfies `tcorrect` (2) `tcorrect_step` states that if `tr` satisfies `tcorrect` and the kernel takes a single step, meaning that after `tr` it gets a request `req`, and responds with `rsp`, then the trace `rsp ++ req ++ tr` (where `++` is list concatenation) also satisfies `tcorrect`. By convention the first action in a trace is the most recent.

The predicate `step_correct` defines correctness for a single iteration of the kernel’s main loop: `step_correct tr req rsp` holds if given the past trace `tr` and a request `req`, the response of the kernel should be `rsp`. The predicate has several constructors (not all shown) enumerating the ways

```

Inductive tcorrect : Trace -> Prop :=
| tcorrect_nil:
  tcorrect nil
| tcorrect_step: forall tr req rsp,
  tcorrect tr ->
  step_correct tr req rsp ->
  tcorrect (rsp ++ req ++ tr).

```

```

Inductive step_correct :
  Trace -> Trace -> Trace -> Prop :=
| step_correct_add_tab: forall tr t,
  step_correct tr
  (MkTab t :: Read stdin "+" :: nil)
  (WroteMsg t Render)
| step_correct_socket_true: forall tr t host port,
  is_safe_soc host (domain_suffix t) = true ->
  step_correct tr
  (ReadMsg t (GetSoc host port))
  (SentSoc t host port)
| step_correct_socket_false: forall tr t host port,
  is_safe_soc host (domain_suffix t) <> true ->
  step_correct tr
  (ReadMsg t (GetSoc host port) ++ tr)
  (WroteMsg t Error ++ tr)
| ...

```

Figure 5: *Kernel Specification*. `step_correct` is a predicate over triples containing a past trace, a request trace, and a response trace; it holds when the response is valid for the given request in the context of the past trace. `tcorrect` defines a correct trace for our kernel to be a sequence of correct steps, *i.e.*, the concatenation of valid request and response trace fragments.

`step_correct` can be established. For example, `step_correct_add_tab` states that typing “+” on `stdin` leads to the creation of a `tab` and sending the `Render` message. The `step_correct_socket_true` case captures the successful socket creation case, whereas `step_correct_socket_false` captures the error case.

5.3 Monads in Ynot Revisited

In the previous section, we explained Ynot’s `ST` monad as being parameterized over a single type `T`. In reality, `ST` takes two additional parameters representing pre- and post-conditions for the computation encoded by the monad. Thus, `ST T P Q` represents a computation which, if started in a state where `P` holds, may perform some I/O and then return a value of type `T` in a state where `Q` holds. For technical reasons, these pre- and post-conditions are expressed using separation logic, but we defer details to a tech report [24].

Following the approach of Malecha et al. [30], we define an *opaque* predicate (`traced tr`) to represent the fact that at a given point during execution, `tr` captures all the past activities; and (`open f`) to represent the fact that channel `f` is currently open. An *opaque* predicate cannot be proven directly. This property allows us to ensure that no part of the kernel can forge a proof of (`traced tr`) for any trace it independently constructs.

```

Axiom readn:
forall (f: chan) (n: positive) {tr: Trace},
ST (list ascii)
{traced tr * open f}
{fun l =>
  traced (ReadN f n l :: tr) *
  [len l = n] * open f }.

Definition read_msg:
forall (t: tab) {tr: Trace},
ST msg
{traced tr * open (tchan t)}
{fun m =>
  traced (ReadMsg t m ++ tr) * open (tchan t)} :=
...

```

Figure 6: *Example Monadic Types*. This Coq code shows the monadic types for the `readn` primitive and for the `read_msg` function which is implemented in terms of `readn`. In both cases, the first expression between curly braces represents a pre-condition and the second represents a post-condition. The asterisk (*) may be read as normal conjunction in this context.

Thus `(traced tr)` can only be true for the current trace `tr`.

Figure 6 shows the full monadic type for the `readn` primitive, which reads `n` bytes of data and returns it. The `*` connective represents the separating conjunction from separation logic. For our purposes, consider it as a regular conjunction. The precondition of `(readn f n tr)` states that `tr` is the current trace and that `f` is open. The post-condition states that the trace after `readn` will be the same as the original, but with an additional `(ReadN f n l)` action at the beginning, where the length of `l` is equal to `n` (`len l = n` is a regular predicate, which is lifted using square brackets into a separation logic predicate). After the call, the channel `f` is still open.

The full type of the `Ynot` bind operation makes sure that when two monads are sequenced, the post-condition of the first monad implies the pre-condition of the second. This is achieved by having `bind` take an additional third argument, which is a proof of this implication. The syntactic sugar for `x <- a; b` is updated to pass the wildcard “_” for the additional argument. When processing the definition of our kernel, Coq will enter into an interactive mode that allows the user to construct proofs to fill in these wildcards. This allows us to prove that the post-condition of each monad implies the pre-condition of the immediately following monad in Coq’s interactive proof environment.

5.4 Back to the Kernel

We now return to our kernel from Figure 3 and show how we prove that it satisfies the spec from Figure 5. We augment the kernel state to additionally include the trace of the kernel so far, and we update our kernel code to maintain this `tr` field. By using a special encoding in

`Ynot` for this trace, the `tr` field is not realized at runtime, it is only used for proof purposes.

We define the `kcorrect` predicate as follows (`s.tr` projects the current trace out of kernel state `s`):

```

Definition kcorrect (s: kstate) :=
  traced s.tr * [tcorrect s.tr]

```

Now we want to show that `kcorrect` is an invariant that holds throughout execution of the kernel. Essentially we must show that `(kcorrect s)` is a loop invariant on the kernel state `s` for the main kernel loop, which boils down to showing that `(kcorrect s)` is valid as both the pre- and post-condition for the loop body, `kstep` as shown in Figure 3.

As mentioned previously, Coq will ask us to prove implications between the post-condition of one monad and the pre-condition of the next. While these proofs are ultimately spelled out in full formal detail, Coq provides facilities to automate a substantial portion of the proof process. `Ynot` further provides a handful of sophisticated tactics which helped automatically dispatch tedious, repeatedly occurring proof obligations. We had to manually prove the cases which were not handled automatically. While we have only shown the key kernel invariant here, in the full implementation there are many additional Hoare predicates for the intermediate goals between program points. We defer details of these predicates and the manual proof process to [24], but discuss proof effort in Section 6.

5.5 Security Properties

Our security properties are phrased as theorems about the spec. We now prove that our spec implies these key security properties, which we intend to hold in QUARK. Figure 7 shows these key theorems, which correspond precisely to the security properties outlined in Section 3.6.

State Integrity. The first security property, `kstate_dep_user`, ensures that the kernel state only changes in response to the user pressing a “control key” (e.g. switching to the third tab by pressing F3). The theorem establishes this property by showing its contrapositive: if the kernel steps by responding with `rsp` to request `req` after trace `tr` and no “control keys” were read from the user, then the kernel state remains unchanged by this step. The function `proj_user_control`, not shown here, simply projects from the trace all actions of the form `(Read c stdin)` where `c` is a control key. The function `kernel_state`, not shown here, just computes the kernel state from a trace. We also prove that at the beginning of any invocation to `kloop` in Figure 3, all fields of `s` aside from `tr` are equal to the corresponding field in `(kernel_state s.tr)`.

Response Integrity. The second security property, `kresponse_dep_kstate`, ensures that every kernel re-

```

Theorem kstate_dep_user:
  forall tr req rsp,
    step_correct tr req rsp ->
      proj_user_control tr
        = proj_user_control (rsp ++ req ++ tr) ->
          kernel_state tr = kernel_state (rsp ++ req ++ tr).

Theorem kresponse_dep_kstate:
  forall tr1 tr2 req rsp,
    kernel_state tr1 = kernel_state tr2 ->
      step_correct tr1 req rsp ->
        step_correct tr2 req rsp.

Theorem tab_NI:
  forall tr1 tr2 t req rsp1 rsp2,
    tcorrect tr1 -> tcorrect tr2 ->
      from_tab t req ->
        (cur_tab tr1 = Some t <-> cur_tab tr2 = Some t) ->
          step_correct tr1 req rsp1 ->
            step_correct tr2 req rsp2 ->
              rsp1 = rsp2 \ /
                (exists m, rsp1 = WroteCMsg (cproc_for t tr1) m /\
                  rsp2 = WroteCMsg (cproc_for t tr2) m).

Theorem no_xdom_sockets: forall tr t,
  tcorrect tr ->
  In (SendSocket t host s) tr ->
  is_safe_soc host (domain_suffic t).

Theorem no_xdom_cookie_set: forall tr1 tr2 cproc,
  tcorrect (tr1 ++ SetCookie key value cproc :: tr2) ->
  exists tr t,
    (tr2 = (SetCookieRequest t key value :: tr) /\
      is_safe_cook (domain cproc) (domain_suffix t))

Theorem dom_bar_correct: forall tr,
  tcorrect tr -> dom_bar tr = domain_suffix (cur_tab tr).

```

Figure 7: *Kernel Security Properties.* This Coq code shows how traces allow us to formalize QUARK’s security properties.

sponse depends solely on the request and the kernel state. This delineates which parts of a trace can affect the kernel’s behavior: for a given request `req`, the kernel will produce the same response `rsp`, for any two traces that *induce the same kernel state*, even if the two traces have completely different sets of requests/responses (recall that the kernel state only includes the current tab and the set of tabs, and most request responses don’t change these). Since the kernel state depends only the user’s control key inputs, this theorem immediately establishes the fact that *our browser will never allow one component to influence how the kernel treats another component unless the user intervenes*.

Note that `kresponse_dep_kstate` shows that the kernel will produce the same response given the same request after any two traces that induce the same kernel state. This may seem surprising since many of the kernel’s operations produce nondeterministic results, *e.g.*, there is no way to guarantee that two web fetches of the same URL will produce the same document. However, such nondeterminism is captured in the request, which

is consistent with our notion of requests as inputs and responses as outputs.

Tab Non-Interference. The second security property, `tab_NI`, states that the kernel’s response to a tab is not affected by any other tab. In particular, `tab_NI` shows that if in the context of a valid trace, `tr1`, the kernel responds to a request `req` from tab `t` with `rsp1`, then the kernel will respond to the same request `req` with an equivalent response in the context of any other valid trace `tr2` which also contains tab `t`, irrespective of what other tabs are present in `tr2` or what actions they take. Note that this property holds in particular for the case where trace `tr2` contains *only* tab `t`, which leads to the following corollary: the kernel’s response to a tab will be the same even if all other tabs did not exist

The formal statement of the theorem in Figure 7 is made slightly more complicated because of two issues. First, we must assume that the focused tab at the end of `tr1` (denoted by `cur_tab tr1`) is `t` if and only if the focused tab at the end of `tr2` is also `t`. This additional assumption is needed because the kernel responds differently based on whether a tab is focused or not. For example, when the kernel receives a `Display` message from a tab (indicating that the tab wants to display its rendered page to the user), the kernel only forwards the message to the output process if the tab is currently focused.

The second complication is that the communication channel underlying the cookie process for `t`’s domain may not be the same between `tr1` and `tr2`. Thus, in the case that kernel responds by forwarding a valid request from `t` to its cookie process, we guarantee that the kernel sends the same payload to the cookie process corresponding to `t`’s domain.

Note that, unlike `kresponse_dep_kstate`, `tab_NI` does not require `tr1` and `tr2` to induce the same kernel state. Instead, it merely requires the request `req` to be from a tab `t`, and `tr1` and `tr2` to be valid traces that both contain `t` (indeed, `t` must be on both traces otherwise the `step_correct` assumptions would not hold). Other than these restrictions, `tr1` and `tr2` may be arbitrarily different. They could contain different tabs from different domains, have different tabs focused, different cookie processes, etc.

Response Integrity and Tab Non-Interference provide different, complimentary guarantees. Response Integrity ensures the response to *any* request `req` is only affected by control keys and `req`, while Tab Non-Interference guarantees that the response to a tab request does not leak information to another tab. Note that Response Integrity could still hold for a kernel which mistakenly sends responses to the wrong tab, but Tab Non-Interference prevents this. Similarly, Tab Non-Interference could hold for a kernel which allows a tab to affect how the kernel responds to a cookie process, but Response Integrity pre-

cludes such behavior.

It is also important to understand that `tab_NI` proves the absence of interference as caused by the *kernel*, not by other components, such as the network or cookie processes. In particular, it is still possible for two websites to communicate with each other through the network, causing one tab to affect another tab’s view of the web. Similarly, it is possible for one tab to set a cookie which is read by another tab, which again causes a tab to affect another one. For the cookie case, however, we have a separate theorem about cookie integrity and confidentiality which states that cookie access control is done correctly.

Note that this property is an adaptation of the traditional non-interference property. In traditional non-interference, the program has “high” and “low” inputs and outputs; a program is non-interfering if high inputs never affect low outputs. Intuitively, this constrains the program to never reveal secret information to untrusted principles.

We found that this traditional approach to non-interference fits poorly with our trace-based verification approach. In particular, because the browser is a non-terminating, reactive program, the “inputs” and “outputs” are infinite streams of data.

Previous research [11] has adapted the notion of non-interference to the setting of reactive programs like browsers. They provide a formal definition of non-interference in terms of possibly infinite input and output streams. A program at a particular state is non-interfering if it produces *similar* outputs from *similar* inputs. The notion of similarity is parameterized in their definition; they explore several options and examine the consequences of each definition for similarity.

Our tab non-interference theorem can be viewed in terms of the definition from [11], where requests are “inputs” and responses are “outputs”; essentially, our theorem shows the inductive case for potentially infinite streams. Adapting our definition to fit directly in the framework from [11] is complicated by the fact that we deal with a unified trace of input and output events in the sequence they occur instead of having one trace of input events and a separate trace of output events. In future work, we hope to refine our notion of non-interference to be between domains instead of tabs, and we believe that applying the formalism from [11] will be useful in achieving this goal. Unlike [11], we prove a version of non-interference for a particular program, the QUARK browser kernel, directly in Coq.

No Cross-domain Socket Creation. The third security property, `no_xdom_sockets`, ensures that the kernel never delivers a socket bound to domain *d* to a tab whose domain does not match *d*. This involves checking URL suffixes in a style very similar to the cookie policy as discussed earlier. This property forces a tab to

Component	Language	Lines of code
Kernel Code	Coq	859
Kernel Security Properties	Coq	142
Kernel Proofs	Coq	4,383
Kernel Primitive Specification	Coq	143
Kernel Primitives	Ocaml/C	538
Tab Process	Python	229
Input Process	Python	60
Output Process	Python	83
Cookie Process	Python	135
Python Message Lib	Python	334
WebKit Modifications	C	250
WebKit	C/C++	969,109

Figure 8: QUARK Components by Language and Size.

use `GetURL` when accessing websites that do not match its domain suffix, thus restricting the tab to only access publicly available data from other domains.

Cookie Integrity/Confidentiality. The fourth security property states cookie integrity and confidentiality. As an example of how cookies are processed, consider the following trace when a cookie is set:

```
SetCookie key value cproc ::
SetCookieRequest tab key value :: ...
```

First, the `SetCookieRequest` action occurs, stating that a given tab just requested a cookie (in fact, `SetCookieRequest` is just defined in terms of a `ReadMsg` action of the appropriate message). The kernel responds with a `SetCookie` action (defined in terms of `WroteMsg`), which represents the fact that the kernel sent the cookie to the cookie process `cproc`. The kernel implementation is meant to find a `cproc` whose domain suffix corresponds to the tab. This requirement is given in the theorem `no_xdom_cookie_set`, which encodes cookie integrity. It requires that, within a correct trace, if a cookie process is ever asked to set a cookie, then it is in immediate response to a cookie set request for the same exact cookie from a tab whose domain matches that of the cookie process. There is a similar theorem `no_xdom_cookie_get`, not shown here, which encodes cookie confidentiality.

Domain Bar Integrity and Correctness. The fifth property states that the domain bar is equal to the domain suffix of the currently selected tab, which encodes the correctness of the address bar.

6 Evaluation

In this section we evaluate QUARK in terms of proof effort, trusted computing base, performance, and security.

Proof Effort and Component Sizes. QUARK comprises several components written in various languages; we summarize their sizes in Figure 8. All Python components share the “Python Message Lib” for messaging

with the kernel. Implementing QUARK took about 6 person months, which includes several iterations redesigning the kernel, proofs, and interfaces between components. Formal shim verification saved substantial effort: we guaranteed our security properties for a million lines of code by reasoning just 859.

Trusted Computing Base. The trusted computing base (TCB) consists of all system components we assume to be correct. A bug in the TCB could invalidate our security guarantees. QUARK’s TCB includes:

- Coq’s core calculus and type checker
- Our formal statement of the security properties
- Several primitives used in Ynot
- Several primitives unique to QUARK
- The Ocaml compiler and runtime
- The underlying Operating System kernel
- Our chroot sandbox

Because Coq exploits the Curry-Howard Isomorphism, its type checker is actually the “proof checker” we have mentioned throughout the paper. We assume that our formal statement of the security properties correctly reflects how we understand them intuitively. We also assume that the primitives from Ynot and those we added in QUARK correctly implement the monadic type they are axiomatically assigned. We trust the OCaml compiler and runtime since our kernel is extracted from Coq and run as an OCaml program. We also trust the operating system kernel and our traditional chroot sandbox to provide process isolation, specifically, our design assumes the sandboxing mechanism restricts tabs to only access resources provided by the kernel, thus preventing compromised tabs from commuting over covert channels.

Our TCB does *not* include WebKit’s large code base or the Python implementation. This is because a compromised tab or cookie process can not affect the security guarantees provided by kernel. Furthermore, the TCB does not include the browser kernel code, since it has been proved correct.

Ideally, QUARK will take advantage of previous formally verified infrastructure to minimize its TCB. For example, by running QUARK in seL4 [27], compiling QUARK’s ML-like browser kernel with the MLCompCert compiler [1], and sandboxing other QUARK components with RockSalt [32], we could drastically reduce our TCB by eliminating its largest components. In this light, our work shows how to build yet another piece of the puzzle (namely a verified browser) needed to for a fully verified software stack. However, these other verified building blocks are themselves research prototypes which, for now, makes them very difficult to stitch together as a foundation for a realistic browser.

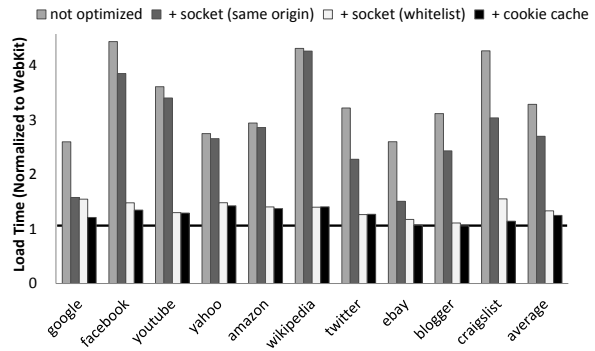


Figure 9: QUARK Performance. This graph shows QUARK load times for the Alexa Top 10 Web sites, normalized to stock WebKit’s load times. In each group, the leftmost bar shows the unoptimized load time, the rightmost bar shows the load time in the final, optimized version of QUARK, and intermediate bars show how additional optimizations improve performance. Smaller is better.

Performance. We evaluate our approach’s performance impact by comparing QUARK’s load times to stock WebKit. Figure 9 shows QUARK load times for the top 10 Alexa Web sites, normalized to stock WebKit. QUARK’s overhead is due to factoring the browser into distinct components which run in separate processes and explicitly communicate through a formally verified browser kernel.

By performing a few simple optimizations, the final version of QUARK loads large, sophisticated websites with only 24% overhead. This is a substantial improvement over a naïve implementation of our architecture, shown by the left-most “not-optimized” bars in Figure 9. Passing bound sockets to tabs, whitelisting content distribution networks for major websites, and caching cookie accesses, improves performance by 62% on average.

The WebKit baseline in Figure 9 is a full-featured browser based on the Python bindings to WebKit. These bindings are simply a thin layer around WebKit’s C/C++ implementation which provide easy access to key callbacks. We measure 10 loads of each page and take the average. Over all 10 sites, the average slowdown in load-time is 24% (with a minimum of 5% for blogger and a maximum of 42% for yahoo).

We also measured load-time for the previous version of QUARK, just before rectangle-based rendering was added. In this previous version, the average load-time was only 12% versus 24% for the current version. The increase in overhead is due to additional communication with the kernel during incremental rendering. Despite this additional overhead in load time, incremental rendering is preferable because it allows QUARK to display content to the user as it becomes available instead of waiting until an entire page is loaded.

Security Analysis. QUARK provides strong, formal guarantees for security policies which are not fully compatible with traditional web security policies, but still

provide some of the assurances popular web browsers seek to provide.

For the policies we have not formally verified, QUARK offers exactly the same level of traditional, unverified enforcement WebKit provides. Thus, QUARK actually provides security far beyond the handful policies we formally verified. Below we discuss the gap between the subset of policies we verified and the full set of common browser security policies.

The *same origin policy* [37] (SOP) dictates which resources a tab may access. For example, a tab is allowed to load cross-domain images using an `img` tag, but not using an `XMLHttpRequest`.

Unfortunately, we cannot easily verify this policy since restricting how a resource may be used after it has been loaded (*e.g.*, in an `img` tag vs. as a JavaScript value) requires reasoning across abstraction boundaries, *i.e.*, analyzing the large, complex tab implementation instead of treating it as a black box.

The SOP also restricts how JavaScript running in different frames on the same page may access the DOM. We could formally reason about this aspect of the SOP by making frames the basic protection domains in QUARK instead of tabs. To support this refined architecture, frames would own a rectangle of screen real estate which they could subdivide and delegate to sub-frames. Communication between frames would be coordinated by the kernel, which would allow us to formally guarantee that all frame access to the DOM conforms with the SOP.

We only formally prove inter-domain cookie isolation. Even this coarse guarantee prohibits a broad class of attacks, *e.g.*, it protects all Google cookies from any non-Google tab. QUARK does enforce restrictions on cookie access between subdomains; it just does so using WebKit as unverified cookie handling code within our cookie processes. Formally proving finer-grained cookie policies in Coq would be possible and would not require significant changes to the kernel or proofs.

Unfortunately, Quark does not prevent all cookie exfiltration attacks. If a subframe is able to exploit the entire tab, then it could steal the cookies of its top-level parent tab, and leak the stolen cookies by encoding the information within the URL parameter of `GetURL` requests. This limitation arises because tabs are principles in Quark instead of frames. This problem can be prevented by refining Quark so that frames themselves are the principles.

Our socket security policy prevents an important subset of cross-site request forgery attacks [9]. Quark guarantees that a tab uses a `GetURL` message when requesting a resource from sites whose domain suffix doesn't match with the tab's one. Because our implementation of `GetURL` does not send cookies, the resources requested by a `GetURL` message are guaranteed to be publicly available ones which do not trigger any privileged

actions on the server side. This guarantee prohibits a large class of attacks, *e.g.*, cross-site request forgery attacks against Amazon domains from non-Ama-zon domains. However, this policy cannot prevent cross-site request forgery attacks against sites sharing the same domain suffix with the tab, *e.g.*, attacks from a tab on `www.ucsd.edu` against `cse.ucsd.edu` since the tab on `www.ucsd.edu` can directly connect to `cse.ucsd.edu` using a socket and cookies on `cse.ucsd.edu` are also available to the tab.

Compatibility Issues. QUARK enforces non-standard security policies which break compatibility with some web applications. For example, Mashups do not work properly because a tab can only access cookies for its domain and subdomains, *e.g.*, a subframe in a tab cannot properly access any page that needs user credentials identified by cookies if the subframe's domain suffix does not match with the tab's one. This limitation arises because tabs are the principles of Quark as opposed to subframes inside tabs. Unfortunately, tabs are too coarse grained to properly support mashups and retain our strong guarantees.

For the same reason as above, Quark cannot currently support third-party cookies. It is worth noting that third-party cookies have been considered a privacy-violating feature of the web, and there are even popular browser extensions to suppress them. However, many websites depend on third party cookies for full functionality, and our current Quark browser does not allow such cookies since they would violate our non-interference guarantees.

Finally, Quark does not support communications like "postMessage" between tabs; again, this would violate our tab non-interference guarantees.

Despite these incompatibilities, Quark works well on a variety of important sites such as Google Maps, Amazon, and Facebook since they mostly comply with Quarks' security policies. More importantly, our hope is that in the future Quark will provide a foundation to explore all of the above features within a formally verified setting.

In particular, adding the above features will require future work in two broad directions. First, frames need to become the principles in Quark instead of tabs. This change will require the kernel to support parent frames delegating resources like screen region to child frames. Second, finer grained policies will be required to retain appropriate non-interference results in the face of these new features, *e.g.* to support interaction between tabs via "postMessage". Together, these changes would provide a form of "controlled" interference, where frames are allowed to communicate directly, but only in a sanctioned manner. Even more aggressively, one may attempt to re-implement other research prototypes like MashupOS [19] within Quark, going beyond the web standards of today, and prove properties of its implementation.

There are also several other features that Quark does not currently support, and would be useful to add, including local storage, file upload, browser cache, browser history, etc. However, we believe that these are not fundamental limitations of our approach or Quark’s current design. Indeed, most of these features don’t involve inter-tab communication. For the cases where they do (for example history information is passed between tabs if visited links are to be correctly rendered), one would again have to refine the non-interference definition and theorems to allow for controlled flow of information.

7 Discussion

In this section we discuss lessons learned while developing QUARK and verifying its kernel in Coq. In hindsight, these guidelines could have substantially eased our efforts. We hope they prove useful for future endeavors.

Formal Shim Verification. Our most essential technique was *formal shim verification*. For us, it reduced the verification burden to proving a small browser kernel. Previous browsers like Chrome, OP, and Gazelle clearly demonstrate the value of kernel-based architectures. OP further shows how this approach enables reasoning about a model of the browser. We take the next step and formally prove the actual browser implementation correct.

Modularity through Trace-based Specification. We ultimately specified correct browser behavior in terms of traces and proved both that (1) the implementation satisfies the spec and (2) the spec implies our security properties. Splitting our verification into these two phases improved modularity by separating concerns. The first proof phase reasons using monads in Ynot to show that the trace-based specification correctly abstracts the implementation. The second proof phase is no longer bound to reasoning in terms of monads – it only needs to reason about traces, substantially simplifying proofs.

This modularity aided us late in development when we proved address bar correctness (Theorem `dom_bar_correct` in Figure 7). To prove this theorem, we only had to reason about the trace-based specification, not the implementation. As a result, the proof of `dom_bar_correct` was only about 300 lines of code, tiny in comparison to the total proof effort. Thus, proving additional properties can be done with relatively little effort over the trace-based specification, without having to reason about monads or other implementation details.

Implement Non-verified Prototype First. Another approach we found effective was to write a non-verified version of the kernel code before verifying it. This allowed us to carefully design and debug the interfaces between components and to enable the right browsing functionality before starting the verification task.

Iterative Development. After failing to build and ver-

ify the browser in a single shot, we found that an iterative approach was much more effective. We started with a text-based browser, where the tab used lynx to generate a text-based version of QUARK. We then evolved this browser into a GUI-based version based on WebKit, but with no sockets or cookies. Then we added sockets and finally cookies. When combined with our philosophy of “write the non-verified version first”, this meant that we kept a working version of the kernel written in Python throughout the various iterations. Just for comparison, the Python kernel which is equivalent to the Coq version listed in Figure 8 is 305 lines of code.

Favor Ease of Reasoning. When forced to choose between adding complexity to the browser kernel or to the untrusted tab implementation, it was *always* better keep the kernel as simple as possible. This helped manage the verification burden which was the ultimate bottleneck in developing QUARK. Similarly, when faced with a choice between flexibility/extensibility of code and ease of reasoning, we found it best to aim for ease of reasoning.

8 Conclusions

In this paper, we demonstrated how *formal shim verification* can be used to achieve strong security guarantees for a modern Web browser using a mechanical proof assistant. We formally proved that our browser provides tab noninterference, cookie integrity and confidentiality, and address bar integrity and correctness. We detailed our design and verification techniques and showed that the resulting browser, QUARK, provides a modern browsing experience with performance comparable to the default WebKit browser. For future research, QUARK furnishes a framework to easily experiment with additional web policies without re-engineering an entire browser or formalizing all the details of its behavior from scratch.

9 Acknowledgments

We thank Kirill Levchenko for many fruitful conversations regarding shim verification. We would also like to thank our shepherd, Anupam Datta, and the anonymous reviewers for helping us improve our paper.

References

- [1] <http://gallium.inria.fr/~dargaye/mlcompcert.html>.
- [2] Chrome security hall of fame. <http://dev.chromium.org/Home/chromium-security/hall-of-fame>.
- [3] Public suffix list. <http://publicsuffix.org/>.
- [4] Pwn2own. <http://en.wikipedia.org/wiki/Pwn2own>.
- [5] AKHAWA, D., BARTH, A., LAMY, P. E., MITCHELL, J., AND SONG, D. Towards a formal foundation of web security. In *Proceedings of CSF 2010* (July 2010), M. Backes and A. Myers, Eds., IEEE Computer Society, pp. 290–304.

- [6] ANSEL, J., MARCHENKO, P., ERLINGSSON, Ú., TAYLOR, E., CHEN, B., SCHUFF, D. L., SEHR, D., BIFFLE, C., AND YEE, B. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *PLDI* (2011), pp. 355–366.
- [7] BALL, T., MAJUMDAR, R., MILLSTEIN, T., AND RAJAMANI, S. K. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, June 2001).
- [8] BARTH, A., JACKSON, C., AND MITCHELL, J. C. Robust defenses for cross-site request forgery. In *ACM Conference on Computer and Communications Security* (2008), pp. 75–88.
- [9] BARTH, A., JACKSON, C., AND MITCHELL, J. C. Robust defenses for cross-site request forgery. In *To appear at the 15th ACM Conference on Computer and Communications Security (CCS 2008)* (2008).
- [10] BARTH, A., JACKSON, C., REIS, C., AND THE GOOGLE CHROME TEAM. The security architecture of the Chromium browser. Tech. rep., Google, 2008.
- [11] BOHANNON, A., PIERCE, B. C., SJÖBERG, V., WEIRICH, S., AND ZDANCEWIC, S. Reactive noninterference. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009).
- [12] CHEN, E. Y., BAU, J., REIS, C., BARTH, A., AND JACKSON, C. App isolation: get the security of multiple browsers with just one. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011).
- [13] CHEN, S., MESEGUER, J., SASSE, R., WANG, H. J., AND MIN WANG, Y. A systematic approach to uncover security flaws in GUI logic. In *IEEE Symposium on Security and Privacy* (2007).
- [14] CHUGH, R., MEISTER, J. A., JHALA, R., AND LERNER, S. Staged information flow for javascript. In *PLDI* (2009).
- [15] COOK, B., PODELSKI, A., AND RYBALCHENKO, A. Terminator: Beyond safety. In *CAV* (2006).
- [16] DAS, M., LERNER, S., AND SEIGLE, M. ESP: Path-sensitive program verification in polynomial time. In *PLDI* (2002).
- [17] GRIER, C., TANG, S., AND KING, S. T. Secure web browsing with the OP web browser. In *IEEE Symposium on Security and Privacy* (2008).
- [18] HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND SUTRE, G. Lazy abstraction. In *POPL* (2002).
- [19] HOWELL, J., JACKSON, C., WANG, H. J., AND FAN, X. MashupOS: operating system abstractions for client mashups. In *HotOS* (2007).
- [20] HUANG, L.-S., WEINBERG, Z., EVANS, C., AND JACKSON, C. Protecting browsers from cross-origin css attacks. In *ACM Conference on Computer and Communications Security* (2010), pp. 619–629.
- [21] JACKSON, C., AND BARTH, A. Beware of finer-grained origins. In *In Web 2.0 Security and Privacy (W2SP 2008)* (May 2008).
- [22] JACKSON, C., BARTH, A., BORTZ, A., SHAO, W., AND BONEH, D. Protecting browsers from dns rebinding attacks. In *ACM Conference on Computer and Communications Security* (2007), pp. 421–431.
- [23] JANG, D., JHALA, R., LERNER, S., AND SHACHAM, H. An empirical study of privacy-violating information flows in JavaScript Web applications. In *Proceedings of the ACM Conference Computer and Communications Security (CCS)* (2010).
- [24] JANG, D., TATLOCK, Z., AND LERNER, S. Establishing browser security guarantees through formal shim verification. Tech. rep., UC San Diego, 2012.
- [25] JANG, D., VENKATARAMAN, A., SAWKA, G. M., AND SHACHAM, H. Analyzing the cross-domain policies of flash applications. In *In Web 2.0 Security and Privacy (W2SP 2011)* (May 2011).
- [26] JIM, T., SWAMY, N., AND HICKS, M. Defeating script injection attacks with browser-enforced embedded policies. In *WWW* (2007), pp. 601–610.
- [27] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: formal verification of an OS kernel. In *SOSP* (2009).
- [28] LEROY, X. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *PLDI* (2006).
- [29] MALECHA, G., MORRISSETT, G., SHINNAR, A., AND WISNESKY, R. Toward a verified relational database management system. In *POPL* (2010).
- [30] MALECHA, G., MORRISSETT, G., AND WISNESKY, R. Trace-based verification of imperative programs with I/O. *J. Symb. Comput.* 46 (February 2011), 95–118.
- [31] MICKENS, J., AND DHAWAN, M. Atlantis: robust, extensible execution environments for web applications. In *SOSP* (2011), pp. 217–231.
- [32] MORRISSETT, G., TAN, G., TASSAROTTI, J., TRISTAN, J.-B., AND GAN, E. Rocksalt: Better, faster, stronger sfi for the x86. In *PLDI* (2012).
- [33] NANEVSKI, A., MORRISSETT, G., AND BIRKEDAL, L. Polymorphism and separation in Hoare type theory. In *ICFP* (2006).
- [34] NANEVSKI, A., MORRISSETT, G., SHINNAR, A., GOVEREAU, P., AND BIRKEDAL, L. Ynot: Dependent types for imperative programs. In *ICFP* (2008).
- [35] PROVOS, N., FRIEDL, M., AND HONEYMAN, P. Preventing privilege escalation. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12* (2003), USENIX Association.
- [36] RATANAWORABHAN, P., LIVSHITS, V. B., AND ZORN, B. G. Nozzle: A defense against heap-spraying code injection attacks. In *USENIX Security Symposium* (2009), pp. 169–186.
- [37] RUDERMAN, J. The same origin policy, 2001. <http://www.mozilla.org/projects/security/components/same-origin.html>.
- [38] SAXENA, P., AKHAWA, D., HANNA, S., MAO, F., MCCAMANT, S., AND SONG, D. A symbolic execution framework for javascript. In *IEEE Symposium on Security and Privacy* (2010), pp. 513–528.
- [39] SINGH, K., MOSHCHUK, A., WANG, H. J., AND LEE, W. On the incoherencies in web browser access control policies. In *IEEE Symposium on Security and Privacy* (2010), pp. 463–478.
- [40] STAMM, S., STERNE, B., AND MARKHAM, G. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web* (2010), WWW '10, pp. 921–930.
- [41] TANG, S., MAI, H., AND KING, S. T. Trust and protection in the illinois browser operating system. In *OSDI* (2010), pp. 17–32.
- [42] WANG, H. J., GRIER, C., MOSHCHUK, A., KING, S. T., CHOUDHURY, P., AND VENTER, H. The multi-principal OS construction of the gazelle web browser. Tech. Rep. MSR-TR-2009-16, MSR, 2009.
- [43] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and understanding bugs in C compilers. In *PLDI* (2011).
- [44] YU, D., CHANDER, A., ISLAM, N., AND SERIKOV, I. Javascript instrumentation for browser security. In *POPL* (2007), pp. 237–249.

Neuroscience Meets Cryptography: Designing Crypto Primitives Secure Against Rubber Hose Attacks

Hristo Bojinov Daniel Sanchez, Paul Reber Dan Boneh Patrick Lincoln
Stanford University Northwestern University Stanford University SRI

Abstract

Cryptographic systems often rely on the secrecy of cryptographic keys given to users. Many schemes, however, cannot resist coercion attacks where the user is forcibly asked by an attacker to reveal the key. These attacks, known as *rubber hose cryptanalysis*, are often the easiest way to defeat cryptography. We present a defense against coercion attacks using the concept of *implicit learning* from cognitive psychology. Implicit learning refers to learning of patterns without any conscious knowledge of the learned pattern. We use a carefully crafted computer game to plant a secret password in the participant's brain without the participant having any conscious knowledge of the trained password. While the planted secret can be used for authentication, the participant cannot be coerced into revealing it since he or she has no conscious knowledge of it. We performed a number of user studies using Amazon's Mechanical Turk to verify that participants can successfully re-authenticate over time and that they are unable to reconstruct or even recognize short fragments of the planted secret.

1 Introduction

Consider the following scenario: a high security facility employs a sophisticated authentication system to check that only persons who know a secret key, possess a hardware token, and have an authorized biometric can enter. Guards ensure that only people who successfully authenticate can enter the facility. Now, suppose a clever attacker captures an authenticated user. The attacker can steal the user's hardware token, fake the user's biometrics, and coerce the victim into revealing his or her secret key. At this point the attacker can impersonate the victim and defeat the expensive authentication system deployed at the facility.

So-called rubber hose attacks have long been the bane of security systems and are often the easiest way to de-

feat cryptography [22]. The problem is that an authenticated user must possess authentication credentials and these credentials can be extracted by force [19] or by other means.

In this work we present a new approach to preventing rubber hose attacks using the concept of *implicit learning* [5, 17] from cognitive psychology. Implicit learning is believed to involve the part of the brain called the *basal ganglia* that learns tasks such as riding a bicycle or playing golf by repeatedly performing those tasks. Experiments designed to trigger implicit learning show that knowledge learned this way is not consciously accessible to the person being trained [17]. An everyday example of this phenomenon is riding a bicycle: we know how to ride a bicycle, but cannot explain how we do it. Section 2 gives more background of the relevant neuroscience.

Implicit learning presents a fascinating tool for designing coercion-resistant security systems. In this paper we focus on user authentication where implicit learning is used to plant a password in the human brain that can be detected during authentication, but cannot be explicitly described by the user. Such a system avoids the problem that people can be persuaded to reveal their password. To use this system, participants would be initially trained to do a specific task called Serial Interception Sequence Learning (SISL), described in the next section. Training is done using a computer game that results in implicit learning of a specific sequence of key strokes that functions as an authentication password. In our experiments, training sessions last approximately 30 to 45 minutes and participants learn a random password that has about 38 bits of entropy. We conducted experiments to show that after training, participants cannot reconstruct the trained sequence and cannot even recognize short fragments of it.

To be authenticated at a later time, a participant is presented with multiple SISL tasks where one of the tasks contains elements from the trained sequence. By exhibiting reliably better performance on the trained ele-

ments compared to untrained, the participant validates his or her identity within 5 to 6 minutes. An attacker who does not know the trained sequence cannot exhibit the user's performance characteristics measured at the end of training. Note that the authentication procedure is an interactive game in which the server knows the participant's secret training sequence and uses it to authenticate the participant. Readers who want to play with the system can check out the training game at brainauth.com/testdrive.

While in this paper we focus on coercion-resistant user authentication systems, authentication is just the tip of the iceberg. We expect that many other coercion-resistant security primitives can be designed using implicit learning.

Threat model. The proposed system is designed to be used as a local password mechanism requiring physical presence. That is, we consider authentication at the entrance to a secure location where a guard can ensure that a real person is taking the test without the aid of any electronics.

To fool the authentication test the adversary is allowed to intercept one or more trained users and get them to reveal as much as they can, possibly using coercion. Then the adversary, on his own, engages in the live authentication test and his goal is to pass the test.

We stress that as with standard password authentication, the system is not designed to resist eavesdropping attacks such as shoulder surfing during the authentication process. While challenge-response protocols are a standard defense against eavesdropping, it is currently an open problem to design a challenge-response protocol based on implicit learning. We come back to this question at the end of the paper.

Benefits over biometric authentication. The trained secret sequence can be thought of as a biometric key authenticating the trained participant. However, unlike biometric keys the authenticating information cannot be surreptitiously duplicated and participants cannot reveal the trained secret even if they want to. In addition, if the trained sequence is compromised, a new identifying sequence can be trained as a replacement, resulting in a change of password.

We discuss other related work in Section 6, but briefly mention here a related result of Denning et al. [4] that uses images to train users to implicitly memorize passwords. This approach is not as resistant to rubber hose attacks since users will remember images they have seen versus ones they have not, giving an attacker information that can be used for authentication. Additionally, image-based methods require large sets of images to be prepared and used only once per user making the system difficult to deploy. Our combinatorial approach lets us

lower bound the entropy of the learned secrets, is simple to set up, and is designed to leave no conscious trace of the trained sequences.

User studies. To validate our proposal we performed a number of user studies using Amazon's Mechanical Turk. We asked the following core questions that explore the feasibility of authentication via implicit learning:

- Is individual identification reliable? That is, can trained users re-authenticate and can they do it over time?
- Can an attacker reverse engineer the sequence from easily obtained performance data from a trained participant?

Across three experiments, we present promising initial results supporting the practical implementation of our design. First, we show that identification is possible with relatively short training and a simple test. Second, the information learned by the user persists over delays of one and two weeks: while there is some forgetting over a week, there is little additional forgetting at two weeks suggesting a long (exponentially shaped) forgetting curve. Finally, in a third experiment we examined an attack based on having participants complete sequences containing all minimal-length fragments needed to try to reconstruct the identification sequence: our results show that participants do not express reliable sequence knowledge under this condition, indicating that the underlying sequence information is resistant to attack until longer subsequences are guessed correctly by the attacker.

2 An Overview of the Human Memory System

The difference between knowing how to perform a well-learned skill and being able to explain that performance is familiar to anyone who has acquired skilled expertise. This dissociation reflects the multiple memory systems in the human brain [14]. Memory for verbally reportable facts, events and episodes depends on the medial temporal lobe memory system (including the hippocampus). Damage to this system due to stroke, Alzheimer's disease neuropathology, or aging leads to impairments in conscious, explicit memory. However, patients with impairments to explicit memory often show an intact ability to acquire new information implicitly, including exhibiting normal learning of several kinds of skills. The types of learning preserved in memory-disordered patients are those learned incidentally through practice: even in healthy participants the information thus acquired cannot be easily verbally described.

Several decades of experimental cognitive psychology have led to the development of tasks that selectively de-

pend on this type of implicit, non-conscious learning system. These tasks typically present information covertly with embedded structure in a set of experimental stimuli. Although participants are not attempting to learn this structure, evidence for learning can be observed in their performance.

The covertly embedded information often takes the form of a statistical structure to a sequence of responses. Participants exhibit improved performance when the responses follow this sequence and performance declines if the structure is changed [12]. The improvement in performance can occur completely outside of awareness, that is, participants do not realize there is any structure nor can they recognize the structure when shown [17]. The lack of awareness of learning indicates the memory system supporting learning is not part of the explicit, declarative memory system and instead is hypothesized to depend on the basal ganglia and connections to motor cortical areas [6].

Less is known about the information processing characteristics of the cortico-striatal memory system operating in the connections between the basal ganglia and motor cortical areas. Most prior research has examined learning of simple structures with small amounts of information, typically repeating sequences of actions 10-12 items in length. However, more recent studies have found that long, complex sequences can be learned fairly rapidly by this memory system and that learning is relatively unaffected by noise [18]. The ability to learn repeating sequences that are at least 80 items long relatively rapidly and the fact that this training can be hidden within irrelevant responses (noise) during training suggests an intriguing possibility for covertly embedding non-reportable cryptographic data within the cortico-striatal memory system in the human brain.

2.1 The SISL Task and Applet

The execution of the Serial Interception Sequence Learning (SISL) task is central to the authentication system that we have developed. Here we introduce the SISL task in the context of the human memory system in order to provide background for describing our design and practical experiments.

Originally introduced in [17], SISL is a task in which human participants develop sensitivity to structured information without being aware of what they have learned. The task requires participants to intercept moving objects (circles) delivered in a pre-determined sequence, much like this is done in the popular game “Guitar Hero”. Initially each object appears at the top of one of four different columns, and falls vertically at a constant speed until it reaches the “sink” at the bottom, at which point it disappears. The goal for the player is to

intercept every object as it nears the sink. Interception is performed by pressing the key that corresponds to the object’s column when the object is in the correct vertical position. Pressing the wrong key or not pressing any key results in an incorrect outcome for that object. In a typical training session of 30-60 minutes, participants complete several thousand trials and the order of the cues follows a covertly embedded repeating sequence on 80% of trials. The game is designed to keep each user at (but not beyond) the limit of his or her abilities by gradually varying the speed of the falling circles to achieve a hit rate of about 70%. Knowledge of the embedded repeating sequence is assessed by comparing the performance rate (percent correct) during times when the cues follow the trained sequence to that during periods when the cues follow an untrained sequence.

All of the sequences presented to the user are designed to prevent conspicuous, easy to remember patterns from emerging. Specifically, training as well as random sequences are designed to contain every ordered pair of characters exactly once with no character appearing twice in a row, and thus the sequence length must be $4 \times 3 = 12$ when four columns (characters) are used. The result is that while the trained sequence is performed better than an untrained sequence, the participant usually does not consciously recognize the trained sequence. In order to confirm this in experimental work, after SISL participants are typically asked to complete tests of explicit recognition in which they specify how familiar various sequences look to them.

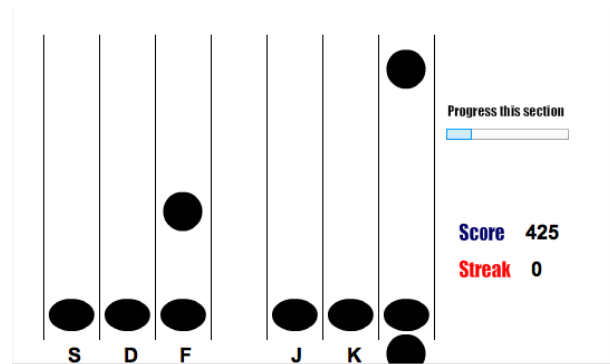


Figure 1: Screenshot of the SISL task in progress.

For the current application, we extended the traditional definition of the SISL task in order to accommodate its use as an authentication mechanism. First, we increased the number of columns to six, which increases the potential complexity of the trained sequence. Using the same constraints on sequence order as the 4-column version of the task, the training sequences are 30 items long. As a result, the number of possible sequences that can be used

as a secret key is increased exponentially from only 256 to nearly 248 billion, as explained in the next section. Second, we added an empty column in the middle of the layout (Figure 1). In early experimental testing we found out that the empty column facilitates the visual perception of the falling objects and helps the user to “map” them to the correct hand, especially for objects in the middle columns which are otherwise easily confused at high speed.

The SISL task is delivered to users as a Flash application via a web browser. Participants navigate to our web site, www.brainauth.com, and are presented with a consent form. Once they agree to participate, the applet downloads a random training sequence and starts the game. Upon completion of the training and test trials, the explicit recognition test is administered, and results are uploaded to the server. Once we describe our authentication system, we will return to describe how the SISL applet functions in the bigger scheme of our experiments with multiple users.

3 The Basic Authentication System Using Implicit Learning

The SISL task provides a method for storing a secret key within the human brain that can be detected during authentication, but cannot be explicitly described by the user. Such a system avoids the problem that people can be persuaded to reveal their password and can form the basis of a coercion-resistant authentication protocol. If the information is compromised, a new identifying sequence can be trained as a replacement—resulting in a change of password.

The identification system operates in two steps: training followed by authentication. In the training phase, the secret key learned by the user is as in the expanded SISL task, namely a sequence of 30 characters over the set $S = \{s, d, f, j, k, l\}$. We only use 30-character sequences that correspond to an Euler cycle in the graph shown in Figure 2 (i.e. a cycle where every edge appears exactly once). These sequences have the property that every non-repeating bigram over S (such as ‘sd’, ‘dj’, ‘fk’) appears exactly once. In order to anticipate the next item (e.g., to show a performance advantage), it is necessary to learn associations among groups of three or more items. This eliminates learning of letter frequencies or common pairs of letters, which reduces *conscious* recognition of the embedded repeating sequence [5].

Let Σ denote the set of all possible secret keys, namely the set of 30-character sequences corresponding to Euler cycles in Figure 2. The number of Euler cycles in this graph can be computed using the BEST theorem [20]

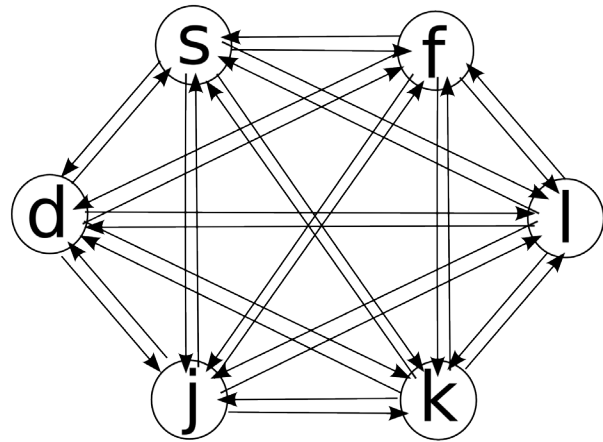


Figure 2: The secret key we generate is a random 30-character sequence from the set of Euler cycles in this directed graph. The resulting sequence contains all bigrams exactly once, excluding repeating characters.

which gives

$$\#keys = 6^4 \cdot 24^6 \approx 2^{37.8} .$$

Hence the learned random secret has about 38 bits of entropy which is far more than the entropy of standard memorized passwords.

Training. Users learn a random 30-item secret key $k \in \Sigma$ by playing the SISL game in a trusted environment. To train users we experimented with the following procedure:

- While performing the SISL task the trainee is presented with the 30-item secret key sequence repeated three times followed by 18 items selected from a random other sequence (subject to the constraint that there will be no back-to-back repetitions of the same cue), for a total of 108 items.
- This sequence is repeated five times, so that the trainee is presented with a total of 540 items.
- At the end of this sequence there is a short pause in the SISL game and then the entire sequence of 540 items (including the pause at the end) is repeated six more times.

During the entire training session the trainee is presented with $7 \times 540 = 3780$ items which takes approximately 30-45 minutes to complete. After the training phase completes, the trainee runs through the authentication test described next to ensure that training succeeded. The system records the final playing speed that the user achieved.

SISL Authentication. To authenticate at a later time, a trained user is presented with the SISL game where the structure of the cues contains elements from the trained authentication sequence and untrained elements for comparison. By exhibiting reliably better performance on the trained elements compared to untrained, the participant validates his or her identity. Specifically we experimented with the following authentication procedure:

- Let k_0 be the trained 30-item sequence and let k_1, k_2 be two additional 30-item sequences chosen at random from Σ . The same sequences (k_0, k_1, k_2) are used for all authentication sessions.
- The system chooses a random permutation π of $(0, 1, 2, 0, 1, 2)$ (e.g., $\pi = (2, 1, 0, 0, 2, 1)$) and presents the user with a SISL game with the following sequence of $540 = 18 \times 30$ items:

$$k_{\pi_1}, k_{\pi_1}, k_{\pi_1}, \dots, k_{\pi_6}, k_{\pi_6}, k_{\pi_6}.$$

That is, each of k_0, k_1, k_2 is shown to the user exactly six times (two groups of three repetitions), but ordering is random. The game begins at the speed at which the training for that user ended.

- For $i = 0, 1, 2$ let p_i be the fraction of correct keys the user entered during all plays of the sequence k_i . The system declares that authentication succeeded if

$$p_0 > \text{average}(p_1, p_2) + \sigma \quad (3.1)$$

Where $\sigma > 0$ is sufficiently large to minimize the possibility that this gap occurred by chance, but without causing authentication failures.

In the above, preliminary formulation, the authentication process is potentially vulnerable to an attack by which an untrained user degrades his performance across two blocks hoping to exhibit an artificial performance difference in favor of the trained sequence (and obtaining a 1/3 chance of passing authentication). We discuss a robust defense against this in Section 5, but for now we mention that two simple precautions offer some protection, even for this simple assessment procedure. First, verifying that the authenticator is a live human makes it difficult to consistently change performance across the foil blocks k_1, k_2 . Second, the final training speed obtained during acquisition of the sequence is known to the authentication server and the attacker is unlikely to match that performance difference between the trained and foil blocks. A performance gap that is substantially different from the one obtained after training indicates an attack.

Analysis. The next two sections discuss two critical aspects of this system:

- Usability: can a trained user complete the authentication task reliably over time?
- Security: can an attacker who intercepts a trained user coerce enough information out of the user to properly authenticate?

4 Usability Experiments

We report on preliminary experiments that demonstrate feasibility and promise of the SISL authentication system. We carried out the experiments in three stages. First, we established that reliable learning was observed with the new expanded version of the SISL task using Mechanical Turk. Second, we verified that users retain the knowledge of the trained sequence after delays of one and two weeks. Finally, we investigated the effectiveness of an attack on participants' sequence knowledge based on sampling the smallest fragments from which the original sequence could potentially be reconstructed.

The experiments were carried out online within Amazon's Mechanical Turk platform. The advantages of Mechanical Turk involve a practically unlimited base of participants, and a relatively low cost. One drawback of running the experiments online is the relative lack of control we had over users coming back at a later time for repeat evaluations. We discuss all of these considerations towards the end of the section.

4.1 Experiment 1: Implicit and Explicit Learning

Our first experiment confirmed that implicit learning can be clearly detected while explicit conscious sequence knowledge was minimal. Experimental data from 35 participants were included in the analysis.

The experiment used the training procedure described in the previous section where the training phase contained 3780 total trials and took approximately 30-45 minutes to complete. Recall that training consists of seven 540-trial training blocks. After the training session, participants completed a SISL authentication test that compares performance on the trained sequence to performance on two random test sequences.

Learning of the trained sequence is shown in Figure 3 as a function of the performance advantage (increase in percent correct responses) for the trained sequence compared with the randomly occurring noise segments. On the test block following training, participants performed the SISL task at an average rate of 79.2% correct for the trained sequence and 70.6% correct for the untrained sequence. The difference of 8.6% correct (SE 2.4%)¹

¹SE is short hand for *Standard Error*.

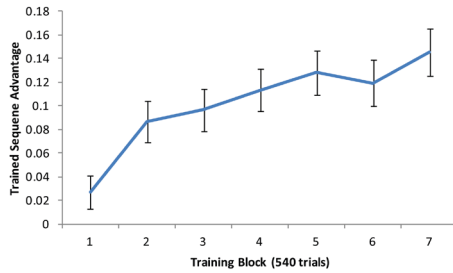


Figure 3: Across training participants gradually begin to express knowledge of the repeating sequence by exhibiting a performance advantage for the trained sequence compared to randomly interspersed noise segments. Note that overall performance on the task stays at around 70% throughout due to the adaptive nature of the task by which the speed is increased as participants become better at general SISL performance.

indicated reliably better performance for the trained sequence. By one-sample *t*-test versus zero, the expected difference between trained and untrained if there was no learning² would be $t(34) = 3.55, p < .01$.

Group-level differences in performance are commonly seen on tests of implicit learning, but being able to reliably assess individual learning is necessary for an authentication method. On an individual participant basis, performance on the trained sequence could be discriminated from the untrained sequence on the 540 trials (by chi-squared analysis at $p < .05$) in 25 of 35 cases. For authentication purposes, the individual reliability of the assessment will need to be further improved by longer training to establish the implicitly learned sequence. However, the ability to identify learning in a large fraction of individuals with relatively short training is a feature of the SISL task not seen in most tests of implicit learning.

Explicit recognition test. After the training and test blocks, participants were presented with five different animated sequences and asked how familiar each looked on a scale of 0 to 10). Of the five sequences, one was the trained sequence and the other four were randomly selected foils. This test assessed explicit recognition memory for the trained sequence.

On the recognition test, participants rated the trained

²In other words, if the percent correct measurements for trained and untrained sequences followed the same normal distribution, the *t*-value calculated with $N = 35$ samples (and thus $N - 1 = 34$ degrees of freedom), should be near zero—less than 3.55 with 99% probability ($p = 0.01$); in contrast, the value we obtained was 8.6. The *t*-test is a standard statistical method used to confirm that the manipulated variable (here, sequence type) affects the measured variable (performance correct).

sequence as familiar at an average of 6.5 (SE 0.4) on the 0-10 scale and rated novel untrained sequences at 5.15 (SE 0.3). The modestly higher recognition of the trained sequence was reliable across the group, $t(34) = 3.69, p < .01$, but did not correlate with SISL performance ($r = 0.13$) indicating that it did not contribute to the implicit test. Slightly higher recognition of the trained sequence is often seen in implicit learning experiments as healthy participants find some parts of the training sequence familiar after practice. It is worth noting that implicit memory does not transform into explicit knowledge, even with repeated use, and the structure and length of the training and test sequences specifically aim to reduce the possibility that explicit knowledge is accumulated over time.

The general small difference in recognition ratings (5.15 vs. 6.5) indicates that participants would not be able to recall the 30-item sequence meaning that they could not consciously produce the training information (e.g. to compromise the security of the authentication method). One participant remarked in a follow-up email message:

“... To be honest I was not that sure of the quizzes at the end. When I played the tempo was so high it was incredibly difficult to keep a track of the circles. Most of the time my fingers moved by themselves, at least it felt that way. I noticed two repeating patterns over all the levels. (I’m not totally sure what the buttons were, was it DFG JKL?) One was D-F-G-F-D I think and the other I’m not quite sure the sequence but it was a four or five button series which went from the left to the right and back to the left...”

We discuss the reconstruction question further in our third experiment.

4.2 Experiment 2: Recall Over Time

An authentication mechanism is only useful if authentication can still be accurately performed at some time after the password is memorized. In Experiment 2, we confirmed that sequence-specific knowledge acquired by users was retained over prolonged periods of time. Although skill learning generally persists over time, a SISL-based test had never been conducted with a substantial delay and a sufficient number of participants.

In Experiment 2, participants agreed to perform the SISL task over two sessions. In the first session, participants completed a training sequence which the same structure as the one in Experiment 1. The training was immediately followed by the same SISL test to assess sequence knowledge before the delay. A group of 32 participants returned to the online applet after 1 week to

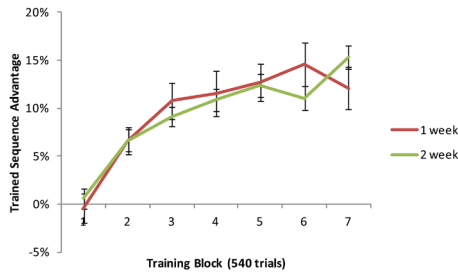


Figure 4: Across training participants gradually begin to express knowledge of the repeating sequence by exhibiting a performance advantage for the trained sequence compared to randomly interspersed noise segments. Learning performance was similar across both groups and similar to Experiment 1, as expected.

perform a retention test and recognition assessment for the trained sequence. A separate group of 80 participants returned after a 2 week delay for the retention and recognition tests. For the 1-week group, the test session consisted of a 540-trial implicit sequence learning assessment. For the 2-week group, the test session was doubled in length to additionally evaluate whether a longer test provided better sensitivity to individual sequence knowledge. For both groups, the initial speed of the test on the delay session was set to match the speed with which the participants had been performing the task at the end of the training session. A short warm-up block of 180 trials was used to adjust this initial speed so that participants were performing at around the target 70% correct at the beginning of the retention test.

Figure 4 shows gradual learning of the trained sequence during the first session for both groups as in Experiment 1. Implicit sequence knowledge at both immediate and delayed tests is shown in Figure 5. On all five assessments, participants exhibited reliable sequence learning as a group, $t_s > 4.3$, $p_s < .01$. On the one-week delay test, 15 of 32 participants exhibited individually reliable sequence knowledge. However, for the two-week delay group, 49 of 80 participants exhibited reliable sequence knowledge reflecting the increased sensitivity in the longer assessment test used. Future research will examine both increased training time and assessment tests with increased sensitivity to individual knowledge to provide a reliable and accurate identification method by SISL performance.

Even at one and two weeks delay, participants exhibited the same modest tendency for better recognition of the trained sequence, $t_s > 2.8$, $p_s < .05$. Again, recognition performance did not correlate with expression of sequence knowledge, $r_s < .16$ and did not suggest any

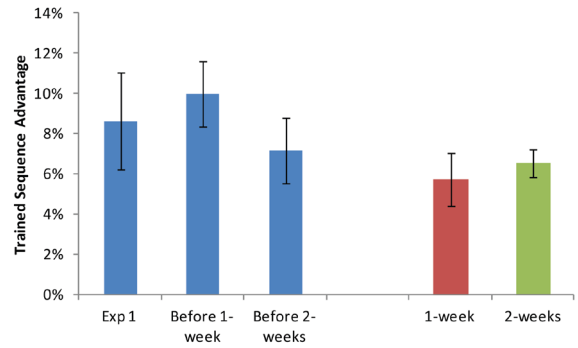


Figure 5: Participants exhibited reliable sequence knowledge on both immediate assessments (shown for Experiment 1 and both conditions of Experiment 2) shown by a performance advantage for the trained sequence compared with untrained, novel sequences at test. Sequence knowledge is retained at both the 1 and 2 week delay test sessions. While there is some reduction in expressed knowledge after either delay, the lack of significant additional decay from 1 to 2 weeks suggests that information is likely to persist for significant periods following 2 weeks (exponential or power-law decay curves are commonly observed for many types of memory).

ability to recall the entire 30-item trained sequences.

4.3 Mechanical Turk

Running our experiments over Mechanical Turk required considerable thought and effort to ensure that the experiments do not suffer from selection bias and are conducted fairly for both participants and researchers.

One of the early initial questions was that of setting the price for user participation. The training block, which comprises the bulk of the initial session, takes approximately 30-40 minutes to complete depending on player skill. We wanted to motivate our participants to perform to the best of their abilities, and thus set a price of \$5.00 for standalone sessions, assuming a total of approximately one hour of work involved. Apart from isolated complaints from users who thought the game moved too slowly (likely due to them not pressing keys, or playing incorrectly), most users were happy to participate and even solicited additional work. We defined our HIT (Human Intelligence Task) such that each worker could participate only once in it and we believe that there were few—if any—cases where the same user submitted multiple responses.

We had to design special incentives for participants to return and complete the second part in the case of two-session experiments. The approach that worked well for

us was to price the initial (much lengthier) part at \$2.00 and the follow-up 15-minute session at \$6.00. We also explained clearly that this is a two-HIT sequence, and that payment for both parts will only be processed once the second part is done. No-shows at the second session would get no payment at all. Additionally we used Amazon’s command line tools to automatically send reminders to participants when the second session was available and due. As a result, we saw over 90% of the people who completed the first session return and finish the second part.

Due to the special requirements of the SISL application we had to create what is considered to be an “external HIT”, exposing the task as a public website. In order to make sure that results submitted in Amazon correspond to valid submissions in our system, we designed a system that involves a receipt code for every successfully completed session. The code is a 6-digit number between 100000 and 999999—we chose this size to prevent people from easily guessing the code, but not make it difficult for them to write it down (especially useful in two-session experiments, where we also have to fetch the correct follow-up sequence that matches the user’s first visit). After follow-up sessions we provided the user with a second code that needed to be submitted to the separate second HIT in order to receive payment.

Naturally we were concerned about the security of our system, so we took measures to only accept limited types of input as parameters, leaving the website open mostly to denial of service attacks which we had no reason to expect. In comparison, our fear of legitimate users trying to cheat the system and getting paid without completing quality work was somewhat more justified. We saw some limited instances of behavior in this category:

- There were users who, against the instructions, submitted an invalid receipt code. We immediately rejected any such submissions.
- Some users submitted sequences that were so long that they did not fit in our generous allowance on the server. Upon examination we found out that these were due primarily to excessive wrong key presses (sometimes 5 or more key presses for the same object, which suggests that possibly an automated tool was used to complete the task).
- In relatively few situations we noticed users who had unusually long intervals of inactivity. We excluded the most outrageous submissions but leaned towards including the rest in the results of the study in order to avoid biasing our data towards people who did well.

The scope of these abuses never amounted to more than 5% of the submissions, and we believe that the

Experiment	Part	Submissions		
		All	Paid	Used
baseline		46	39	34
1 week delay	initial	35	32	32
1 week delay	follow-up	45	32	32
2 week delay	initial	100	95 (a)	82
2 week delay	follow-up	111	84 (b)	82
trigrams		37	34	32

Table 1: Total number of participants in each experiment. The higher number of submissions on follow-up session are due to more failed opportunistic attempts by users to get paid \$6.00 for no work because HIT assignments were remaining available longer, waiting for eligible users to show up. Notes: (a) we paid more people than necessary due to the 16-day auto-approval configuration of the HIT; (b) we paid, but did not evaluate a submission which came in after the cut-off time; (c) the variation in number of participants across experiments was due to varying response and acceptance rates—our primary goal was to collect enough data to be able to make statistical inferences, and we deliberately collected more data for the most difficult experiment (the 2-week delay).

organization of the Mechanical Turk system is at least partially to thank: workers need to register, and provide some sort of payment account which makes their identity relatively easy to track; moreover, rejected work negatively affects a worker’s score and as a result most users genuinely try to do the best they can, get entertained if possible, and earn some extra money in the process. Overall, we consider our use of Mechanical Turk to have been a big success: it allowed us to conduct each experiment practically overnight, drawing on the huge available pool of participants.

5 Security Analysis

In this section we analyze the security of the basic authentication protocol from Section 3 and propose a number of extensions that improve security. We also experiment with a particular attack that attempts to extract the secret sequence from the user one fragment at a time. Our Mechanical Turk experiment shows that this attack works poorly on humans.

5.1 Implicit Learning as a Cryptographic Primitive

We begin with an abstract model of the new functionality enabled by implicit learning. Traditional modeling

of participants in a cryptographic protocol are as entities who hold secrets unknown to the adversary. These assumptions fall apart in the face of coercion since all secrets can be extracted from the participant.

Implicit learning provides the following new abstract functionality: the training phase embeds a predicate

$$p : \Sigma \rightarrow \{0, 1\}$$

in the user's brain for some large set Σ . Anyone can ask the user to evaluate his or her predicate p at a point $k \in \Sigma$. The predicate evaluates to 1 when k has been learned by the user and evaluates to 0 otherwise. The number of inputs at which p evaluates to 1 is relatively small. Most often p will only evaluate to 1 at a single point meaning that the user has been trained on only one secret sequence.

The key feature of implicit learning is that even under duress it is impossible to extract a point $k \in \Sigma$ from the user for which $p(k) = 1$. This abstract property captures the fact that the secret sequence k is implicitly learned by the user and not consciously accessible. In this paper, we use the implicit learning primitive to construct an authentication system, but one can imagine it being used more broadly in security systems.

The authentication procedure described in Section 3 provides an implementation of the predicate $p(\cdot)$ for some sequence k_0 in Σ . If the procedure declares success we say that $p(k_0) = 1$ and otherwise $p(k_0) = 0$. The predicate p is embedded in the user's brain during the training session.

The basic coercion threat model. The SISL authentication system from Section 3 is designed to resist an adversary who tries to fool the authentication test. We assume the test requires physical presence and begins with a liveness check to ensure that a real person is taking the test without the aid of any instruments. To fool the authentication test the adversary is allowed the following sequence of steps:

- Extraction phase: intercept one or more trained users and get them to reveal as much as they can, possibly using coercion.
- Test phase: the adversary, on his own, submits to the authentication test and his or her goal is to pass the test. In real life this could mean that the adversary shows up at the entrance to a secure facility and attempts to pass the authentication test there. If he fails he could be detained for questioning.

This basic threat model gives the attacker a single chance at the authentication test. We consider a model where the attacker may iterate the extraction and test

phases, alternating between extraction and testing, later on in this section.

We also note that the basic threat model assumes that during the training phase, when users are taught the credential, users are following the instructions and are not deliberately trying to mislead the training process. In effect, the adversary is only allowed to coerce a user after the training process completes.

It is straight-forward to show that the system of Section 3 is secure under this basic threat model, assuming the training procedure embeds an implicitly learned predicate p in the user's brain. Indeed, if the attacker intercepts u trained users and subjects each one to q queries, his chances of finding a valid sequence is at most $qu/|\Sigma|$. Since each test takes about five minutes, we can assume an upper bound of $q = 10^5$ trials per captured user (this amounts to about one year of non-stop testing per user which will either interfere with the user's learned password rendering the user useless to the attacker, or alert security administrators due to the user's absence prompting a revocation of the credentials). Hence, even after capturing $u = 100$ users, the attacker's success probability is only

$$100 \times 10^5 / |\Sigma| \approx 2^{-16}.$$

Further complicating the attacker's life is the fact that subjecting a person to many random SISL games may obliterate the learned sequence or cause the person to learn an incorrect sequence thereby making extraction impossible.

We note that physical presence is necessary in authentication systems designed to resist coercion attacks. If the system supported remote authentication then an attacker could coerce a trained user to authenticate to a remote server and then hijack the session.

Security enhancements. The security model above gives the attacker one chance to authenticate and the attacker must succeed with non-negligible probability. If the attacker is allowed multiple authentication attempts — iterating the extraction and test phases, alternating between the two — then the protocol may become insecure. The reason is that during an authentication attempt the attacker sees the three sequences k_0, k_1, k_2 and could memorize one of them (30 symbols). He would then train offline on that sequence so that at the next authentication attempt he would have a 1/3 chance in succeeding. If the attacker could memorize all three sequences (90 symbols), he could offline subject a trained user to all three sequences and reliably determine which is the correct one and then train himself on that sequence. He is then *guaranteed* success at the next authentication trial. We note that this attack is non-trivial to pull off since

it can be difficult for a human attacker to memorize an entire sequence at the speed the game is played.

Another potential attack, already discussed in Section 3, is an attacker who happens to be an expert player, but deliberately degrades his performance on two of the sequences presented. With probability $1/3$ he will show a performance gap on the correct sequence and pass the authentication test. We described a number of defenses in Section 3. Here we describe a more robust defense.

Both attacks above can be defeated with combinatorics. Instead of training the user on a single sequence, we train the user on a small number of sequences, say four. Experiments [18] suggest that the human brain can learn multiple sequences and these learned sequences do not interfere with one another. Equivalently we could train the user on a longer sequence and use its fragments during authentication. While this will increase training time, we show that it can enhance security.

During authentication, instead of using one correct sequence and two foils, we use the four correct sequences randomly interspersed within 8 foils. Authentication succeeds if the attacker shows a measurable performance gap on the correct 4 out of 12 presented sequences. An attacker who slows down on random sequences will now have at most a $1/\binom{12}{4} \approx 1/500$ chance in passing the test. The number of trained sequences (4) and the number of foils (8) can be adjusted to achieve an acceptable tradeoff between security and usability.

Similarly, a small number of authentication attempts will not help a direct attacker pass the test. However, memorizing the authentication test (360 symbols) and later presenting it to a coerced user could give the adversary an advantage. To further defend against this memorization attack we add one more step to the authentication procedure: once the authentication server observes that the user failed to demonstrate a measurable gap on some of the trained sequences, all remaining trained sequences are replaced with random foils. This ensures that an attacker who tries to authenticate with no prior knowledge will not see all the trained sequences and therefore cannot extract all trained sequences from a coerced user. Consequently, a *one-shot* attack on a coerced user is not possible. Nevertheless, by iterating this process — taking the authentication test, memorizing the observed sequences, and then testing them out on a coerced trained user — the attacker may eventually learn all trained sequences and succeed in fooling the authentication test. During this process, however, the attacker must engage in the authentication test where he demonstrates knowledge of a strict subset of the trained sequences, but cannot demonstrate knowledge of all sequences. This is a clear signal to the system that it is under attack at which point the person engaging in the authentication test could

be detained for questioning and the legitimate user is blocked from authenticating with the system until he or she is retrained on a new set of sequences.

Eavesdropping security. Traditional password authentication is vulnerable to eavesdropping (either via client-side malware or shoulder surfing) and so is the authentication system presented here. An eavesdropper who obtains a number of valid authentication transcripts with a trained user will be able to reconstruct the learned sequence(s). It is a fascinating direction for future research to devise a coercion-resistant system where an implicitly learned secret is used in a challenge-response protocol with the server. We come back to this question at the end of the paper.

5.2 An Experiment: Extracting Sequence Fragments

One of the potential attacks on our system involves a malicious party profiling the legitimate user's knowledge and using that information to reverse engineer the trained sequence to be able to pass the authentication test. Although the number of possible trained sequences is too large to exhaustively test on any single individual each sequence is constructed according to known constraints and knowledge of subsequence fragments might enable the attacker to either reconstruct the original sequence or enough of it to pass an authentication test.

The training sequences are constrained to use all 6 response keys equally often, so analysis of individual response probabilities cannot provide information about the trained sequence. Likewise all 30 possible response key pairs ($6 * 5 = 30$, since keys are not repeated) occur equally often during training meaning that bigram frequency also provides no information about the trained sequence. However, each 30-item sequence has 30 unique trigrams (of 150 possible). If the specific training trigram fragments could be identified, the underlying training sequence could be reconstructed.

An attack based on this information would be to have a trained user perform a SISL test that contains all 150 trigrams equally often. If the user exhibited better performance on the 30 trained trigrams than the 120 untrained, the sequence could be reconstructed. This attack would weaken the method's relative resistance to external pressure to reveal the authentication information.

However, while the sequence information can be determined at the trigram level it is not known if participants reliably exhibit sequence knowledge in such short fragments. In Experiment 3, we evaluated performance on this type of trigram test to assess whether the sequence information could be reconstructed.

Participants were again recruited through Mechanical Turk and completed the same training sessions used in Experiments 1 and 2. At test, participants performed a sequence constructed to provide each of the 150 trigrams exactly 10 times by constructing ten different 150-trial units that each contain all possible trigrams in varying order. Performance on each trigram was measured by percent correct as a function of the current response and two responses prior.

To evaluate whether these data could be used to reconstruct the sequence, the percent correct on each trigram was individually calculated and a rank order of all trigrams was created for each individual. If performance on the trained trigrams was superior to others, the trained trigram ranks should tend to be lower (e.g., performance expression would lead the sequence trigrams to be the 30 best performed responses). However, average rank and average percent correct on the trained trigrams was indistinguishable from untrained trigrams. Participants did not exhibit their trained sequence knowledge on this type of test, indicating that their sequence knowledge cannot be attacked with a trigram-based method. More specifically, for each user we compared the average percent correct measurements for the 30 trained-sequence trigrams to those for the 120 remaining trigrams. The 34 participants averaged 73.9% correct (SE 1.2%) for trigrams from the trained sequence and 73.2% correct (SE 1.1%) for the rest. The difference was not reliable.

While the trigram test did not lead to expression of sequence knowledge, it is likely that participants' sequence knowledge could be assessed for some longer fragments. However, the number of fragments to assess grows exponentially with the length to be assessed and the ability to test all fragments is limited by the need to rely on human performance to do the assessment. For example, for length 4 fragments (quad-grams), there are 750 fragments to assess multiple times each to try to identify which ones had been trained.

Future work. In future work we will assess sequence expression at various lengths to find the minimal length at which sequence knowledge can be expressed. This minimal length likely reflects a basic operating characteristic of the brain regions that support implicit sequential skill learning. If this length suggests the possibility of attack, the sequence can be increased in complexity by increasing the number of characters, using inter-response timing (known to be important to learning [7]) or more complex sequence structures than simple repeating sequences.

Recall that in our experiments we assumed that users are honest during the training phase and the adversary only gets to coerce users after they have been trained.

We leave it for future work to design a coercion-resistant authentication protocol that remains secure when users can be coerced during the training phase.

6 Related Work

There is a large body of related work in user authentication and biometrics for user access control. The work can be broken down into biometrics (“who you are”), tokens (“what you have”), and passwords (“what you know”). There is significant past work in each of the three main areas. Our work may fall into a new category of implicit learning (“what you know you know but do not know”), or could be categorized as a subclass of behavioral biometric measurement.

Classic biometrics identifying a user based on who they physically are can be grouped into physiological and behavior categories. Physiological characteristics include fingerprint, face recognition, DNA prints, palm print, hand geometry, iris recognition, and retinal scans. Behavioral characteristics include measurements of typing rhythm and other dynamics, dynamic signature, walking gait, voiceprints, and eye movement patterns [11, 10, 2, 15]. Our work differs from these in enabling quick training in new randomly seeded patterns. It might be very difficult to learn to walk a new way, and nearly impossible to change one's iris pattern, but it should be easy to learn a new cortical crypto sequence with a modest training regime. Further, if one relies on retinal patterns for identification, each system could capture all the information content of the retina, and thus a single compromised retina reader could reveal to an adversary the entire set of information. Our approach enables key revocation and multiple keys per user for different systems where there need not be any information leakage from one system to the next.

Denning et al. [4] propose an authentication model based on implicit learning of sets of images. An earlier study [21] compared the learning of images, artificial words, and outputs from finite-state automata. Both of these works develop authentication systems that allow users to easily memorize strong passwords, however the resulting systems are not as resistant to rubber hose attacks because they depend on the user consciously studying sets of images or strings and as a result the user retains some conscious knowledge of the credential. When using the SISL task we were able to verify that little conscious knowledge of the trained secret is retained. Image-based authentication mechanisms also require curated image sets in order to reduce errors in the authentication process; in contrast SISL-based authentication uses automatically generated sequences sampled from a well-defined high entropy combinatorial space.

Deniable encryption. In the context of encryption, *deniable encryption* [3, 13] enables a user who encrypts a message to open the ciphertext in multiple ways to produce different cleartexts from the same ciphertext. Such systems enable a user to reveal an encryption key, which produces a document that contains plausible cleartext, but which is different from the actual document the user wishes to protect. This technique protects encrypted documents, but does not apply to authentication credentials. Further, a properly motivated user of deniable encryption could choose to reveal the correct decryption key, enabling the coercive adversary offline access to all versions of the document. Our approach develops a system where the user cannot, even if strongly motivated, reveal to another any information useful for an adversary to replicate the user's access without the user being present. Deniability has also been studied in the context of elections [9].

Coercion detection. Since our aim is to prevent users from effectively transmitting the ability to authenticate to others, there remains an attack where an adversary coerces a user to authenticate while they are under adversary control. It is possible to reduce the effectiveness of this technique if the system could detect if the user is under duress. Some behaviors such as timed responses to stimuli may detectably change when the user is under duress. Alternately, we might imagine other modes of detection of duress, including video monitoring, voice stress detection, and skin conductance monitoring [8, 16, 1]. The idea here would be to detect by out-of-band techniques the effects of coercion. Together with in-band detection of altered performance, we may be able to reliably detect coerced users.

7 Conclusions and Future Work

We have presented a new approach to protecting against coercion attacks using the concept of *implicit learning* from cognitive psychology. We described a proof of concept protocol and preliminary experiments conducted through Mechanical Turk demonstrating a basis for confidence that it is possible to construct rubber hose resistant authentication.

Much work remains. We hope to further analyze the rate at which implicitly learned passwords are forgotten, and the required frequency of refresher sessions. In addition we would like to find methods to detect or predict when individual users reliably learn (collecting more demographic data about our users might be a good first step in this direction, along with multi-session long-term experiments). We also hope to explore some of the limits of the approach, for example by finding out the minimum lengths at which parts of learned sequences are distin-

guishable to an attacker versus a legitimate authenticator, as well as by strengthening the test procedures and analysis to increase reliability across a larger fraction of users, or reduce the required testing time, false positives, and false negatives. Using variable timing between cues and measuring user performance as a function of game speed can further help in making the test protocol more reliable. Implicit learning of multiple credentials is yet another area that can benefit from additional experiments, building upon prior work that has so far found no evidence of interference when users learn distinct 12-item sequences, while also being capable of learning implicitly sequences as long as 80 items.

Another future direction for this work is in testing whether more complex structures—for example Markov models—can be learned implicitly. We would like to use such learning to build challenge-response authentication which is resistant to eavesdropping in addition to coercion. Finally, beyond authentication, we would like to investigate the construction of a variety of cryptographic primitives based on implicit learning.

Acknowledgment

We would like to thank all the paid volunteers who have contributed to our user studies through their participation. This work was funded by NSF and a MURI grant.

References

- [1] J. Benaloh and D. Tuinstra. Uncoercible communication. Technical Report TR-MCS-94-1, Clarkson University, 1994.
- [2] Christoph Bregler. Learning and recognizing human dynamics in video sequences. In *IEEE Conf. on Computer Vision and Pattern Recognition*, pages 568–574, 1997.
- [3] Ran Canetti, Cynthia Dwork, Moni Naor, and Rafail Ostrovsky. Deniable encryption. In *CRYPTO*, pages 90–104, 1997.
- [4] Tamara Denning, Kevin D. Bowers, Marten van Dijk, and Ari Juels. Exploring implicit memory for painless password recovery. In Desney S. Tan, Saleema Amershi, Bo Begole, Wendy A. Kellogg, and Manas Tungare, editors, *CHI*, pages 2615–2618. ACM, 2011.
- [5] A. Destrebecqz and A. Cleeremans. Can sequence learning be implicit? new evidence with the process dissociation procedure. *Psychonomic Bulletin & Review*, 8:343–350, 2001.

- [6] Julien Doyon, Pierre Bellec, Rhonda Amsel, Virginia Penhune, Oury Monchi, Julie Carrier, Stéphane Lehericy, and Habib Benali. Contributions of the basal ganglia and functionally related brain structures to motor learning. *Behavioural Brain Research*, 199(1):61–75, April 2009.
- [7] E. Gobel, D. Sanchez, and P. Reber. Integration of temporal and ordinal information during serial interception sequence learning. *Journal of Experimental Psychology: Learning, Memory & Cognition*, 37:994–1000, 2011.
- [8] Payas Gupta and Debin Gao. Fighting coercion attacks in key generation using skin conductance. In *USENIX Security Symposium*, pages 469–484, 2010.
- [9] Ari Juels, Dario Catalano, and Markus Jakobsson. Coercion-resistant electronic elections. In *Proceedings of the 2005 ACM workshop on Privacy in the electronic society*, WPES '05, pages 61–70, New York, NY, USA, 2005. ACM.
- [10] A. Kale, A.N. Rajagopalan, N. Cuntoor, V. Krueger, and R. Chellappa. Identification of humans using gait. *IEEE Transactions on Image Processing*, 13:1163–1173, 2002.
- [11] Fabian Monrose, Michael Reiter, and Susanne Wetzel. Password hardening based on keystroke dynamics. *Int. J. of Inf. Sec.*, 1(2):69–83, 2002.
- [12] Mary J. Nissen and Peter Bullemer. Attentional requirements of learning: Evidence from performance measures. *Cognitive Psychology*, 19(1):1–32, January 1987.
- [13] Adam O’Neill, Chris Peikert, and Brent Waters. Bi-deniable public-key encryption. In *Proc. of Crypto’11*, volume 6841 of *LNCS*, pages 525–542, 2011.
- [14] Paul Reber. Cognitive neuroscience of declarative and non-declarative memory. *Parallels in Learning and Memory*, Eds. M.Guadagnoli, M.S. deBelle, B. Emyre, T. Polk, A. Benjamin, pages 113–123, 2008.
- [15] Douglas A. Reynolds, Thomas F. Quatieri, and Robert B. Dunn. Speaker verification using adapted gaussian mixture models. In *Digital Signal Processing*, 2000.
- [16] Robert Ruiz, Claude Legros, and Antonio Guell. Voice analysis to predict the psychological or physical state of a speaker, 1990.
- [17] D. Sanchez, E. Gobel, and P. Reber. Performing the unexplainable: Implicit task performance reveals individually reliable sequence learning without explicit knowledge. *Psychonomic Bulletin & Review*, 17:790–796, 2010.
- [18] D.J. Sanchez and P.J. Reber. Operating characteristics of the implicit learning system during serial interception sequence learning. *Journal of Experimental Psychology: Human Perception and Performance*, in press.
- [19] Chris Soghoian. Turkish police may have beaten encryption key out of TJ Maxx suspect, 2008. news.cnet.com/8301-13739_3-10069776-46.html.
- [20] T. van Aardenne-Ehrenfest and N. G. de Bruijn. Circuits and trees in oriented linear graphs. *Simon Stevin*, 28:203–217, 1951.
- [21] Daphna Weinshall and Scott Kirkpatrick. Passwords you’ll never forget, but can’t recall. In *CHI Extended Abstracts*, pages 1399–1402, 2004.
- [22] Wikipedia. Rubber-hose cryptanalysis, 2011.

On the Feasibility of Side-Channel Attacks with Brain-Computer Interfaces

Ivan Martinovic*, Doug Davies[†], Mario Frank[†], Daniele Perito[†], Tomas Ros[‡], Dawn Song[†]
*University of Oxford** *UC Berkeley[†]* *University of Geneva[‡]*

Abstract

Brain computer interfaces (BCI) are becoming increasingly popular in the gaming and entertainment industries. Consumer-grade BCI devices are available for a few hundred dollars and are used in a variety of applications, such as video games, hands-free keyboards, or as an assistant in relaxation training. There are application stores similar to the ones used for smart phones, where application developers have access to an API to collect data from the BCI devices.

The security risks involved in using consumer-grade BCI devices have never been studied and the impact of malicious software with access to the device is unexplored. We take a first step in studying the security implications of such devices and demonstrate that this upcoming technology could be turned against users to reveal their private and secret information. We use inexpensive electroencephalography (EEG) based BCI devices to test the feasibility of simple, yet effective, attacks. The captured EEG signal could reveal the user's private information about, e.g., bank cards, PIN numbers, area of living, the knowledge of the known persons. This is the first attempt to study the security implications of consumer-grade BCI devices. We show that the entropy of the private information is decreased on the average by approximately 15% - 40% compared to random guessing attacks.

1 Motivation

Brain-Computer Interfaces (BCIs) enable a non-muscular communication between a user and an external device by measuring the brain's activities. In the last decades, BCIs have been primarily applied in the medical domain with the goal to increase the quality of life of patients with severe neuromuscular disorders. Most BCIs are based on electroencephalography (EEG) as it provides a non-invasive method for recording the elec-

trical fields directly produced by neuronal synaptic activity. The EEG signal is recorded from scalp electrodes by a differential amplifier in order to increase the Signal-to-Noise Ratio of the electrical signal that is attenuated by the skull. This signal is continuously sampled (typically 128 Hz - 512 Hz) to provide a high temporal resolution, making EEG an ideal method for capturing the rapid, millisecond-scale dynamics of brain information processing with a simple setup.

Particular patterns of brain waves have been found to differentiate neurocognitive states and to offer a rich feature space for studying neurological processes of both disabled and healthy users. For example, EEG has not only been used for neurofeedback therapy in attention deficit hyperactivity disorder (ADHD) [20], epilepsy monitoring [6], and sleep disorders [28], but also to study underlying processes of skilled performance in sports and changes in vigilance [14, 31], in estimating alertness and drowsiness in drivers [22] and the mental workload of air-traffic control operators [39].

Besides medical applications, BCI devices are becoming increasingly popular in the entertainment and gaming industries. The ability to capture a user's cognitive activities enables the development of more adaptive games responsive to the user's affective states, such as satisfaction, boredom, frustration, confusion, and helps to improve the gaming experience [26]. A similar trend can be seen in popular gaming consoles such as Microsoft's Xbox 360, Nintendo's Wii, or Sony's Playstation3, which already include different sensors to infer user's behavioral and physiological states through pressure, heartbeat, facial and voice recognition, gaze-tracking, and motion.

In the last couple of years, several EEG-based gaming devices have made their way onto the market and became available to the general public. Companies such as Emotiv Systems [5] and NeuroSky [25] are offering low-cost EEG-based BCI devices (e.g., see Figure 1) and software development kits to support the expansion of



(a) An EPOC device (Emotiv Systems)



(b) A MindSet device (NeuroSky)

Figure 1: Popular consumer-grade BCI devices are available as multi-channel (EPOC) or single-channel (MindSet) wireless headsets using bluetooth transmitters.

tools and games available from their application markets. Currently, there are more than 100 available applications ranging from accessibility tools, such as a mind-controlled keyboard and mouse and hands-free arcade games, to so-called serious games, i.e., games with a purpose other than pure entertainment, such as attention and memory training games. For example, in [2], the authors used the Emotiv BCI device to implement a hands-free brain-to-mobile phone dialing application.

Marketing is another field that has shown increasing interest in commercial applications of BCI devices. In 2008, The Nielsen Company (a leading market research company) acquired NeuroFocus, a company specialized in neuroscience research, and it has recently developed an EEG-based BCI device called Mynd such that “...market researchers will be able to capture the highest quality data on consumers’ deep subconscious responses in real time wirelessly, revolutionizing mobile in-market research and media consumption at home.”¹

In light of the progress of this technology, we believe that the trend in using EEG-based BCI devices for non-medical applications, in particular gaming, entertainment, and marketing, will continue. Given that this technology provides information on our cognitive pro-

¹NeuroFocus Press Release (March 21, 2011): www.neurofocus.com/pdfs/Mynd_NeuroFocus.pdf



Figure 2: Example photo of a videogame controlled with the Emotiv Device.

cessing and allows inferences to be made with regard to our intentions, conscious and unconscious interests, or emotional responses, we are concerned with its security and privacy aspects. More specifically, we are interested in understanding how easily this technology can be turned against its users to reveal their private information, that is, information they would not knowingly or willingly share. In particular, we investigate how third-party EEG applications could infer private information about the users, by manipulating the visual stimuli presented on screen and by analyzing the corresponding responses in the EEG signal.

1.1 Contributions

To justify how crucial the security and privacy concerns of this upcoming technology are, we provide some concrete answers in terms of demonstrating practical attacks using existing low-cost BCI devices. More specifically, the main contributions of this paper are:

- We explore, for the first time, EEG gaming devices as a potential attack vector to infer secret and private information about their users. This attack vector is entirely unexplored and qualitatively different from previously explored side-channels. This calls for research to analyze their potential to leak private information before these devices gain widespread adoption.
- We design and implement BCI experiments that show the possibility of attacks to reveal a user’s private and secret information. The experiments are implemented and tested using a Emotiv EPOC BCI device. Since 2009, this consumer-grade device has been available on the market for the entertainment and gaming purposes.

- In a systematic user study, we analyze the feasibility of these attacks and show that they are able to reveal information about the user’s *month of birth, area of living, knowledge of persons known to the user, PIN numbers, name of the user’s bank, and the user’s preferred bank card.*

2 A Brief Introduction to P300 Event-Related Potentials

In this section, we provide a brief introduction to the specifics of the EEG signal that are required to understand the rationale behind this work.

An important neurophysiological phenomenon used in studies of EEG signals is the Event-Related Potential (ERP). An ERP is detected as a pattern of voltage change after a certain auditory or visual stimulus is presented to a subject. Every ERP is time-locked to the stimulus, i.e., the time frame at which an EEG voltage change is expected to occur is known given the timing of the stimuli.

The most prominent ERP component which is sensitive to complex cognitive processing is the P300, so-called because it can be detected as an amplitude peak in the EEG signal at ≈ 300 ms after the stimulus (see Figure 3). The complexity of the stimulus and individual differences contribute to the variability of the amplitude and latency (e.g., the latency varies between 250 - 500 ms), yet the P300 is considered to be a fundamental physiological component and is reliably measured (for a recent overview of the P300 from a neuroscience perspective, please see, e.g., [27]). While there are two sub-components of the P300, called P3a and P3b, both are related to complex cognitive processing, such as recognition and classification of external stimuli. In this paper, we take advantage of the subcomponent P3b of the P300, and for the sake of simplicity we will refer to it as the P300, which is also a convention in neuroscience.

The P300 is elicited when subjects discriminate between task-relevant and task-irrelevant stimuli using a so-called “oddball paradigm” (for more information, see, e.g., [16]). During an oddball task the number of task-relevant stimuli (called *target* stimuli) is less frequent than the number of task-irrelevant stimuli (called *non-target*). Probably the most well-known application of the P300 in an oddball task is the P300-Speller. In this application the alphanumeric characters are arranged in a matrix where rows and columns flash on the screen in a rapid succession. The target stimulus is the character that a subject desires to spell and the P300 is evoked each time the target letter is flashed due to a neuronal response triggered by increased attention of recognition. This application has been used to establish a communication channel for patients with locked-in syndrome or

with severe neurodegenerative disorders.

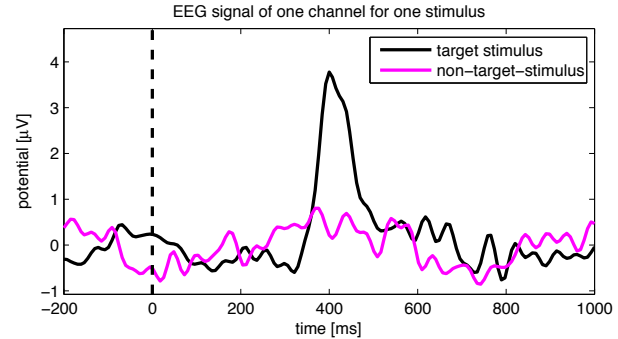


Figure 3: The P300 ERP elicited as a brain response to a target stimuli (in this experiment the non-target stimuli were pictures of unknown faces, while the target stimuli was the picture showing President Obama).

The P300 is seen in response to target stimuli defined by the task, but it has also been observed to be elicited during stimuli that are personally meaningful to participants. For example, if a random sequence of personal names is presented to a subject, the P300 will be the largest during the presentation of the subject’s own name [32]. Likewise, it has been shown that the P300 discriminates familiar from unfamiliar faces within randomly presented sequences [24].

3 BCI Attacks: Threat Model and Assumptions

In this section, we explore a number of possible scenarios in which consumer EEG devices could be abused to capture sensitive or private information from users. Currently, both Emotiv and NeuroSky have “App Stores” where the users can download a wide variety of applications. Similarly to application stores for smart phones, the applications are developed by third parties that rely on a common API to access the devices. In the case of the EEG devices, this API provides unrestricted access to the raw EEG signal. Furthermore, such applications have complete control over the stimuli that can be presented to the users.

In this scenario, the attacker is a malicious third-party developer of applications that are using EEG-based BCI devices. Its goal is to learn as much information as possible about the user. Hence, we are neither assuming any malware running on the machine of the victim nor a tampered device, just “brain spyware”, i.e., a software intentionally designed to detect private information. Our attacker model cannot access more computer resources than any third party application for the respective BCI device. The attacker can read the EEG signal from the

device and can display text, videos, and images on the screen. Therefore, the attacker can specifically design the videos and images shown to the user to maximize the amount of information leaked while trying to conceal the attacks.

The type of information that could be discovered by such an attack is only bound by the quality of the signal coming from the EEG device and the techniques used to extract the signal. We note that all involved parties (users of BCI devices, their developers, and also attackers) share the same objective: to maximize the signal quality in order to best perform their task. Hence, it is expected that the signal and the measurement processes will improve and, as a result, facilitate the attacks.

In this work we will focus on categorization tasks, in which the mind of the user is probed to detect whether certain stimuli (faces, banks, locations) are familiar to or relevant for the user. However, we note that in the future such attack could be extended to include other sensitive information. For instance, EEG devices have been used, under optimized lab conditions, to study prejudices, sexual orientation, religious beliefs [18], and deviant sexual interests [38, 10].

At the moment, low-cost devices are still very noisy and need a calibration phase to work properly (three minutes in our experiments). However, we note that the attacker could find a natural situation in which to expose the user to target stimuli to extract information and thus gather enough data to succeed in an unnoticed way. Also, such a calibration phase can be concealed in the normal training phase that EEG applications require for proper functioning and that the user is willing to support. Moreover, we expect that BCI devices will become increasingly robust and accurate in the future, resolving many current technical problems.

The experiments presented in this study are meant to show feasibility in favorable conditions. The subjects were partially cooperating in an attack situation and were following our instructions. However, we minimized the interaction between the supervisor and subjects to simulate a realistic environment, where a user is only interacting with his computer (see Appendix A).

4 Experimental Design and Results

The main question, which this paper attempts to answer is: *Can the signal captured by a consumer-grade EEG device be used to extract potentially sensitive information from the users?* In the following, we detail the technical setup, the experimental design, and the analytical methods of our experiments.

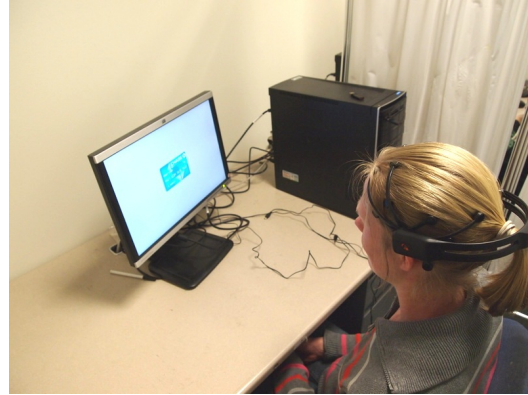


Figure 4: Experimental setup. The instructor sits behind the curtain to minimize interaction during the experiments. In this case, a sequence of credit cards is presented to the user.

4.1 The Setup

After obtaining the approval of the Institutional Review Board (IRB), we recruited 30 Computer Science students for the experiments. For two participants, the experiments could not be conducted due to faulty equipment (low battery on the EEG device). Of the 28 participants remaining, 18 were male and 10 female. In total, the experiment lasted about 40 minutes. The participants were informed that they were going to participate in an experiment involving the privacy implications of using gaming EEG devices, but we explained neither the details of the experiment nor our objectives. Each participant was seated in front of the computer used for the experiments (see Figure 4). The operator then proceeded to mount the Emotiv EEG device on the participants.

4.2 The Protocol

After the initial setup, the participants were asked to try to remain relaxed for the entire duration of the experiments, as blinking or other face movements cause significant noise. The exact script used during the experiments can be found in Appendix A. The interaction with the participants was kept as short and concise as possible. The order of the experiments was kept fixed in the order found in Appendix A.

Each experiment consisted of three main steps:

1. (Optional) Brief verbal explanation of the task by the operator;
2. (Optional) Message on screen for 2 seconds;
3. Images being flashed in random order for the duration of the experiment.

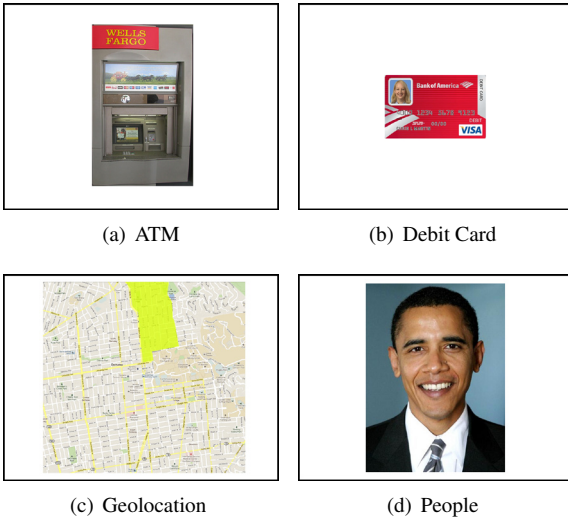


Figure 5: Layout of four of the experiments: Bank ATMs, Debit Cards, Geolocation and People. Each frame shows how the stimuli were flashed on the screen.

Each image was shown to the users for a fixed duration of 250ms. On the screen in Figure 4, a photo is being shown to a test participant.

The time of the target and non-target stimuli and the stimulus identifiers were recorded alongside the raw signal coming from the EEG device. After the experiment, we used the classification techniques detailed in Section 4.4 to infer information about the secrets of the participant.

4.3 The Experimental Scenarios

In this section, we describe the calibration of the device and six different experiments. In each experiment, the attacker tries to gain information about a different secret. Each experiment lasted approximately 90 seconds.

4.3.1 Training Phase

This experiment was set up to learn a model to detect the P300 signal from each user. The users were presented with a randomly permuted sequence of numbers from 0 to 9 and were asked by the operator to count the number of occurrences of a target number x . Each number was shown 16 times, with a stimulus duration of 250 ms and a pause between stimuli randomly chosen between 250 ms and 375 ms. At the end of experiment the participants were asked for their count to check for correctness.

We also developed a method to calibrate the classifier without this active training phase. This could be used for a concealed attack in cases where the intended application of the user does not require the detection of P300. We explain this on-the-fly calibration phase in Section 5.

4.3.2 Experiment 1: Pin Code

This experiment has the goal to gather partial information about a user’s chosen 4-digit PIN. Given the sensitivity in studying the users’ real PINs, we asked the participants to choose and memorize a randomly generated PIN just for the experiment. Furthermore, the participants were asked not to reveal the PIN until after the end of the experiment session. The participants were told that there were no special instructions for the experiment, e.g., no counting numbers. They were just informed that, at the end of the experiment, they would be asked to enter the first digit of their PIN (refer to Appendix A for the exact script). In this way, we bring the information of interest to the attention of the user which makes the subject focus on the desired stimulus without requiring their active support of the classifier. After the instructions were given, the operator started the experiment. There was *no* on-screen message shown at the beginning of the experiment. The experiment images consisted of a sequence of randomly permuted numbers between 0 and 9 that were shown on the screen one by one. Each number was shown 16 times and the experiment lasted approximately 90 seconds.

4.3.3 Experiment 2: Bank Information

The aim of this experiment was to obtain the name of the bank of the participant by reading their response to visual stimuli that involved photos related to banks. The first iteration of this experiment, whose results are not reported, consisted of showing the logo of 10 different banks². The intuition was that the participants would show a higher response when seeing the logo of their bank. However, this attack was unsuccessful. After de-briefing with the early test participants, we realized that they simply recognized the logos of all the banks.

In the second and final iteration of the experiment, we showed two different sets of images: automatic teller machines (ATMs) and credit cards. Rationale for choosing to display ATM or credit card photos, rather than logo images, is that while users might be familiar with all logos, they might be only familiar with the look of their own local bank ATM and debit card. The results are reported in Section 5.

The protocol for this experiment was as follows. Each participant was asked by the operator whether they were a customer of one of the banks in a list. Four participants answered negatively, therefore the experiment was skipped. In case of an affirmative answer, the experiment was started. The screen in front of the participants showed the question “What is the name of your bank?”

²List of banks: Bank of America, Chase, Wells Fargo, ING, Barclays, Citi Bank, Postbank, Unicredit, Deutsche Bank



Figure 6: Stimuli for the debit card experiment. Each card was shown separately, full-screen, for the short stimulus duration.

for 2 seconds. Then, for the ATM experiment, images of teller machines were flashed on the screen. For the credit card experiment, images of credit cards were flashed.

4.3.4 Experiment 3: Month of Birth

The operator did not give any specific instructions to the participants and only informed them that the instructions would be provided on the screen. The participants were simply asked in which month they were born by an on-screen message that lasted for 2 seconds, then, a randomly permuted sequence of the names of the months was shown on the screen.

In many access-restricted websites the date of birth or similar information serves as a backup function for resetting a user's password. If an attacker needs this information, the BCI device could provide a potential attack vector.

4.3.5 Experiment 4: Face Recognition

For this experiment, the operator again did not give any specific instruction to the participants and only informed them that the instructions would be provided on the screen. The participants were simply asked "Do you know any of these people?" by an on-screen message that lasted 2 seconds. Then the images of people were randomly flashed for the duration of the experiment.

The goal of this experiment was to understand whether we could infer who the participants knew by reading their EEG response when being showed a sequence of photos of known and unknown people. We used photos of 10 unknown persons and one photo of the current President of the United States of America, Barack Obama. The photo of the president was chosen because, being in a US institution, we were confident that each participant would recognize the President.

One interesting application of such an attack would be scenarios in which the knowledge of particular individual is used as a form of authentication. For example, in recent years, Facebook has started showing photos of friends for the purpose of account verification³.

4.3.6 Experiment 5: Geographic Location

The purpose of this experiment was to accurately pinpoint the geographic location of the residence of the participants. Each participant was asked if they lived in an area close to campus. Eight participants in total did not live close to campus and did not complete this experiment. In case of an affirmative answer, the participants were shown a sequence of highlighted maps of an area of approximately 4 square kilometers around campus. Each image showed the same area overall, but with a different highlighted zone on the map.

While IP addresses provide a rather accurate way to localize the location of a user, there are cases in which the users actively try to hide their geographic location using proxies. Even though our experiment showed only a predefined map of a rather small geographic area, we envision possible future attacks in which the true geographic location of a user is leaked by showing maps or landmarks with increased accuracy.

While for all the other experiments we did not instruct the user to do particular things except for watching the screen, here we asked the users to count how often their region was highlighted. This experiment was devised to study the influence of active user support, as counting assures a higher attention from the user which is known to improve the detection of P300.

4.4 Analysis Methodology

In this section, we detail how the attacker processes and analyzes the data and provide the specification of the data recorded by the BCI device.

Data characteristics and acquisition The data consists of several parts. The amplitudes of the EEG signal are recorded with 14 different electrodes. Each electrode represents one 'channel' of the signal. According to the standard 10-20 system [19], the 14 channels are called 1: 'AF3', 2: 'F7', 3: 'F3', 4: 'FC5', 5: 'T7', 6: 'P7', 7: 'O1', 8: 'O2', 9: 'P8', 10: 'T8', 11: 'FC6', 12: 'F4', 13: 'F8', and 14: 'AF4'. The location of the channel electrodes can be seen in Figure 7.

Each channel is recorded at a sampling rate of 128Hz. The software for showing stimuli to the user outputs the time stamp for each stimulus and the indicator of the

³<http://www.facebook.com/help/search/?q=security+verification>

stimulus. In this way, the EEG signal can be related to the stimuli.

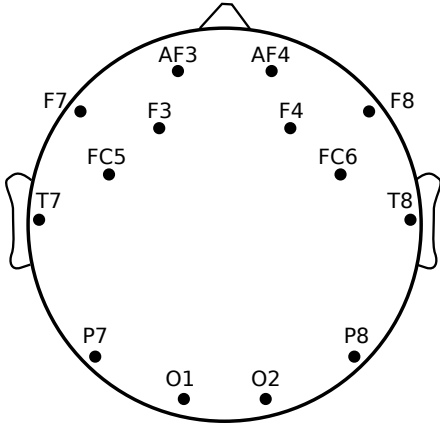


Figure 7: Position of the electrodes of the EPOC device.

As explained in Section 3, our attack vector exploits the occurrence of P300 peaks in the EEG signal triggered by target-stimuli. This requires the attacker to be able to reliably detect these peaks and to discriminate them from all other EEG signals measured on non-target stimuli. This task is very similar to the P300-Speller, where the EEG signal for the intended letter must be discriminated from the signal of unintended letters (as described in Section 2). However, in contrast to the spelling scenario the attacker is dealing with a passive user. This makes an attack much harder than spelling. In our case, the user does not *intend* to provide a discriminative signal for the target stimulus. This means that the user does not support the classifier with increased attention on the target stimulus, as can be achieved, for instance, by counting the number of occurrences of this stimulus. As a consequence, the data available to the attacker is less discriminative between target and non-target stimuli than in the spelling scenario.

An additional challenge for our attack is that the gaming device we are using is not made for detecting P300. For instance, they have more electrodes on the frontal part of the scalp (see Figure 7). This enables them to recognize facial expressions which provide a stronger signal than the EEG signal itself and thus are more robust for controlling games. The P300 is mostly detected at the parietal lobe, optimally with electrodes attached at Pz position, which is a centered on the median line at the top of the head. As we want to investigate the attack in a realistic home-use scenario we did not use other devices optimized for P300 detection and did not adapt the gaming device (for instance by turning it around, which would provide more sampling points in the Pz area).

Classification of target stimuli Detecting P300 in EEG data is a binary classification task. The input is a set of **epochs**. Each epoch is associated with a stimulus. In our setting a stimulus is an image depicted on a computer screen in front of the user. Let n_c be the number of EEG channels and let f be the sampling rate of the device (in our case the signal is sampled with 128 Hz). An epoch consists of n_c time series starting t_p milliseconds prior to the stimulus and ending t_a milliseconds after the stimulus. The number of measurements per time series is then $q = (t_p + t_a)f$. Typically, t_p is a few hundred milliseconds and t_a is between 800 ms and 1500 ms. The signals of all channels are concatenated and each epoch is represented as a real vector $\mathbf{x} \in \mathbb{R}^p$, where $p = qn_c$ is the dimensionality of the vector space.

The classification task consists of two phases, the training phase and the classification phase. The input of the training phase is a set of epochs $\mathbf{X}^{\text{tr}} = \{\mathbf{x}_i^{\text{tr}} \in \mathbb{R}^p, i = 1 \dots n_1\}$ and a vector of labels $\mathbf{y} \in \{0, 1\}^{n_1}$, where each label y_i indicates whether the epoch \mathbf{x}_i^{tr} corresponds to a target stimulus ($y_i = 1$) or not ($y_i = 0$). The signal of each epoch has been recorded while the corresponding stimulus was shown to the user on the screen for a short time (we used 500 ms). The stimuli labels \mathbf{y} are known to the classifier as the system knows what it shows to the user. Given this input, the classifier must learn a function g that maps epochs to target stimuli labels:

$$\begin{aligned} g : \mathbb{R}^p &\rightarrow \{0, 1\} \\ \mathbf{x} &\mapsto y \end{aligned} \quad (1)$$

In the beginning of Section 5, we explain how to practically carry out the training phase with users that actively support this training phase and with passive users.

In the classification phase the classifier gets a collection of n_2 new epochs $\mathbf{X}^{\text{test}} = \{\mathbf{x}_i^{\text{test}} \in \mathbb{R}^p, i = 1, \dots, n_2\}$ as an input and must output an estimate $\hat{\mathbf{y}} = \{\hat{y}_i = g(\mathbf{x}_i^{\text{test}}), i = 1, \dots, n_2\}$ of the corresponding labels. This means, for each of the new epochs, the classifier must decide whether the epoch is associated with the target stimulus or not.

The test labels $\hat{\mathbf{y}}$ provide a ranking of the K unique stimuli presented to the user. We sort all stimuli in descending order according to the number of their positive classifications. For stimulus k this number is $N_k^{(+)} = \sum_{i \in E_k} \hat{y}_i$. The set E_k is the set of epoch indices containing all epochs that are associated with stimulus k . In this notation $i \in E_k$ means that we sum over all epochs of stimulus k . For instance, if there are three different stimuli repeatedly shown to the user in random order (three different faces, say), then the classifier would guess that the true face (the one familiar to the user) is the face where the most associated epochs have been classified as

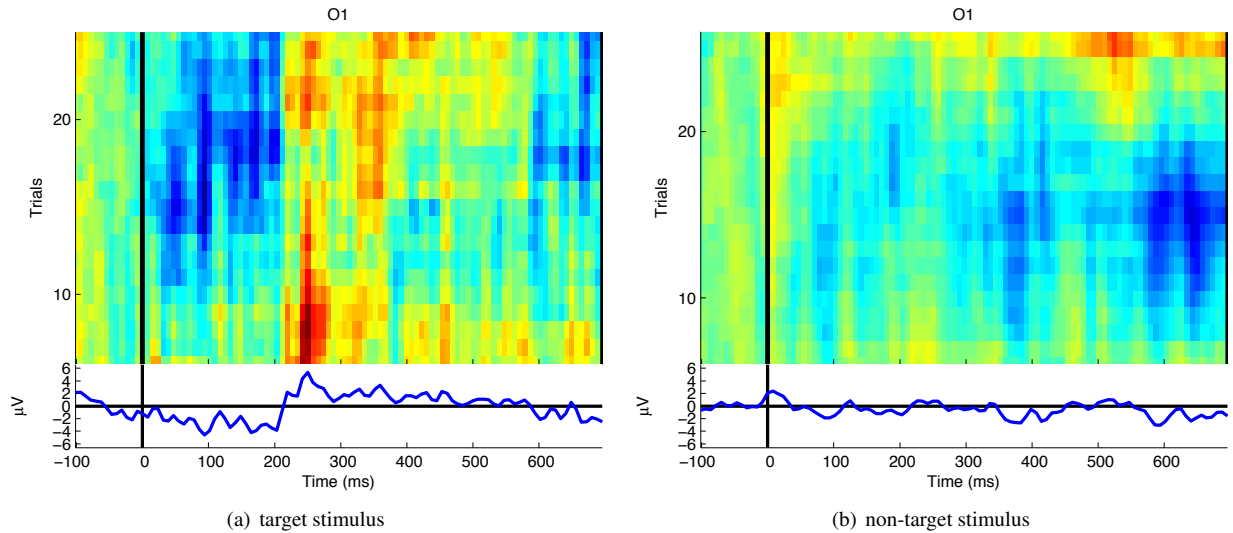


Figure 8: Event-related potentials for two different stimuli. Both signals have been recorded on the left back-side of the scalp (Channel 7: ‘O1’). The plots have been produced with EEGLab [4]. The scale of the averaged plots (bottom) as well as the colorscale of the heatmap plots (top) are constant over the two stimuli.

target-stimulus. Figure 8 depicts event-related potentials (ERP) for one channel and two different stimuli (target and non-target). In this example one row of one plot represents an epoch and all rows of one plot constitute the set E_k of epochs associated with event k .

The stimulus with the topmost positive classifications is the estimated target-stimulus, the stimulus with the second most positively classified epochs is ranked second, and so on. Most classifiers output a continuous score s_i for each epoch instead of binary labels \hat{y}_i . For instance, this could be a probability $s_i = p(y_i = 1)$. In such a case, we sum over all scores of each unique stimulus k to get its vote $N_k^{(+)} = \sum_{i \in E_k} s_i$. In the experiments, we will use this ranking to decide which of the presented stimuli is the target stimulus, that is which of the answers is the true answer for the current user.

In the following we explain two different classifiers that we used in our experiments. The first classifier is a boosting algorithm for logistic regression (bLogReg) and was proposed for P300 spelling in [17]. The second classifier is the publicly available BCI2000 P300 classifier. BCI2000 uses stepwise linear discriminant analysis (SWLDA). In [21] a set of different P300 classifiers, including linear and non-linear support vector machines, was compared and SWLDA performed best.

4.4.1 Boosted logistic regression

This method uses a logistic regression model as the classifier function g . The model is trained on the training data by minimizing the negative Bernoulli log-likelihood

of the model in a stepwise fashion as proposed in [11, 12].

As follows, we briefly describe a variant, proposed in [17], where the method has been used to design a P300 speller. The classifier consists of an ensemble of M weak learners. Each weak learner f_m is a regression function minimizing a quadratic cost function:

$$f_m = \operatorname{argmin}_f \sum_{i=1}^{n_1} (\tilde{y}_i - f(\mathbf{x}_i^{\text{tr}}; \mathbf{w}))^2, \quad (2)$$

where $f(\mathbf{x}_i^{\text{tr}}; \mathbf{w}) = \mathbf{w}^T \mathbf{x}_i^{\text{tr}}$ with coefficients $\mathbf{w} \in \mathbb{R}^p$. The score \tilde{y}_i in Equation (2) is obtained from the first-order condition of maximizing the logarithm of the Bernoulli likelihood

$$L(g_m; \mathbf{X}^{\text{tr}}, \mathbf{y}) = \prod_{i=1}^{n_1} p(y_i = 1 | \mathbf{x}_i^{\text{tr}})^{y_i} (1 - p(y_i = 1 | \mathbf{x}_i^{\text{tr}}))^{1-y_i} \quad (3)$$

with

$$p(y_i = 1 | \mathbf{x}_i^{\text{tr}}) = \frac{\exp(g_m(\mathbf{x}_i^{\text{tr}}))}{\exp(g_m(\mathbf{x}_i^{\text{tr}})) + \exp(-g_m(\mathbf{x}_i^{\text{tr}}))} \quad (4)$$

In step m of the algorithm, the current classifier g_{m-1} is updated by adding the new weak classifier f_m : $g_m = g_{m-1} + \gamma_m f_m$. Thereby, the weight γ_m is selected such that the likelihood Eq. (3) is maximized.

The number of weak classifiers M controls the trade-off between overfitting and underfitting. This number is determined by cross-validation on random subsets of the training data \mathbf{X}^{tr} .

Data preprocessing Before training the classifier and prior applying it to each new observation, we process the data in the following way. The input data consists of n_c different time series, whereas n_c is the number of channels. First we epochize the signal with a time frame around the stimuli that starts 200 ms before the respective stimulus and ends 1000 ms after the stimulus. Then, for each epoch, we subtract the mean amplitude of the first 200 ms from the entire epoch as it represents the baseline.

In order to reduce the high-frequency noise, we apply a low-pass FIR filter with a pass band between 0.35 and 0.4 in normalized frequency units. An example of such a preprocessed signal is depicted in Figure 3.

4.4.2 Stepwise Linear Discriminant Analysis

The BCI2000 P300 classifier uses stepwise linear discriminant analysis, an extension of Fisher’s linear discriminant analysis. As follows, we briefly explain these two methods.

Fisher’s linear discriminant analysis (LDA) LDA was first proposed in [9]. This classifier is a linear hyperplane that separates the observations of the two classes. The hyperplane is parameterized by the coefficient vector $\mathbf{w} \in \mathbb{R}^p$ which is orthogonal to the hyperplane. A new observation \mathbf{x}_i is labeled to belong to either of the two classes by projecting it on the class separation $\mathbf{w}^T \mathbf{x}_i$. LDA assumes observations in both classes to be Gaussian distributed with parameters $(\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j), j = 1, 2$ and computes the optimally separating coefficients by $\mathbf{w} = (\boldsymbol{\mu}_1^t - \boldsymbol{\mu}_2^t)(\boldsymbol{\Sigma}_1^t + \boldsymbol{\Sigma}_2^t)^{-1}$.

Stepwise Linear Discriminant Analysis (SWLDA)

SWLDA extends LDA with a feature selection mechanism that sets many of the coefficients in \mathbf{w} to zero. This classifier is supposedly more robust to noise and was first applied to P300 spelling in [7]. The algorithm iteratively adds or removes components of the coefficient vector according to their statistical significance for the label outcome as measured by their p-value. The thresholds $(p_{\text{add}}, p_{\text{rem}})$ for adding or removing features as well as the total number of features must be pre-defined.

In our experiments we used the default configuration of the the BCI2000 P300 classifier with 60 features and $(p_{\text{add}}, p_{\text{rem}}) = (0.1, 0.15)$. The algorithm uses the 800 ms period after the stimulus for classification.

For each stimulus presented, we sum up the scores $\mathbf{w}^T \mathbf{x}_i$ of the corresponding epochs in order to obtain a ranking of the stimuli. Then, the highest ranked stimulus is presumably the target-stimulus.

5 Results

In this section, we evaluate the classification results on each of the experiments described in Section 4.3.

User-supported calibration and on-the fly calibration

We calibrate the classifiers on a set of training observations. Thereby, we distinguish two training situations.

In the first situation we have a partially cooperating user, that is, a user who actively supports the training phase of the BCI but then does not actively provide evidence for the target stimulus later. This is a realistic scenario. Each gamer has a strong incentive to support the initial calibration phase of his device, because he will benefit from a high usability and a resulting satisfying gaming experience. The attacker can use the training data to train his own classifier. Despite the user supporting the calibration phase, we do not assume that the user actively supports the detection of target stimuli when the attacker later carries out his attack by suddenly presenting new stimuli on the screen.

In the second training situation, the user is passive. This means that the user does not support the training phase but also does not actively try to disturb it. As a consequence, the attacker must present a set of stimuli where, with high probability, the user is familiar with one of the stimuli and unfamiliar with all other stimuli. In this way the attacker can provide a label vector $\mathbf{y} \in \{0, 1\}^{n_1}$ that can be used for training. We used the people experiment as training data. We showed 10 images of random people to the user as well as one image of President Barack Obama. Assuming that i) every user knows Obama and that ii) it is unlikely that a user knows one of the random face images downloaded from the internet, we can use the Obama image as a target stimulus and the others as non-target stimuli.

Success statistics We report the results of all experiments in Figure 9. Each plot corresponds to one experimental scenario. The black crosses depict the results of the SWLDA classifier used by the BCI2000 P300 speller. The red diamonds are the results of boosted logarithmic regression (bLogReg) trained by the counting experiment, and the blue crosses show the results for bLogReg when trained on the people experiment. The dashed black line depicts the expected result of a random guess.

We depict the results in terms of a cumulative statistic of the rank of the correct answer. This measure provides the accuracy together with a confidence interval at the same time as it includes the probability distribution of the deviation from the optimal rank. The plots read as follows. The x -axis of each plot is the rank of the correct

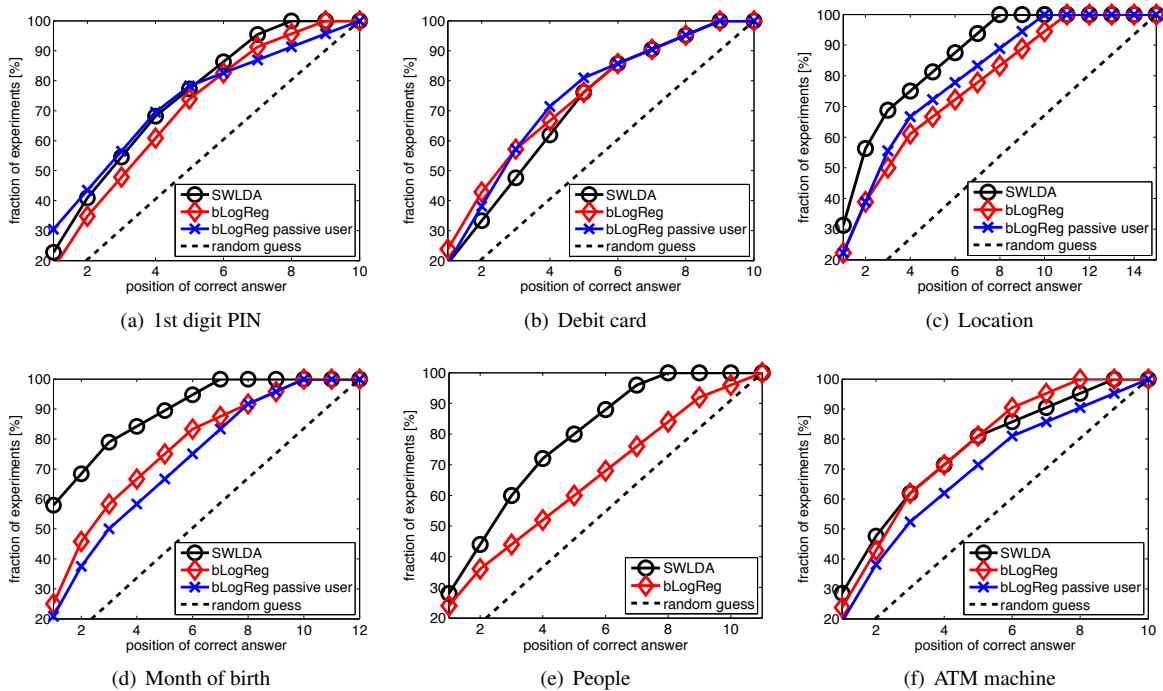


Figure 9: Cumulative statistics of the ranking of the correct answer according to the classification result. The faster this measure converges towards 100%, the better the classifier. One can directly read the confidence intervals as follows: In more than 20% of the experiments the bLogReg classifier ranked the correct face at the first position. In more than 40% it had the correct face among the first three guesses. Please note that for the passive user, the classifier was trained on the people experiment and the corresponding curve in Fig. 9(e) would depict the training error.

answer as estimated by the respective classifier. For instance, if the correct answer in the month of birth experiment is ‘April’ and the classifier ranks this month at the third position in the classification output, then x is 3. The y -axis is the fraction (in %) of the users having the correct answer in **at most** ranking position x . In our example with the month of birth, the point $(x; y) = (3; 80\%)$ of the SWLDA classifier means that for 80% of the users the correct bank was among the first three guesses of SWLDA. Please note that we truncated the y -axis at 20% to get a better resolution of the dynamic range.

Overall, one can observe that the attack does not always reveal the correct information on the first guess. However, the classifiers perform significantly better than the random attack. The SWLDA classifier provided the most accurate estimates, except for the experiment on the PIN and the debit card.

The correct answer was found by the first guess in 20% of the cases for the experiment with the PIN, the debit cards, people, and the ATM machine. The location was exactly guessed for 30% of users, month of birth for almost 60% and the bank based on the ATM machines for almost 30%. All classifiers performed consistently good on the location experiment where the users actively

concentrated by counting the occurrence of the correct answer. SWLDA performed exceptionally good on the month of birth experiment, even though this experiment was carried out without counting.

Relative reduction of entropy In order to quantify the information leak that the BCI attack provides, we compare the Shannon entropies of guessing the correct answer for the classifiers against the entropy of the random guess attack.

This measure models the guessing attack as a random experiment with the random variable X . Depending of the displayed stimuli, X can take different values. For instance, in the PIN experiment, the set of hypotheses consists of the numbers 0 to 9 and the attack guess would then take one out of these numbers. Now, let’s assume we have no other information than the set of hypotheses. Then we would guess each answer with equal probability. This is the random attack. Let the number of possible answers (the cardinality of the set of hypotheses) be K , then the entropy of the random attack is $\log_2(K)$.

More formally, let the ranking of a classifier clf be $\mathbf{a}^{(clf)} = \{a_1^{(clf)}, \dots, a_K^{(clf)}\}$, where the first-ranked answer is $a_1^{(clf)}$, the second-ranked answer is $a_2^{(clf)}$, and so on. Let

$p(a_k^{(clf)}) := p(X = a_k^{(clf)} | \mathbf{a}^{(clf)})$ be the probability that the classifier ranks the correct answer at position $k \in K$. Please note that the $p(X = a_k^{(clf)})$ that we will use are empirical relative frequencies obtained from the experiments instead of true probability distributions. Using these probabilities, the empirical Shannon entropy is

$$H(X | \mathbf{a}^{(clf)}) = - \sum_{k=1}^K p(a_k^{(clf)}) \log_2(p(a_k^{(clf)})) \quad (5)$$

In case of the random attack, the position of the correct answer is uniformly distributed, which results in the said entropy $H(X | \mathbf{a}^{(rand)}) = \log_2(K)$. In case of attacking with a classifier, the attacker would pick a_1 , the answer ranked highest, to maximize his success. As our empirical results, depicted in Figure 9, suggest, the rankings are not fully reliable, i.e. the answer ranked highest is not always the correct answer. However, the ranking statistics provide a new non-uniform distribution over the set of possible answers. For instance, we know that for bLogReg the empirical probability that the first-ranked location is the correct one is $p(X = a_1^{(bLogReg)}) = 0.2$, the probability of the second-ranked answer to be correct is also $p(X = a_2^{(bLogReg)}) = 0.2$, and so on.

The redistributed success probabilities reduce the entropy of the guessing experiment. We take the random guess attack as the baseline and compare the entropies of all other attacks against its entropy $H(X | \mathbf{a}^{(rand)})$. We evaluate to what extent a generic classifier *clf* reduces the entropy relative to $H(X | \mathbf{a}^{(rand)})$. The relative reduction of entropy with respect to the random guess attack (in %) is then:

$$\begin{aligned} r(\text{clf}) &:= 100 \frac{H(X | \mathbf{a}^{(rand)}) - H(X | \mathbf{a}^{(clf)})}{H(X | \mathbf{a}^{(rand)})} \\ &= 100 \left(1 - \frac{H(X | \mathbf{a}^{(clf)})}{\log_2(K)} \right) \end{aligned} \quad (6)$$

A perfect classifier always has the correct answer at the first position, resulting in zero entropy and a relative reduction r of 100%. A poor classifier provides a uniform distribution of the position of the correct rank. As a consequence, its entropy would be maximal and the relative reduction r would be 0%. The entropy difference directly measures the information leaked by an attack. Thereby, comparing the classifier entropies in a relative way enables one to compare results over different experiments with different numbers of possible answers.

We report the relative reduction of entropy for each experimental setting and for each classifier in Figure 10. As one can see, the reduction approximately ranges from 15% to 40% for SWLDA and from 7% to 18% for the two bLogReg variants. Please note that the plot does not report the result of the classifier that has been trained on

the people experiment for this very experiment, as this entropy reduction merely refers to the training error of the classifier and provides no information on how well the classifier generalizes to unseen data.

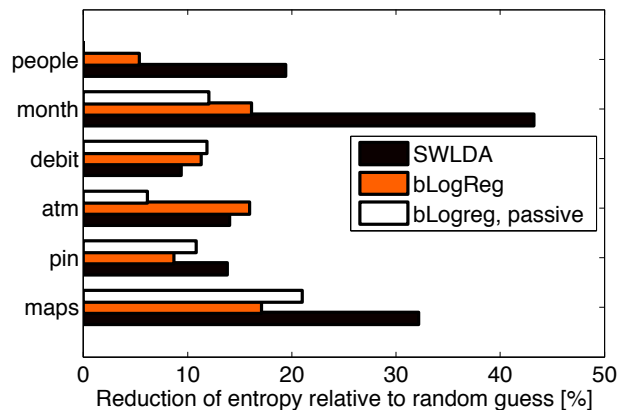


Figure 10: Relative reduction of entropy with respect to the random guess attack. The scale reaches from 0% (no advantage over random guessing) to 100% (correct answer always found). Please note that 'bLogReg, passive' has been trained on the people experiment. We do not report its score on this experiment, as it refers to the training error.

For most scenarios, the information leaked corresponds to approximately 10% to 20% for the best classifier SWLDA with peaks for *maps* (32%) and *month* (43%). The average information leak over all classifiers in the *maps* experiment stands out compared to the other result. The reason for this is that the maps experiment is a counting experiment, in which the users were asked to count the number of occurrences of the target stimulus. This experiment was included to underline the improvement in accuracy with a cooperative user.

Using Prior Knowledge to Improve Accuracy For some secrets there exist global statistics that can improve the success chances of the attack. For instance, often the distribution of customers of different banks in a population is approximately known. Also there might be prior knowledge about the area someone lives in. We did not include such prior knowledge in our experiments. However, such information could improve both the random guess as well as the classifier guesses. Prior probabilities could be included to Bayesian classifiers or could be used for heuristically post-processing classifier output.

For some experiments such as the PINs and the month of birth, the possible answers are approximately uniformly distributed, such that prior knowledge would provide no information. For other experiments prior knowledge might simply be unavailable and thus can not be used for more sophisticated models.

6 Related work

In this section, we overview related papers that use EEG signals in security-relevant applications.

EEG-based identification and authentication EEG signal has successfully been used for user identification (selecting the user identity out of a set of identities) and user authentication (verifying if a claimed user identity is true). In [30], the authors provide an overview of *cognitive biometrics*, an emerging research area that investigates how different biosignals can be used for the purpose of authentication and identification. The authors cover recent papers on biometrics based on EEG, the electrocardiogram (ECG), and the skin conductance, also called electrodermal response (EDR). An **identification** mechanism based on the alpha rhythm has been proposed in [29]. The mechanism uses convex polygon intersections to map new observations to a user identity. The authors report a high true positive rate of 95% and a true negative rate of 87% for experiments on 79 users. In method proposed in [23] uses Gaussian mixture models for user **authentication**. The authors test their method with different authentication protocols and report that with increasing temporal distance from the sign-up phase, the accuracy degrades. Using a sign-up phase over several days improves the accuracy. In [36] the authors describe *pass-thoughts*, another **authentication** mechanism that instead of typing a password requires the user to think of a password. The idea is very similar to the conventional P300-Speller scenario we mentioned in Section 2. A matrix containing characters is shown to a user and he focuses on the characters required to spell the password. This way, many shoulder-surfing attacks could be avoided. The main drawback of this authentication method (also mentioned by the authors) is a very low throughput rate of the spelling, which is ≈ 5 characters per minute for the 90% accuracy. Another problem is that the user gets no feedback until the complete passphrase is spelled, and hence the whole process must be repeated if a single character is wrongly classified.

More recently, in [15], the authors introduce a **key-generation technique** resistant against coercion attacks. The idea is to incorporate the user's emotional status through skin conductance measurements into the cryptographic key generation. This way, the generated keys contain a dynamic component that can detect whether a user is forced to grant an access to the system. Skin conductance is used as an indicator of the person's overall arousal state, i.e., the skin conductance of the victim in a stressful scenario significantly changes compared to a situation when the keys were generated.

Another highly related work to ours is described in [37]. The authors exploit an ERP called N400 to detect

if a person is actively thinking about a certain stimuli without explicitly looking at it. In contrast to the P300 which is related to attention, the N400 has been associated with semantic processing of words. For example, in an experiment where subjects are shown incongruent sentences like “*I drink coffee with milk and socks*”, the amplitude of the N400 would be maximal at the last (incorrect) word. This phenomenon is then used to detect which out of several possible objects the user is actively thinking of. While this paper is not focusing on security issues but rather on **assisting a user in efficient search**, the N400 could serve as another attack vector for similar attacks as those described in this work.

While all listed contributions support our belief that such devices may be used in everyday tasks, they follow an orthogonal approach by considering how to assist users in various tasks like, for instance, authentication. Contrary to that, our objective is to turn the table and to demonstrate that such technology might create significant threats to the security and privacy of the users.

Guilty-Knowledge Test The most closely related work on EEG signals addresses using P300 in lie detection, particularly in the so-called Guilty-Knowledge Test (GKT) [3]. The operating hypothesis of the GKT is that familiar items will evoke different responses when viewed in the context of similar unfamiliar items. It has been shown that the P300 can be used as a discriminative feature in detecting whether or not the relevant information is stored in the subject's memory. For this reason, a GKT based on the P300 has a promising use within interrogation protocols that enable detection of potential criminal details held by the suspect, although some data suggest low detection rates [13]. In contrast, recent GKT experiments based on the P300 have reported detection accuracies as high as 86% [1]. Of course, as with the polygraph-based GKT, the P300-GKT is vulnerable to specific countermeasures, but to a much lesser extent [33, 34].

Such applications in interrogation protocols have quite a number of differences from our work. For instance, we concentrate on consumer-grade devices that have considerably lower signal-to-noise ratios, therefore are more difficult to analyze. The largest difference between our approach and in the GTK is the attacker model. While the GKT-interrogator has full control over the BCI user, in that he can attach high-precision electrodes in a supportive way and force user to collaborate, our attacker must use the low-cost gaming device selected and attached by the user herself. This makes our attack considerably harder. Moreover, while the GTK victim clearly knows that she is interrogated and can prepare for that, in our case the user does not know that she is attacked. This might increase the validity of revealed information.

7 Discussion and Future Directions

In this section we discuss possible ways to defend against the investigated attacks and describe potential future directions.

Conscious Defenses Users of the BCI devices could actively try to hinder probing by, for instance, concentrating on non-target stimuli. To give a concrete example, users could count the number of occurrences of an unfamiliar face in our people experiment. The effectiveness of such defensive techniques has been tested in the context of guilty knowledge tests, however, there is no definitive conclusion on whether efforts to conceal knowledge are effective [35] or ineffective [8]. It is important to notice that, as we mentioned before, our scenario differs considerably from the GKT scenario. In our case, we assume that the EEG application has control of the user input for extended periods of time and that it conceals the attack in the normal interaction with the application. It would be difficult to imagine a realistic scenario in which a concerned user could try to conceal information from the EEG application for extended periods of normal usage.

An alternative to limiting the scope of the attacks presented in this paper is not to expose the raw data from EEG devices to third-party applications. In this model, the EEG vendor would create a restricted API that could only access certain features of the EEG signal. For example, applications could be restricted to accessing only movement related information (reflected in the spectral power). On the other hand, this poses higher performance demands on the device and limits the potential of developing third-party software.

Another possible way to deal with leaking information through the P300 signal would be adding noise to the EEG raw data before making it available to the applications that must use it. However, it would be difficult to strike a balance between the security of such an approach and the drawbacks in terms of decrease in accuracy of legitimate applications.

Future Directions The overall success of these attacks highly depends on the user's attention to the stimuli. Hence, there are still many open questions concerning the trade-off between obtrusiveness (in order to increase the user's attention during the classification task) and concealment to avoid the discovery of the attacker's true intentions. As part of our future work we intend to explore this trade-off in more detail. Specifically, by asking what is the impact of an uncooperative user who attempts to "lie" during the attack, e.g., similar to guilty-knowledge test settings? How can these attacks be made more stealthy, i.e., to what extent can they be integrated

into some benign everyday tasks, games, or videos? How effective is the social engineering approach? For example, by offering fake monetary awards or by simply confusing the user (such as asking him to verify whether his PIN is truly random and telling him to count the number of the PIN occurrences).

8 Conclusion

The broad field of possible applications and the technological progress of EEG-based BCI devices indicate that their pervasiveness in our everyday lives will increase. In this paper, we focus on the possibility of turning this technology against the privacy of its users. We believe that this is an important first step in understanding the security and privacy implications of this technology.

In this paper, we designed and carried out a number of experiments which show the feasibility of using a cheap consumer-level BCI gaming device to partially reveal private and secret information of the users. In these experiments, a user takes part in classification tasks made of different images (i.e., stimuli). By analyzing the captured EEG signal, we were able to detect which of the presented stimuli are related to the user's private or secret information, like information related to credit cards, PIN numbers, the persons known to the user, or the user's area of residence, etc. The experiments demonstrate that the information leakage from the user, measured by the information entropy is 10 %-20 % of the overall information, which can increase up to ≈ 43 %.

The simplicity of our experiments suggests the possibility of more sophisticated attacks. For example, an uninformed user could be easily engaged into "mind-games" that camouflage the interrogation of the user and make them more cooperative. Furthermore, with the ever increasing quality of devices, success rates of attacks will likely improve. Another crucial issue is that current APIs available to third-party developers offer full access to the raw EEG signal. This cannot be easily avoided, since the complex EEG signal processing is outsourced to the application. Consequently, the development of new attacks can be achieved with relative ease and is only limited by the attacker's own creativity.

Acknowledgements

This work was supported in part by the National Science Foundation under grants TRUST CCF-0424422 and grant No. 0842695, by the Intel ISTC for Secure Computing, and by the Carl-Zeiss Foundation.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

- [1] ABOOTALEBI, V., MORADI, M. H., AND KHALILZADEH, M. A. A new approach for EEG feature extraction in P300-based lie detection. *Computer Methods and Programs in Biomedicine* 94 (April 2009), 48–57.
- [2] CAMPBELL, A., CHOUDHURY, T., HU, S., LU, H., MUKERJEE, M. K., RABBI, M., AND RAIZADA, D. Neurophone: brain-mobile phone interface using a wireless EEG headset. In *Proceedings of the Second ACM SIGCOMM Workshop on Networking, Systems, and Applications on Mobile Handhelds* (2010), MobiHeld '10, pp. 3–8.
- [3] COMMITTEE TO REVIEW THE SCIENTIFIC EVIDENCE ON THE POLYGRAPH. *The Polygraph and Lie Detection*. Board on Behavioral Cognitive and Sensory Sciences, National Research Council. The National Academies Press, 2003.
- [4] DELORME, A., AND MAKEIG, S. EEGLAB: an open source toolbox for analysis of single-trial eeg dynamics including independent component analysis. *Journal of Neuroscience Methods* 134, 1 (2004), 9–21.
- [5] EMOTIV SYSTEMS. www.emotiv.com. (last accessed: Feb. 12 2012).
- [6] ENGEL, J., KUHL, D. E., PHELPS, M. E., AND CRANDALL, P. H. Comparative localization of foci in partial epilepsy by PCT and EEG. *Annals of Neurology* 12, 6 (1982), 529–537.
- [7] FARWELL, L., AND DONCHIN, E. Talking off the top of your head: toward a mental prosthesis utilizing event-related brain potentials. *Electroencephalography and Clinical Neurophysiology* 70, 6 (1988), 510 – 523.
- [8] FARWELL, L., AND SMITH, S. Using brain merger testing to detect knowledge despite efforts to conceal. *Journal of Forensic Sciences* 46, 2 (Jan 2001), 135–43.
- [9] FISCHER, R. A. The use of multiple measurements in taxonomic problems. *Annals of Human Genetics* 7, 2 (1936), 179–188.
- [10] FLOR-HENRY, P., LANG, R., KOLES, Z., AND FRENZEL, R. Quantitative EEG studies of pedophilia. *International Journal of Psychophysiology* 10, 3 (1991), 253 – 258.
- [11] FRIEDMAN, J., HASTIE, T., AND TIBSHIRANI, R. Additive logistic regression: a statistical view of boosting. *Annals of Statistics* 28 (1998), 2000.
- [12] FRIEDMAN, J. H. Greedy function approximation: A gradient boosting machine. *Annals of Statistics* 29 (2000), 1189–1232.
- [13] GAMER, M. Does the guilty actions test allow for differentiating guilty participants from informed innocents? a re-examination. *International Journal of Psychophysiology* 76 (apr 2010), 19–24.
- [14] GRUZELIER, J., EGNER, T., AND VERNON, D. Validating the efficacy of neurofeedback for optimising performance. *Event-Related Dynamics of Brain Oscillations: Progress in Brain Research* 159 (2006), 421–431.
- [15] GUPTA, P., AND GAO, D. Fighting coercion attacks in key generation using skin conductance. In *Proceedings of the 19th USENIX Conference on Security* (2010), USENIX Security'10, pp. 30–30.
- [16] HALGREN, E., MARINKOVIC, K., AND CHAUVEL, P. Generators of the late cognitive potentials in auditory and visual oddball tasks. *Electroencephalography and Clinical Neurophysiology* 106, 2 (1998), 156 – 164.
- [17] HOFFMANN, U., GARCIA, G., VESIN, J.-M., DISERENS, K., AND EBRAHIMI, T. A boosting approach to P300 detection with application to brain-computer interfaces. In *2nd International IEEE EMBS Conference on Neural Engineering* (2005), pp. 97 –100.
- [18] INZLICHT, M., MCGREGOR, I., HIRSH, J. B., AND NASH, K. Neural markers of religious conviction. *Psychological Science* 20, 3 (2009), 385–392.
- [19] J. MALMIVUO AND R. PLONSEY. Bioelectromagnetism: Principles and applications of bioelectric and biomagnetic fields. <http://www.bem.fi/book/> (last accessed: Feb. 16 2012).
- [20] KROPOTOV, J. D., GRIN-YATSENKO, V. A., PONOMAREV, V. A., CHUTKO, L. S., YAKOVENKO, E. A., AND NIKISHENA, I. S. ERPs correlates of EEG relative beta training in ADHD children. *International Journal of Psychophysiology* 55 (2004), 23–34.
- [21] KRUSIENSKI, D. J., SELLERS, E. W., CABESTAING, F., BAYOUDH, S., MCFARLAND, D. J., VAUGHAN, T. M., AND WOLPAW, J. R. A comparison of classification techniques for the P300 Speller. *Journal of Neural Engineering* 3, 4 (Dec. 2006), 299–305.

- [22] LIN, C.-T., WU, R.-C., LIANG, S.-F., CHAO, W., CHEN, Y.-J., AND JUNG, T.-P. EEG-based drowsiness estimation for safety driving using independent component analysis. *IEEE Transactions On Circuits and Systems. Part I: Regular Papers* (2005), 2726–2738.
- [23] MARCEL, S., AND MILLAN, J. Person authentication using brainwaves (EEG) and maximum a posteriori model adaptation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 29, 4 (April 2007), 743–752.
- [24] MEIJER, E., SMULDERS, F., AND WOLF, A. The contribution of mere recognition to the P300 effect in a concealed information test. *Applied Psychophysiology and Biofeedback* 34 (2009), 221–226.
- [25] NEUROSKY INC. www.neurosky.com. (last accessed: Feb. 11 2012).
- [26] NIJHOLT, A. BCI for games: A ‘state of the art’ survey. In *Proceedings of the 7th International Conference on Entertainment Computing* (2009), ICEC ’08, pp. 225–228.
- [27] POLICH, J. Updating P300: An integrative theory of P3a and P3b. *Clinical Neurophysiology* 118, 10 (2007), 2128–2148.
- [28] PORTAS, C. M., KRAKOW, K., ALLEN, P., JOSEPHS, O., ARMONY, J. L., AND FRITH, C. D. Auditory processing across the sleep-wake cycle: Simultaneous EEG and fMRI monitoring in humans. *Neuron* 28, 3 (2000), 991–999.
- [29] POULOS, M., RANGOUSI, M., CHRISSIKOPOULOS, V., AND EVANGELOU, A. Parametric person identification from the EEG using computational geometry. In *The 6th IEEE International Conference on Electronics, Circuits and Systems* (Sep 1999), vol. 2, pp. 1005–1008 vol.2.
- [30] REVETT, K., AND MAGALHES, S. T. Cognitive biometrics: Challenges for the future. In *Global Security, Safety, and Sustainability*, vol. 92. 2010, pp. 79–86.
- [31] ROS, T., MOSELEY, M. J., BLOOM, P. A., BENJAMIN, L., PARKINSON, L. A., AND GRUZELIER, J. H. Optimizing microsurgical skills with EEG neurofeedback. *BMC Neuroscience*, 1 (2009), 10–87.
- [32] ROSENFELD, J. P., BIROSCHAK, J. R., AND FUREDY, J. J. P300-based detection of concealed autobiographical versus incidentally acquired information in target and non-target paradigms. *International Journal of Psychophysiology* 60, 3 (2006), 251–259.
- [33] ROSENFELD, J. P., AND LABKOVSKY, E. New P300-based protocol to detect concealed information: Resistance to mental countermeasures against only half the irrelevant stimuli and a possible ERP indicator of countermeasures. *Psychophysiology* 47, 6 (2010), 1002–1010.
- [34] ROSENFELD, J. P., SOSKING, M., BOSH, G., AND RYAN, A. Simple, effective countermeasures to P300-based tests of detection of concealed information. *Psychophysiology* 41, 1 (2004), 205–219.
- [35] ROSENFELD, J. P., SOSKINS, M., BOSH, G., AND RYAN, A. Simple, effective countermeasures to P300-based tests of detection of concealed information. *Psychophysiology* 41 (mar 2004), 208.
- [36] THORPE, J., VAN OORSCHOT, P. C., AND SOMAYAJI, A. Pass-thoughts: authenticating with our minds. In *Proceedings of the 2005 Workshop on New Security Paradigms* (New York, NY, USA, 2005), NSPW ’05, ACM, pp. 45–56.
- [37] VAN VLIET, M., MHL, C., REUDERINK, B., AND POEL, M. Guessing what’s on your mind: Using the n400 in brain computer interfaces. vol. 6334. 2010, pp. 180–191. 10.1007/978-3-642-15314-3.17.
- [38] WAISMANN, R., FENWICK, P., WILSON, G., HEWETT, D., AND LUMSDEN, J. EEG responses to visual erotic stimuli in men with normal and paraphilic interests. *Archives of Sexual Behavior* 32 (2003), 135–144. 10.1023/A:1022448308791.
- [39] WILSON, G. F., AND RUSSELL, C. A. Operator functional state classification using multiple psychophysiological features in an air traffic control task. *The Journal of the Human Factors and Ergonomics* 45, 3 (2003), 381–389.

A Session Script

Preparation. “We will now run a series of experiments. Each one of them takes approximately 1.30 minutes. Please find a comfortable position. Please try to stay still and not move your face.” (*Participants are shown EEG feed and show the effects if the participants move their body and face*)

Training. “We will now run through a basic experiment to train our software. The system will display a random sequence of digits zero through nine. Please count the number of times [x] is shown. Please do not count the occurrences of a different number or otherwise attempt to fool the system.”

Password. “Please choose and write down a 4 digit PIN and keep it by yourself. Do not show it to me and do not use a PIN code that you normally use.”

“There are no special instructions for this experiment. However, at the end of this experiment, you will have to enter the first digit of the PIN you just chose.”

Banks ATM. “Are you a customer of any of those ten banks on the list?”

“Are you a customer with just one?”

(If yes to both) “For this experiment, instructions are displayed on-screen”

Message on screen: What is the name of your bank?

Banks Debit Cards. “For this experiment, instructions are displayed on-screen”

Message on screen: What is the name of your bank?

Geographic Location. “Do you live close to campus?”
If yes: “Instructions are displayed on-screen.”

Message on screen: Where do you live? Count the number of occurrences.

Month of Birth. “Instructions are displayed on-screen”

Message on screen: When were you born?

People “For this experiment, instructions are displayed on-screen”

Message on screen: Do you know any of these people?

Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud

Zhenyu Wu Zhang Xu Haining Wang
The College of William and Mary
{adamwu, zxu, hnw}@cs.wm.edu

Abstract

Information security and privacy in general are major concerns that impede enterprise adaptation of shared or public cloud computing. Specifically, the concern of virtual machine (VM) physical co-residency stems from the threat that hostile tenants can leverage various forms of side channels (such as cache covert channels) to exfiltrate sensitive information of victims on the same physical system. However, on virtualized x86 systems, covert channel attacks have not yet proven to be practical, and thus the threat is widely considered a “potential risk”. In this paper, we present a novel covert channel attack that is capable of high-bandwidth and reliable data transmission in the cloud. We first study the application of existing cache channel techniques in a virtualized environment, and uncover their major insufficiency and difficulties. We then overcome these obstacles by (1) re-designing a pure timing-based data transmission scheme, and (2) exploiting the memory bus as a high-bandwidth covert channel medium. We further design and implement a robust communication protocol, and demonstrate realistic covert channel attacks on various virtualized x86 systems. Our experiments show that covert channels do pose serious threats to information security in the cloud. Finally, we discuss our insights on covert channel mitigation in virtualized environments.

1 Introduction

Cloud vendors today are known to utilize virtualization heavily for consolidating workload and reducing management and operation cost. However, due to the relinquished control from data owners, data in the cloud is more susceptible to leakage by operator errors or theft attacks. Cloud vendors and users have used a number of defense mechanisms to prevent data leakage, ranging from network isolation to data encryption. Despite the efforts being paid on information safeguarding, there re-

main potential risks of data leakage, namely the covert channels in the cloud [14, 18, 22, 24, 30, 31].

Covert channels exploit imperfections in the isolation of shared resources between two unrelated entities, and enable communications between them via unintended channels, bypassing mandatory auditing and access controls placed on standard communication channels. Previous research has shown that on a non-virtualized system, covert channels can be constructed using a variety of shared media [3, 12, 16, 19, 23]. However, to date there is no known practical exploit of covert channels on virtualized x86 systems.

Exposing cloud computing to the threat of covert channel attacks, Ristenpart *et al.* [18] have implemented an L2 cache channel in Amazon EC2 [18], achieving a bandwidth of 0.2 bps (bits-per-second), far less than the one bps “acceptable” threshold suggested by the Trusted Computer System Evaluation Criteria (TCSEC, a.k.a. the “Orange Book”) [5]. A subsequent measurement study of cache covert channels [30] has achieved slightly improved speeds—a theoretical channel capacity of 1.77 bps¹. Given such low reported channel capacities from previous research, it is widely believed that covert channel attacks could only do very limited harm in the cloud environment. Coupled with the fact that the cloud vendors impose non-trivial extra service charges for providing physical isolation, one might be tempted to disregard the concerns of covert channels as only precautionary, and choose the lower cost solutions.

In this paper, we show that the threat of covert channel attacks in the cloud is real and practical. We first study existing cache covert channel techniques and their applications in a virtualized environment. We reveal that these techniques are rendered ineffective by virtualization, due to three major insufficiency and difficulties, namely, *addressing uncertainty*, *scheduling uncertainty*,

¹This value is derived from the results presented in the original paper—a bandwidth of 3.20 bps with an error rate of 9.28%, by assuming a binary symmetric channel.

and *cache physical limitations*. We tackle the addressing and scheduling uncertainty problems by designing a pure timing-based data transmission scheme with relaxed dependencies on precise cache line addressing and scheduling patterns. Then, we overcome the cache physical limitations by discovering a high-bandwidth memory bus covert channel, exploiting the atomic instructions and their induced cache–memory bus interactions on x86 platforms. Unlike cache channels, which are limited to a physical processor or a silicon package, the memory bus channel works system-wide, across physical processors, making it a very powerful channel for cross-VM covert data transmission.

We further demonstrate the real world exploitability of the memory bus covert channel by designing a robust data transmission protocol and launching realistic attacks on our testbed server as well as in the Amazon EC2 cloud. We observe that the memory bus covert channel can achieve (1) a bandwidth of over 700 bps with extremely low error rate in a laboratory setup, and (2) a real world transmission rate of over 100 bps in the Amazon EC2 cloud. Our experimental results show that, contrary to previous research and common beliefs, covert channels are able to achieve high bandwidth and reliable transmission on today’s x86 virtualization platforms.

The remainder of this paper is structured as follows. Section 2 surveys related work on covert channels. Section 3 describes our analysis of the reasons that existing cache covert channels are impractical in the cloud. Section 4 details our exploration of building high-speed, reliable covert channels in a virtualized environment. Section 5 presents our evaluation of launching covert channel attacks using realistic setups. Section 6 provides a renewed view of the threats of covert channels in the cloud, and discusses plausible mitigation avenues. Section 7 concludes this paper.

2 Related Work

Covert channel is a well known type of security attack in multi-user computer systems. Originated in 1972 by Lampson [12], the threats of covert channels are prevalently present in systems with shared resources, such as file system objects [12], virtual memory [23], network stacks and channels [3, 19, 20], processor caches [16, 24], input devices [21], etc. [5, 13].

Compared to other covert channel media, the processor cache is more attractive for exploitation, because its high operation speed could yield high channel bandwidth and the low level placement in the system hierarchy can bypass many high level isolation mechanisms. Thus, cache-based covert channels have attracted serious attention in recent studies.

Percival [16] introduced a technique to construct inter-process high bandwidth covert channels using the L1 and L2 caches, and demonstrated a cryptographic key leakage attack through the L1 cache side channel. Wang and Lee [24] deepened the study of processor cache covert channels, and pointed out that the insufficiency of software isolation in virtualization could lead to cache-based cross-VM covert channel attacks. Ristenpart *et al.* [18] further exposed cloud computing to covert channel attacks by demonstrating the feasibility of launching VM co-residency attacks, and creating an L2 cache covert channel in the Amazon EC2 cloud. Xu *et al.* [30] conducted a follow up measurement study on L2 cache covert channels in a virtualized environment. Based on their measurement results, they concluded that the harm of data exfiltration from cache covert channels is quite limited due to low achievable channel capacity.

In response to the discovery of cache covert channel attacks, a series of architectural solutions have been proposed to limit cache channels, including RCache [24], PLCache [11], and NewCache [25]. RCache and NewCache employ randomization to prevent data transmission by establishing a location-based coding scheme. PLCache, however, is based on enforcing resource isolation by cache partitioning.

One drawback of hardware-based solutions is their high adaptation cost and latency. With the goal of offering immediately deployable protection, HomeAlone [31] proposes to proactively detect the co-residence of unfriendly VMs. Leveraging the knowledge of existing cache covert channel techniques [16, 18], HomeAlone detects the presence of a malicious VM by acting like a covert channel receiver and observing cache timing anomalies caused by another receiver’s activities.

The industry has taken a more pragmatic approach to mitigating covert channel threats. The Amazon EC2 cloud provides a featured service called dedicated instances [1], which ensures VMs belonging to each tenant of this service do not share physical hardware with any other cloud tenants’ VMs. This service effectively eliminates various covert channels induced by the shared platform hardware, including cache covert channel. However, in order to enjoy this service, the cloud users have to pay a significant price premium².

Of historical interest, the study of covert channels in virtualized systems is far from a brand new research topic—legacy research that pioneered this field dates back over 30 years. During the development of the VAX security kernel, a significant amount of effort has been

²As of the time of writing (January, 2012), each dedicated instance incurs a 23.5% higher per-hour cost than regular usage. In addition, there is a \$10 fee per hour/user/region. Thus, for a user of 20 small instances, the overall cost of using dedicated instances is 6.12 times more than that of using regular instances.

Algorithm 1 Classic Cache Channel Protocol

$Cache[N]$: A shared processor cache, conceptually divided into N regions;

Each cache region can be put in one of two states, *cached* or *flushed*.

$D_{Send}[N], D_{Recv}[N]$: N bit data to transmit and receive, respectively.

Sender Operations:

(Wait for receiver to initialize the cache)

```
for  $i := 0$  to  $N - 1$  do
  if  $D_{Send}[i] = 1$  then
    {Put  $Cache[i]$  into the flushed state}
    Access memory maps to  $Cache[i]$ ;
  end if
end for
```

(Wait for receiver to read the cache)

Receiver Operations:

```
for  $i := 0$  to  $N - 1$  do
  {Put  $Cache[i]$  into the cached state}
  Access memory maps to  $Cache[i]$ ;
end for
```

(Wait for sender to prepare the cache)

```
for  $i := 0$  to  $N - 1$  do
  Timed access memory maps to  $Cache[i]$ ;
  {Detect the state of  $Cache[i]$  by latency}
  if  $AccessTime > Threshold$  then
     $D_{Recv}[i] := 1$ ; { $Cache[i]$  is flushed}
  else
     $D_{Recv}[i] := 0$ ; { $Cache[i]$  is cached}
  end if
end for
```

paid to limit covert channels within the Virtual Machine Monitor (VMM). Hu [8, 9] and Gray [6, 7] have published a series of follow up research on mitigating cache channels and bus contention channels, using timing noise injection and lattice scheduling techniques. However, this research field has lost its momentum until recently, probably due to the cancellation of the VAX security kernel project, as well as the lack of ubiquity of virtualized systems in the past.

3 Struggles of the Classic Cache Channels

Existing cache covert channels (namely, the classic cache channels) employ variants of Percival’s technique, which uses a hybrid timing and storage scheme to transmit information over a shared processor cache, as described in Algorithm 1.

The classic cache channels work very well on hyper-threaded systems, achieving transmission rates as high as hundreds of kilobytes per second [16]. However, when applied in today’s virtualized environments, the achievable rates drop drastically, to only low single-digit bits per second [18, 30]. The multiple orders of magnitude reduction in channel capacity clearly indicates that the classic cache channel techniques are no longer suitable for cross-VM data transmission. In particular, we found that on virtualized platforms, the data transmis-

sion scheme of a classic cache channel suffers three major obstacles—addressing uncertainty, scheduling uncertainty, and cache physical limitation.

3.1 Addressing Uncertainty

Classic cache channels modulate data by the states of cache regions, and hence a key factor affecting channel bandwidth is the number of regions a cache being divided. From information theory’s perspective, a specific cache region pattern is equivalent to a transmitted symbol. And the number of regions in a cache thus corresponds to the number of symbols in the alphabet set. The higher symbol count in an alphabet set, the more information can be passed per symbol.

On hyper-threaded single processor systems, for which classic cache channels are originally designed, the sender and receiver are executed on the same processor core, using the L1 cache as the transmission medium. Due to its small capacity, the L1 cache has a special property that its storage is addressed purely by virtual memory addresses, a technique called VIVT (virtually indexed, virtually tagged). With a VIVT cache, two processes can impact the same set of associative cache lines by performing memory operations with respect to the same virtual addresses in their address spaces, as illustrated in Figure 1(a). This property enables processes to precisely control the status of the cache lines, and thus

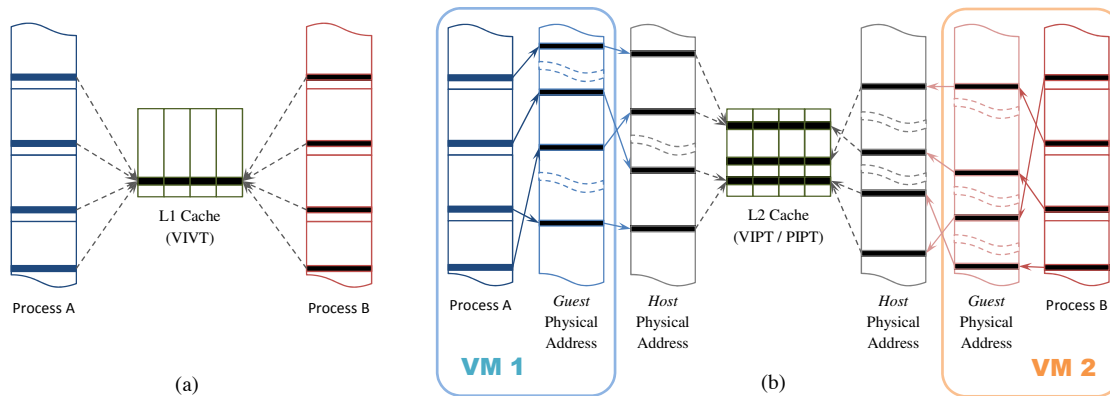


Figure 1: Memory Address to Cache Line Mappings for L1 and L2 Caches

allows for the L1 cache to be finely divided, such as 32 regions in Percival’s cache channel [16].

However, on today’s production virtualization systems, hyper-threading is commonly disabled for security reasons (i.e., eliminating hyper-threading induced covert channels). Therefore, the sender and receiver could only communicate by interleaving their executions. Since the L1 cache is completely flushed at context switches, only those higher level caches (e.g., the L2 cache) whose contents are preserved across a context switch can be leveraged for classic cache channel transmission. Unlike the L1 cache, the storage in these higher level caches is not addressed purely by virtual memory addresses, but either by physical memory addresses (PIPT, physically indexed, physically tagged), or by a mixture of virtual and physical memory addresses (VIPT, virtually indexed, physically tagged). With physical memory addresses involved in cache line addressing, given only knowledge of its virtual address space, a process cannot be completely certain of the cache line a memory access would affect due to address translation.

Server virtualization has further complicated the addressing uncertainty by adding another layer of indirection to memory addressing. As illustrated in Figure 1(b), the “physical memory” of a guest VM is still virtualized, and access to it must be further translated. As a result, it is very difficult, if not impossible, for a process in a guest VM (especially for a full virtualization VM) to discover the actual physical memory addresses of a memory region. Due to the addressing uncertainty, for classic covert channels on virtualized systems, the number of cache regions is reduced to a minimum of only two [18, 30].

3.2 Scheduling Uncertainty

Classic cache channel data transmission depends on a cache pattern “round-trip”—the receiver completely resets the cache and correctly passes it to the sender; and the sender completely prepares the cache pattern and cor-

rectly passes it back to the receiver. Therefore, to successfully transmit one cache pattern, the sender and receiver must be strictly round-robin scheduled.

However, without special scheduling arrangements (i.e., collusion) from the hypervisor, such idealistic scheduling rarely happens. On production virtualized systems, the physical processors are usually oversubscribed in order to increase utilization. In other words, each physical processing core serves more than one virtual processor from different VMs. As a result, there exist many scheduling patterns that prevent successful cache pattern “round-trip”, such as:

- * *Channel not cleared for send:* The receiver is de-scheduled before it finishes resetting the cache.
- * *Channel invalidated for send:* The receiver finishes resetting the cache, but another unrelated VM is scheduled to run immediately after.
- * *Sending incomplete:* The sender is de-scheduled before it finishes preparing the cache.
- * *Symbol destroyed:* The sender finishes preparing the cache, but another unrelated VM is scheduled to run immediately after.
- * *Receiving incomplete:* The receiver is de-scheduled before it finishes reading the cache.
- * *Channel access collision:* The sender and receiver are executed in parallel on processor cores that share the L2 cache.

Xu *et al.* [30] have clearly illustrated the problem of scheduling uncertainty in two of their measurements. First, in a laboratory setup, the error rate of their covert channel increases from near 1% to 20–30% after adding just one non-participating VM with moderate workload. Second, in the Amazon EC2 cloud, they have discovered that only 10.5% of the cache measurements at the receiver side are valid for data transmission, due to the fact that the hypervisor’s scheduling is different from the idealistic scheduling.

Algorithm 2 Timing-based Cache Channel Protocol

CLines: Several sets of associative cache lines picked by both the sender and the receiver;

These cache lines can be put in one of two states, *cached* or *flushed*.

$D_{Send}[N]$, $D_{Receive}[N]$: N bit data to transmit and receive, respectively.

Sender Operations:

```
for  $i := 0$  to  $N - 1$  do
  if  $D_{Send}[i] = 1$  then
    for an amount of time do
      {Put CLines into the flushed state}
      Access memory maps to CLines;
    end for
  else
    {Leave CLines in the cached state}
    Sleep of an amount of time;
  end if
end for
```

Receiver Operations:

```
for  $i := 0$  to  $N - 1$  do
  for an amount of time do
    Timed access memory maps to CLines;
  end for
  {Detect the state of CLines by latency}
  if  $Mean(AccessTime) > Threshold$  then
     $D_{Receive}[i] := 1$ ; {CLines is flushed}
  else
     $D_{Receive}[i] := 0$ ; {CLines is cached}
  end if
end for
```

3.3 Cache Physical Limitation

Besides the two uncertainties, classic cache channels also face an insurmountable limitation—the necessity of a *shared* and *stable* cache.

If the sender and receiver of classic cache channels are executed on processor cores that do not share any cache, obviously no communication could be established. On a multi-processor system, it is quite common to have processor cores that do not share any cache, since there is usually no shared cache between different physical processors. And sometimes even processor cores residing on the same physical processor do not share any cache, such as an Intel Core2 Quad processor, which contains two dual-core silicon packages with no shared cache in between.

Even if the sender and receiver could share a cache, external interferences can make the cache unstable. Modern multi-core processors often include a large last-level cache (LLC) shared between all processor cores. To facilitate a simpler cache coherence protocol, the LLC usually employs an inclusive principle, which requires that all data contained in the lower level caches must also exist in the LLC. In other words, when a cache line is evicted from the LLC, it must also be evicted from all the lower level caches. Thus, any non-participating processes executing on those processor cores that share the LLC with the sender and receiver can interfere with the communication by indirectly evicting the data in the cache used for the covert channel. The more cores on a processor, the higher the interference.

Overall, virtualization induced changes to cache operations and process scheduling render the data transmission scheme of classic cache channels obsolete. First, the effectiveness of data modulation is severely reduced by addressing uncertainty. Second, the critical procedures of

signal generation, delivery, and detection are frequently interrupted by less-than-ideal scheduling patterns. And finally, the fundamental requirement of stably shared cache is hard to satisfy as processors are having more cores.

4 Covert Channel in the Hyper-space

In this section, we present our techniques to tackle the existing difficulties and develop a high-bandwidth, reliable covert channel on virtualized x86 systems. At first, we describe our redesigned, pure timing-based data transmission scheme, which overcomes the negative effects of addressing and scheduling uncertainties with a simplified design. After that, we detail our findings of a powerful covert channel medium, exploiting the atomic instructions and their induced cache–memory bus interactions on x86 platforms. And finally, we specify our designs of a high error-tolerance transmission protocol for cross-VM covert channels.

4.1 A Stitch In Time

We first question the reasoning behind using cache state patterns for data modulation. Originally, Percival [16] designed this transmission scheme mainly for the use of side channel cryptographic key stealing on a hyper-threaded processor. In this specific usage context, the critical information of memory access patterns are reflected by the states of cache regions. Therefore, cache region-based data modulation is an important source of information. However, in a virtualized environment, the regions of the cache no longer carry useful information due to addressing uncertainty, making cache region-based data modulation a great source of interference.

We therefore redesign a data transmission scheme for the virtualized environment. Instead of using the cache

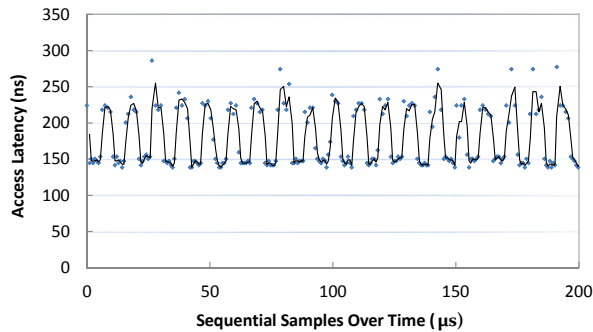


Figure 2: Timing-based Cache Channel Bandwidth Test

region-based encoding scheme, we modulate the data based on the state of cache lines over time, resulting in a pure timing-based transmission protocol, as described in Algorithm 2.

Besides removing cache region-based data modulation, the new transmission scheme also features a significant change in the scheduling requirement, i.e., signal generation and detection are performed instantaneously, instead of being interleaved. In other words, data are transmitted while the sender and receiver run in parallel. This requirement is more lenient than strict round-robin scheduling, especially with the trend of increasing number of cores on a physical processor, making two VMs more likely to run in parallel than interleaved.

We conduct a simple raw bandwidth estimation experiment to demonstrate the effectiveness of the new cache covert channel. In this experiment, interleaved bits of zeros and ones are transmitted, and the raw bandwidth of the channel can thus be estimated by manually counting the number of bits transmitted over a period of time.

We build the cache covert channel on an Intel Core2 system with two processor cores sharing a 2 MB 8-way set-associative L2 cache. Using a simple profiling test, accessing a random³ sequence of memory addresses separated by multiples of 256KB, we observe that these memory addresses can be mapped to up to 64 cache lines. Therefore, we select *CLines* as a set of 64 cache lines mapped by memory addresses following the pattern $M + X \cdot 256K$, where M is a small constant and X is a random positive integer. The sender puts these cache lines into the *flushed* state by accessing a sequence of *CLines*-mapping memory addresses. The receiver times the access latency of another sequence of *CLines*-mapping memory addresses. The length of the receiver's access sequence should be smaller than, but not too far away from the cache line set size, for example, 48.

As shown in Figure 2, the x-value of each sample point is the observed memory access latency by the receiver, and the trend line is created by plotting the mov-

³The randomness is introduced to avoid the interference of hardware prefetching.

ing average of two samples. According to the measurement results, 39 bits can be transmitted over a period of 200 micro-seconds, yielding a raw bandwidth of over 190.4 kilobits per second, about five orders of magnitude higher than the previously studied cross-VM cache covert channels.

Having resolved the negative effects of addressing and scheduling uncertainties and achieved a high raw bandwidth, our new cache covert channel, however, still performs poorly on the system with non-participating workloads. We discover that the sender and receiver have difficulty in establishing a stable communication channel. And the cause of instability is that the hypervisor frequently migrates the virtual processors across physical processor cores, which is also observed by Xu *et al.* [30]. The outgrowth of this behavior is that the sender and receiver frequently reside on processor cores that do not share any cache, making our cache channel run into the insurmountable cache physical limitation just like the classic cache channels.

4.2 Aria on the *B-String*

The prevalence of virtual processor core migration handicaps cache channels in cross-VM covert communication. In order to reliably establish covert channels across processor cores that do not share any cache, a commonly shared and exploitable resource is needed as the communication medium. And the memory bus comes into our sight as we extend our scope beyond the processor cache.

4.2.1 Background

Interconnecting the processors and the system main memory, the memory bus is responsible for delivering data between these components. Because contention on the memory bus results in a system-wide observable effect of increased memory access latency, a covert channel can be created by programmatically triggering contention on the memory bus. Such a covert channel is called a bus-contention channel.

The bus contention channels have long been studied as a potential security threat for virtual machines on the VAX VMM, on which a number of techniques have been developed [6–8] to effectively mitigate this threat. However, the x86 platforms we use today are significantly different from the VAX systems, and we suspect similar exploits can be found by probing previously unexplored techniques. Unsurprisingly, by carefully examining the memory related operations of the x86 platform, we have discovered a bus-contention exploit using atomic instructions with exotic operands.

Atomic instructions are special x86 memory manipulation instructions, designed to facilitate multi-processor

Algorithm 3 Timing-based Memory Bus Channel Protocol

M_{Exotic} : An exotic configuration of a memory region that spans two cache lines.

$D_{Send}[N], D_{Recv}[N]$: N bit data to transmit and receive, respectively.

Sender Operations:

```
for  $i := 0$  to  $N - 1$  do
  if  $D_{Send}[i] = 1$  then
    for an amount of time do
      {Put memory bus into contended state}
      Perform atomic operation with  $M_{Exotic}$ ;
    end for
  else
    {Leave memory bus in contention-free state}
    Sleep of an amount of time;
  end if
end for
```

Receiver Operations:

```
for  $i := 0$  to  $N - 1$  do
  for an amount of time do
    Timed uncached memory access;
  end for
  {Detect the state of memory bus by latency}
  if  $Mean(AccessTime) > Threshold$  then
     $D_{Recv}[i] := 1$ ; {Bus is contended}
  else
     $D_{Recv}[i] := 0$ ; {Bus is contention-free}
  end if
end for
```

synchronization, such as implementing mutexes and semaphores—the fundamental building blocks for parallel computation. Memory operations performed by atomic instructions (namely, atomic memory operations) are guaranteed to complete uninterrupted, because accesses to the affected memory regions by other processors or devices are temporarily blocked from execution.

4.2.2 Analysis

Atomic memory operations, by their design, generate system-wide observable contentions in the target memory regions they operate on. And this particular feature of atomic memory operations caught our attention. Ideally, contention generated by an atomic memory operation is well bounded, and is only evident when the affected memory region is accessed in parallel. Thus, atomic memory operations are not exploitable for cross-VM covert channels, because VMs normally do not implicitly share physical memory. However, we have found out that the hardware implementations of atomic memory operations do not match the idealistic specification, and memory contentions caused by atomic memory operations could propagate much further than expected.

Early generations (before Pentium Pro) of x86 processors implement atomic memory operations by using bus lock, a dedicated hardware signal that provides exclusive access of the memory bus to the device who asserts it. While providing a very convenient means to implement atomic memory operations, the sledgehammer-like approach of locking the memory bus results in system-wide memory contention. In addition to being exploitable for covert channels, the bus-locking implementation of atomic memory operations also causes performance and scalability problems.

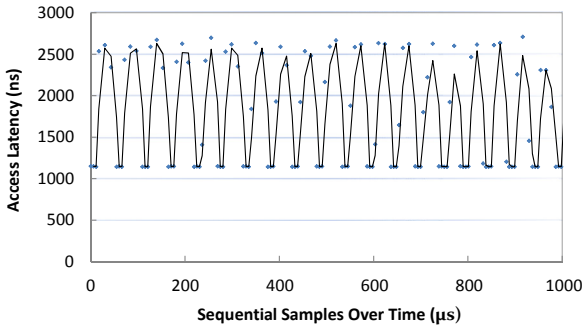
Modern generations (before Intel Nehalem and AMD K8/K10) of x86 processors improve the implementation of atomic memory operations by significantly re-

ducing the likelihood of memory bus locking. In particular, when an atomic operation is performed on a memory region that can be entirely cached by a cache line, which is a very common case, the corresponding cache line is locked, instead of asserting the memory bus lock [10]. However, on these platforms, atomic memory operations can still be exploited for covert channels, because the triggering conditions for bus-locking are not eliminated. Specifically, when atomic operations are performed on memory regions with an exotic⁴ configuration—unaligned addresses that span two cache lines, atomicity cannot be ensured by cache line locking, and bus lock signals are thus asserted.

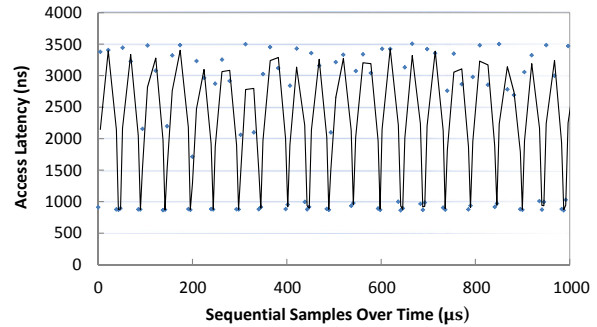
Remarkable architecture evolutions have taken place in the latest generations (Intel Nehalem and AMD K8/K10) of x86 processors, one of which is the removal of the shared memory bus. On these platforms, instead of having a unified central memory storage for the entire system, the main memory is divided into several pieces, each assigned to a processor as its local storage. While each processor has direct access to its local memory, it can also access memory assigned to other processors via a high-speed inter-processor link. This non-uniform memory access (NUMA) design eliminates the bottleneck of a single shared memory bus, and thus greatly improves processor and memory scalability. As a side effect, the removal of the shared memory bus has seemingly invalidated memory bus covert channel techniques at their foundation. Interestingly, however, the exploit of atomic memory operation continues to work on the newer platforms, and the reason for this requires a bit more in-depth explanation.

On the latest x86 platforms, normal atomic memory operations (i.e., operating on memory regions that can be

⁴The word “exotic” here only means that it is very rare to encounter such an unaligned memory access in modern programs, due to automatic data field alignments by the compilers. However, manually generating such an access pattern is very easy.



(a) Intel Core2, Hyper-V, Windows Guest VMs



(b) Intel Xeon (Nehalem), Xen, Linux Guest VMs

Figure 3: Timing-based Memory Bus Channel Bandwidth Tests

cached by a single cache line) are handled by the cache line locking mechanism similar to that of the previous generation processors. However, for exotic atomic memory operations (i.e., operating on cache-line-crossing memory regions), because there is no shared memory bus to lock, the atomicity is achieved by a set of much more complex operations: all processors must coordinate and completely flush in-flight memory transactions that are previously issued. In a sense, exotic atomic memory operations are handled on the newer platform by “emulating” the bus locking behavior of the older platforms. As a result, the effect of memory access delay is still observable, despite the absence of the shared memory bus.

4.2.3 Verification

With the memory bus exploit, we can easily build a memory bus covert channel by adapting our timing-based cache transmission scheme with minor modifications, as shown in Algorithm 3.

Compared with Algorithm 2, there are only two differences in the memory bus channel protocol. First, we substitute the set of cache lines (*CLines*) with the memory bus as the transmission medium. Similar to the cache lines, the memory bus can also be put in two states, *contended* and *contention-free*, depending on whether exotic atomic memory operations are performed. Second, instead of trying to evict contents of the selected cache lines, the sender changes the memory bus status by performing exotic atomic memory operations. And correspondingly, the receiver must make uncached memory accesses to detect contentions.

We demonstrate the effectiveness of the memory bus channel by performing bandwidth estimation experiments, similar to the one in Section 4.1, on two systems running different generations of platforms, hypervisors and guest VMs. Specifically, the first system uses an older shared memory bus platform and runs Hyper-V with Windows guest VMs, while the second system utilizes the newer platform without a shared memory bus

and runs Xen with Linux guest VMs. As Figure 3 shows, the x-value of each sample point is the observed memory access latency by the receiver, and the trend lines are created by plotting the moving average of two samples. According to the measurement results, on both systems, 39 bits can be transmitted over a period of 1 millisecond, yielding a raw bandwidth of over 38 kilobits per second. Although an order of magnitude lower in bandwidth than our cache channel, the memory bus channel enjoys its unique advantage of working across different physical processors. And notably, the same covert channel implementation works on both systems, regardless of the guest operating systems, hypervisors, and hardware platform generations.

4.3 Whispering into the Hyper-space

We have demonstrated that the memory bus channel is capable of achieving high speed data transmission on virtualized systems. However, the preliminary protocol described in Algorithm 3 is prone to errors and failures in a realistic environment, because the memory bus is a very noisy channel, especially on virtualized systems running many non-participating workloads.

Figure 4 presents a realistic memory bus channel sample, taken using a pair of physically co-resident VMs in the Amazon EC2 cloud. From this figure, we can observe that both the “contention free” and “contended” signals are subject to frequent interferences. The “contention free” signals are intermittently disrupted by workloads of other non-participating VMs, causing the memory access latency to moderately raise above the baseline. In contrast, the “contended” signals experience much heavier interferences, which originate from two sources: scheduling and non-participating workloads. The scheduling interference is responsible for the periodic drop of memory access latency. In particular, context switches temporarily de-schedule the sender process from execution, and thereby briefly relieving memory bus contention. The non-participating workloads exe-

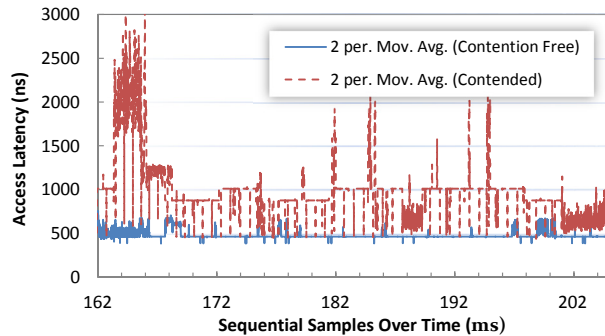


Figure 4: Memory Bus Channel Quality Sample in EC2

cuted *in parallel* with the sender process worsen memory bus contention and cause the spikes in the figure, while non-participating workloads executed *concurrently* with the sender process reduce memory bus contention, and result in the dips in the figure. All these interferences can degrade the signal quality in the channel, and make what the receiver observes different from what the sender intends to generate, which leads to *bit-flip* errors.

Besides the observable interferences shown in Figure 4, there are also unobservable interferences, i.e., the scheduling interferences to the receiver, which can cause an entirely different phenomenon. When the receiver is de-scheduled from execution, there is no observer in the channel, and thus all data being sent is lost. And to make matters worse, the receiver could not determine the amount of information being lost, because the sender may also be de-scheduled during that time. As a result, the receiver suffers from *random erasure* errors.

Therefore, three important issues need to be addressed by the communication protocol in order to ensure reliable cross-VM communication: receiving confirmation, clock synchronization, and error correction.

Receiving Confirmation: The *random erasure* errors can make the transmitted data very discontinuous, significantly reducing its usefulness. To alleviate this problem, it is very important for the sender to be aware of whether the data it sent out has been received.

We avoid using message based “send-and-ack”, a commonly employed mechanism for solving this problem, since this mechanism requires the receiver to actively send data back to the sender, reversing the roles of sending and receiving, and subjects the acknowledgment sender (i.e., the data receiver) to the same problem. Instead, we leverage the system-wide effect of memory bus contention to achieve simultaneous data transmission and receiving confirmation. Here the sender infers the presence of receiver by observing increased memory access latencies generated by the receiver.

The corresponding changes to the data transmission protocol include:

1. Instead of making uncached memory accesses, the receiver performs exotic atomic memory operations, just like the sender transmitting a one bit.
2. Instead of sleeping when transmitting a zero bit, the sender performs uncached memory accesses. In addition, the sender always times its memory accesses.
3. While the receiver is in execution, the sender should always observe high memory access latencies; otherwise, the sender can assume the data has been partially lost, and retry at a later time.

Clock Synchronization: Since the sender and receiver belong to two independent VMs, scheduling differences between them tend to make the data transmission and detection procedures de-synchronized, which can cause a significant problem to pure timing-based data modulation. We overcome clock de-synchronization by using self-clocking coding—a commonly used technique in telecommunications. Here we choose to transmit data bits using differential Manchester encoding, a standard network coding scheme [28].

Error Correction: Even with self-clocking coding, *bit-flip* errors are expected to be common. Similar to resolving the receiving confirmation problem, we again avoid using acknowledgment-based mechanisms. Assuming only a one-way communication channel, we resolve the error correction problems by applying forward error correction (FEC) to the original data, before applying self-clocking coding. More specifically, we use the Reed-Solomon coding [17], a widely applied block FEC code with strong multi-bit error correction performance.

In addition, we strengthen the communication protocol’s resilience to clock drifting and scheduling interruption by employing data framing. We break the data into segments of fixed-length bits, and frame each segment with a start-and-stop pattern. The benefits of data framing are twofold. First, when the sender detects transmission interruption, instead of retransmitting the whole piece of data, only the affected data frame is retried. Second, some data will inevitably be lost during transmission. With data framing, the receiver can easily localize the erasure errors and handle them well through the Reed-Solomon coding.

The finalized protocol with all the improvements in place is presented in Algorithm 4.

5 Evaluation

We evaluate the exploitability of memory bus covert channels by implementing the reliable Cross-VM communication protocol, and demonstrate covert channel attacks on our in-house testbed server, as well as on the Amazon EC2 cloud.

Algorithm 4 Reliable Timing-based Memory Bus Channel Protocol

$M_{ExoticS}, M_{ExoticR}$: Exotic memory regions for the sender and the receiver, respectively.

D_{Send}, D_{Recv} : Data to transmit and receive, respectively.

Sender Prepares D_{Send} by:

{ $DM_{Send}[]$: Segmented encoded data to send}

$RS_{Send} := \text{ReedSolomon}_{Encode}(D_{Send});$

$FD_{Send}[] := \text{Break } RS_{Send} \text{ into segments};$

$DM_{Send}[] := \text{DiffManchester}_{Encode}(FD_{Send}[]);$

Sending Encoded Data in a Frame:

{ $Data$: A segment of encoded data to send}

{ $FrmHead, FrmFoot$: Unique bit patterns signifying start and end of frame, respectively}

$Result := \text{SendBits}(FrmHead);$

if $Result$ is **not** *Aborted* **then**

$Result := \text{SendBits}(Data);$

if $Result$ is **not** *Aborted* **then**

 {Ignore error in sending footer}

$\text{SendBits}(FrmFoot);$

return *Succeed*;

end if

end if

return *Retry*;

Sending a Block of Bits:

{ $Block$: A block of bits to send}

{ $Base_1, Base_0$: Mean contention-free access time for sending bit 1 and 0, respectively}

for each Bit in $Block$ **do**

if $Bit = 1$ **then**

for an amount of time **do**

 Timed atomic operation with $M_{ExoticS}$;

end for

$Latency := \text{Mean}(\text{AccessTime}) - Base_1;$

else

for an amount of time **do**

 Timed uncached memory access;

end for

$Latency := \text{Mean}(\text{AccessTime}) - Base_0;$

end if

if $Latency < \text{Threshold}$ **then**

 {Receiver not running, abort}

return *Aborted*;

end if

end for

return *Succeed*;

Receiver Recovers D_{Recv} by:

{ $DM_{Recv}[]$: Segmented encoded data received}

$FD_{Recv}[] := \text{DiffManchester}_{Decode}(DM_{Recv}[]);$

$RS_{Recv} := \text{Concatenate } FD_{Recv}[];$

$D_{Recv} := \text{ReedSolomon}_{Decode}(RS_{Recv});$

Receiving Encoded Data in a Frame:

{ $Data$: A segment of encoded data to receive}

Wait for frame header;

$Result := \text{RecvBits}(Data);$

if $Result$ is *Aborted* **then**

return *Retry*;

end if

$Result := \text{Match frame footer};$

if $Result$ is **not** *Matched* **then**

 {Clock synchronization error, discard $Data$ }

return *Erased*;

else

return *Succeed*;

end if

Receiving a Block of Bits:

{ $Block$: a block of bits to receive}

for each Bit in $Block$ **do**

for an amount of time **do**

 Timed atomic operation with $M_{ExoticR}$;

end for

 {Detect the state of memory by latency}

if $\text{Mean}(\text{AccessTime}) > \text{Threshold}$ **then**

$Bit := 1$; {Bus is *contended*}

else

$Bit := 0$; {Bus is *contention-free*}

end if

 {Detect sender de-schedule}

if too many consecutive 0 or 1 bits **then**

 {Sender not running}

 Sleep for some time;

 {Sleep makes sender abort, then we abort}

return *Aborted*;

end if

end for

return *Succeed*;

5.1 In-house Experiments

We launch covert channel attacks on our virtualization server equipped with the latest generation x86 platform (i.e., with no shared memory bus). The experimental setup is simple and realistic. We create two Linux VMs, namely VM-1 and VM-2, each with a single virtual processor and 512 MB of memory. The covert channel sender runs as an unprivileged user program on VM-1, while the covert channel receiver runs on VM-2, also as an unprivileged user program.

We first conduct a quick profiling to determine the optimal data frame size and error correction strength. And we find out that a data frame size of 32 bits (including an 8 bit preamble), and a ratio of 4 parity symbols (bytes) per 4 data bytes works well. Effectively, each data frame transmits 8 bits of preamble, 12 bits of data, and 12 bits of parity, yielding an efficiency of 37.5%. In order to minimize the impact of burst errors, such as multiple frame losses, we group 48 data and parity bytes, and randomly distribute them across 16 data frames using a linear congruential generator (LCG).

We then assess the capacity (i.e., bandwidth and error rate) of the covert channel by performing a series of data transmissions using these parameters. For each transmission, a one kilobyte data block is sent from the sender to the receiver. With 50 repeated transmissions, we observe a stable transmission rate of 746.8 ± 10.1 bps. Data errors are observed, but at a very low rate of 0.09%.

5.2 Amazon EC2 Experiments

We prepare the Amazon EC2 experiments by spawning physically co-hosted Linux VMs. Thanks to the operational experiences presented in [18, 30], using only two accounts, we successfully uncover two pairs of physically co-hosted VMs (micro instances) in four groups of 40 VMs (i.e. each group consists of 20 VMs spawned by each account). Information disclosed in `/proc/cpuinfo` shows that these servers use the shared-memory-bus platform, one generation older than our testbed server used in the previous experiment.

Similar to our in-house experiments, we first conduct a quick profiling to determine the optimal data frame size and error correction strength. Compared to our in-house system profiles, memory bus channels on Amazon EC2 VMs have a higher tendency of clock desynchronization. We compensate for this deficiency by reducing the data frame size to 24 bits. The error correction strength of 4 parity symbols per 4 data bytes still works well. And the overall transmission efficiency thus becomes 33.3%.

We again perform a series of data transmissions and measure the bandwidth and error rates. Our initial results

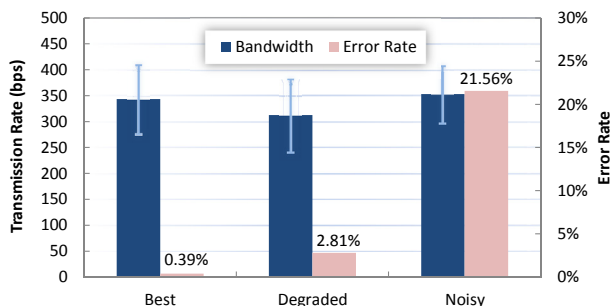


Figure 5: Memory Bus Channel Capacities in EC2

are astonishingly good. A transmission rate of 343.5 ± 66.1 bps is achieved, with error rate of 0.39%. However, as we continue to repeat the measurements, we observe an interesting phenomenon. As illustrated in Figure 5, three distinct channel performances are observed through our experiment. The best performance is achieved during the initial 12–15 transmissions. After that, for the next 5–8 transmissions, the performance degrades. The bandwidth slightly reduces, and the error rate slightly increases. Finally, for the rest of the transmissions, the performance becomes very bad. While the bandwidth is still comparable to that of the best performance, the error rate becomes unacceptably high.

By repeating this experiment, we uncover that the three-staged behavior can be repeatedly observed after leaving both VMs idle for a long period of time (e.g., one hour). Therefore, we believe that the cause of this behavior can be explained by scheduler preemption [29] as discussed in [30]. During the initial transmissions, the virtual processors of VMs at both the sender and receiver sides have high scheduling priorities, and thus they are very likely to be executed in parallel, resulting in a very high channel performance. Then, the sender VM’s virtual processor consumes all its scheduling credits and is throttled back by the Xen scheduler, causing the channel performance to degrade. Soon after that, the receiver VM’s virtual processor also uses up its scheduling credits. Since both the sender and receiver are throttled back, their communication is heavily interrupted. This “offensive” scheduling pattern subjects the communication channel to heavy random erasures beyond the correction capability of the FEC mechanism.

Fortunately, our communication protocol is designed to handle very unreliable channels. We adapt to the scheduler preemption by tuning two parameters to be more “defensive”. First, we increase the ratio of parity bits to 4 parity symbols per 2 data bytes. Although it reduces transmission efficiency by 11.1%, the error correction capability of our FEC is increased by 33.3%. Second, we reduce the transmission symbol rate by about 20%. By lengthening the duration of the receiving confir-

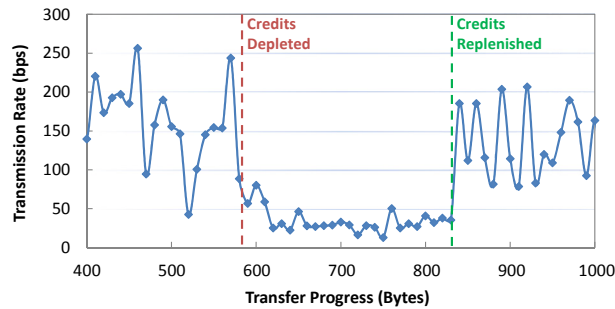


Figure 6: Reliable Transmission with Adaptive Rates

mation, we effectively increase the probability of discovering scheduling interruptions. After the parameter adjustment, we can achieve a transmission rate of 107.9 ± 39.9 bps, with an error rate of 0.75%, even under scheduler preemption.

Figure 6 depicts the adjusted communication protocol in action. During the first period of preemption-free scheduling, the transmission rate can be as high as 250 bps. However, when preemption starts, the sender responds to frequent transmission failures with increased retries, allowing the receiver continue to receive and decode data without uncorrectable error. And correspondingly, the transmission rate drops to below 50 bps. Finally, when the harsh scheduling condition is alleviated, the transmission rate is automatically restored. The capability of adaptively adjusting transmission rates to channel conditions, evidences the versatility of our reliable communication protocol.

6 Discussion

In this section, we first reassess the threat of covert channel attacks based on our experimental results. Then, we discuss possible means to mitigate the covert channel attacks in virtualized environments.

6.1 Damage Assessment

We extrapolate the threat of the memory bus covert channel from four different aspects—attack scenario, achievable bandwidth, mitigation difficulties, and cross-platform applicability.

6.1.1 Attack Scenario

Covert channel attacks are distinct from a seemingly similar threat, side channel attacks. Side channels extrapolate information by observing an unknowing sender, while covert channels transfer data between two collaborating parities. As a result, a successful covert channel attack requires an “insider” to function as a data source.

However, this additional requirement does not significantly reduce the usefulness of covert channels in data theft attacks.

Data theft attacks are normally launched in two steps, *infiltration* and *exfiltration*. In the infiltration step, attackers leverage multiple attack vectors, such as buffer overflow [4], VM image pollution [2, 26], and various social engineering techniques [15, 27], to place “insiders” in the victim and gain partial control over it. And then, in the exfiltration step, the “insiders” try to traffic sensitive information from the victim back to the attackers. Because the “insiders” usually would only have very limited control of the victim, their behaviors are subjected to strict security surveillance, e.g., firewall, network intrusion detection, traffic logging, etc. Therefore, covert channels become ideal choices for secret data transmissions under such circumstances.

6.1.2 Achievable Bandwidth

Due to their very low channel capacities [18, 30], previous studies conclude that covert channels can only cause very limited harms in a virtualized environment. However, the experimental results of our covert channel lead us to a different conclusion that covert channels indeed pose realistic and serious threats to information security in the cloud.

With over 100 bits-per-second high speed and reliable transmission, covert channel attacks can be applied to a wide range of mass-data theft attacks. For example, a hundred byte credit card data entry can be silently stolen in less than 30 seconds; and a thousand byte private key file can be secretly transmitted under 3 minutes. Working continuously, over 1 MB of data, equivalent to tens of thousands of credit card entries or hundreds of private key files, can be trafficked every 24 hours.

6.1.3 Mitigation Difficulties

In addition to high channel capacity, the memory bus covert channel has two other intriguing properties which make it difficult to be detected or prevented:

- *Stealthiness*: Because processor cache is not used as channel medium, the memory bus covert channel incurs negligible impact on cache performance, making it totally transparent to cache based covert channel detection, such as HomeAlone [31].
- *“Future proof”*: Our in-house experiment shows that even on a platform that is one generation ahead of Amazon EC2’s systems, the memory bus covert channel continues to perform very well.

6.1.4 Cross-platform Applicability

Due to hardware availability, we have only evaluated memory bus covert channels on the Intel x86 platforms. On one hand, we make an intuitive inference that similar covert channels can also be established on the AMD x86 platforms, since they share compatible specifications on atomic instructions with the Intel x86 platforms. On the other hand, the atomic instruction exploits may not be applicable on platforms that use alternative semantics to guarantee operation atomicity. For example, MIPS and several other platforms use the load-linked/store-conditional paradigm, which does not result in high memory bus contention as atomic instructions do.

6.2 Mitigation Techniques

The realistic threat of covert channel attacks calls for effective and practical countermeasures. We discuss several plausible mitigation approaches from three different perspectives—tenants, cloud providers, and device manufactures.

6.2.1 Tenant Mitigation

Mitigating covert channels on the tenant side has the advantages of trust and deployment flexibility. With the implementation of mitigation techniques inside a tenant owned VMs, the tenant has the confidence of covert channel security, regardless whether the cloud provider addresses this issue.

However, due to the lack of lower level (hypervisor and/or hardware) support, the available options are very limited, and the best choice is performance anomaly detection. Although not affecting the cache performances, memory bus covert channels do cause memory performance degradation. Therefore, an approach similar to that of HomeAlone [31] could be taken. In particular, the defender continuously monitors memory access latencies, and asserts alarms if significant anomalies are detected. However, since memory accesses incur much higher cost and non-determinism than cache probing, this approach may suffer from high performance overhead and high false positive rate.

6.2.2 Cloud Provider Mitigation

Compared to their tenants, cloud providers are much more resourceful. They control not only the hypervisor and hardware platform on a single system, but also the entire network and systems in a data center. As a result, cloud providers can tackle covert channels through either preventative or detective countermeasures.

The preventative approaches, e.g., the dedicated instances service provided by the Amazon EC2 cloud [1],

thwart covert channel attacks by eliminating the exploiting factors of covert channels. As the significant extra service charge of the dedicated instance service reduces its attractiveness, the “no-sharing” guarantee may be too strong for covert channel mitigation. We envision a low cost alternative solution that allows tenants to share system resources in a controlled and deterministic manner. For example, the cloud provider may define a policy that each server might be shared by up to two tenants, and each tenant could only have a predetermined neighbor. Although this solution does not eliminate covert channels, it makes attacking arbitrary tenants in the cloud very difficult.

In addition to preventative countermeasures, cloud providers can easily take the detective approach by implementing low overhead detection mechanisms, because of their convenient access to the hypervisor and platform hardware. For both cache and memory bus covert channels, being able to generate observable performance anomalies is the key to their success in data transmission. However, modern processors have provided a comprehensive set of mechanisms to monitor and discover performance anomalies with very low overhead. Instead of actively probing cache or accessing memory, cloud providers can leverage the hypervisor to infer the presence of covert channels, by keeping track of the increment rates of the cache miss counters or memory bus lock counters [10]. Moreover, when suspicious activities are detected, cloud providers can gracefully resolve the potential threat by migrating suspicious VMs onto physically isolated servers. Without penalizing either the suspect or the potential victims, the negative effects of false positives are minimized.

6.2.3 Device Manufacture Mitigation

The defense approaches of both tenant and cloud providers are only secondary in comparison to mitigation by the device manufactures, because the root causes of the covert channels are imperfect isolation of the hardware resources.

The countermeasures at the device manufacture side are mainly preventative, and they come in various forms of resource isolation improvements. For example, instead of handling exotic atomic memory operations in hardware and causing system-wide performance degradation, the processor may be redesigned to trap these rare situations for the operating systems or hypervisors to handle, without disrupting the entire system. A more general solution is to tag all resource requests from guest VMs, enabling the hardware to differentiate requests by their owner VMs, and thereby limiting the scope of any performance impact. While incurring high cost in hardware upgrades, the countermeasures at the device manufacture

side are transparent to cloud providers and tenants, and can potentially yield the lowest performance penalty and overall cost compared to other mitigation approaches.

7 Conclusion and Future Work

Covert channel attacks in the cloud have been proposed and studied. However, the threats of covert channels tend to be down-played or disregarded, due to the low achievable channel capacities reported by previous research. In this paper, we presented a novel construction of high-bandwidth and reliable cross-VM covert channels on the virtualized x86 platform.

With a study on existing cache channel techniques, we uncovered their application insufficiency and limitations in a virtualized environment. We then addressed these obstacles by designing a pure timing-based data transmission scheme, and discovering the bus locking mechanism as a powerful covert channel medium. Leveraging the memory bus covert channel, we further designed a robust data transmission protocol. To demonstrate the real-world exploitability of our proposed covert channels, we launched attacks on our testbed system and in the Amazon EC2 cloud. Our experimental results show that, contrary to previous research and common beliefs, covert channel attacks in a virtualized environment can achieve high bandwidth and reliable transmission. Therefore, covert channels pose formidable threats to information security in the cloud, and they must be carefully analyzed and mitigated.

For the future work, we plan to explore various mitigation techniques we have proposed. Especially, we view the countermeasures at the cloud provider side a highly promising field of research. Not only do cloud providers have control of rich resources, they also have strong incentive to invest in covert channel mitigation, because ensuring covert channel security gives them a clear edge over their competitors.

References

- [1] Amazon Web Services. Amazon EC2 dedicated instances. <http://aws.amazon.com/dedicated-instances/>.
- [2] S. Bugiel, S. Nürnberger, T. Pöppelmann, A.-R. Sadeghi, and T. Schneider. AmazonIA: when elasticity snaps back. In *Proceedings of the 18th ACM conference on Computer and communications security (CCS'11)*, pages 389–400, 2011.
- [3] S. Cabuk, C. E. Brodley, and C. Shields. IP covert timing channels: design and detection. In *Proceedings of the 11th ACM conference on Computer and communications security (CCS'04)*, pages 178–187, 2004.
- [4] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th conference on USENIX Security Symposium*, pages 63–78, 1998.
- [5] Department of Defense. TCSEC: Trusted computer system evaluation criteria. Technical Report 5200.28-STD, U.S. Department of Defense, 1985.
- [6] J. W. Gray III. On introducing noise into the bus-contention channel. In *Proceedings of the 1993 IEEE Symposium on Security and Privacy (S&P'93)*, pages 90–98, 1993.
- [7] J. W. Gray III. Countermeasures and tradeoffs for a class of covert timing channels. Technical report, Hong Kong University of Science and Technology, 1994.
- [8] W. Hu. Reducing timing charmers with fuzzy time. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy (S&P'91)*, pages 8–20, 1991.
- [9] W. Hu. Lattice scheduling and covert channels. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'92)*, pages 52–61, 1992.
- [10] Intel. The Intel 64 and IA-32 architectures software developer's manual. <http://www.intel.com/products/processor/manuals/>.
- [11] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *Proceedings of the IEEE 15th International Symposium on High Performance Computer Architecture (HPCA'09)*, pages 393–404, 2009.
- [12] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16:613–615, 1973.
- [13] F. G. G. Meade. A guide to understanding covert channel analysis of trusted systems. Manual NCSC-TG-030, U.S. National Computer Security Center, 1993.
- [14] D. G. Murray, S. H. and M. A. Fetterman. Satori: Enlightened page sharing. In *Proceedings of the USENIX Annual Technical Conference (ATC'09)*, pages 1–14, 2009.

- [15] G. L. Orgill, G. W. Romney, M. G. Bailey, and P. M. Orgill. The urgency for effective user privacy-education to counter social engineering attacks on secure computer systems. In *Proceedings of the 5th conference on Information technology education (CITC5'04)*, pages 177–181, 2004.
- [16] C. Percival. Cache missing for fun and profit. In *Proceedings of the BSDCan 2005*, 2005.
- [17] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.
- [18] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS'09)*, pages 199–212, 2009.
- [19] C. H. Rowland. Covert channels in the TCP/IP protocol suite. *First Monday*, 2, 1997.
- [20] G. Shah and M. Blaze. Covert channels through external interference. In *Proceedings of the 3rd USENIX conference on Offensive technologies (WOOT'09)*, pages 1–7, 2009.
- [21] G. Shah, A. Molina, and M. Blaze. Keyboards and covert channels. In *Proceedings of the 15th conference on USENIX Security Symposium*, pages 59–75, 2006.
- [22] K. Suzaki, K. Iijima, T. Yagi, and C. Artho. Software side channel attack on memory deduplication. page Poster, 2011.
- [23] T. V. Vleck. Timing channels. Poster session, IEEE TCSP conference, 1990.
- [24] Z. Wang and R. B. Lee. Covert and side channels due to processor architecture. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 473–482, 2006.
- [25] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture (MICRO'41)*, pages 83–93, 2008.
- [26] J. Wei, X. Zhang, G. Ammons, V. Bala, and P. Ning. Managing security of virtual machine images in a cloud environment. In *Proceedings of the 2009 ACM workshop on Cloud computing security (CCSW'09)*, pages 91–96, 2009.
- [27] I. S. Winkler and B. Dealy. Information security technology?...don't rely on it: a case study in social engineering. In *Proceedings of the 5th conference on USENIX UNIX Security Symposium*, pages 1–5, 1995.
- [28] J. Winkler and J. Munn. Standards and architecture for token-ring local area networks. In *Proceedings of 1986 ACM Fall joint computer conference (ACM'86)*, pages 479–488, 1986.
- [29] XenSource. Xen credit scheduler. <http://wiki.xensource.com/xenwiki/CreditScheduler>.
- [30] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop (CCSW'11)*, pages 29–40, 2011.
- [31] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (S&P'11)*, pages 313–328, 2011.

Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services

Nuno Santos, Rodrigo Rodrigues[†], Krishna P. Gummadi, Stefan Saroiu[‡]
MPI-SWS, [†]CITI/Universidade Nova de Lisboa, [‡]Microsoft Research

Abstract

Accidental or intentional mismanagement of cloud software by administrators poses a serious threat to the integrity and confidentiality of customer data hosted by cloud services. Trusted computing provides an important foundation for designing cloud services that are more resilient to these threats. However, current trusted computing technology is ill-suited to the cloud as it exposes too many internal details of the cloud infrastructure, hinders fault tolerance and load-balancing flexibility, and performs poorly. We present Excalibur, a system that addresses these limitations by enabling the design of trusted cloud services. Excalibur provides a new trusted computing abstraction, called *policy-sealed data*, that lets data be *sealed* (i.e., encrypted to a customer-defined policy) and then *unsealed* (i.e., decrypted) only by nodes whose configurations match the policy. To provide this abstraction, Excalibur uses attribute-based encryption, which reduces the overhead of key management and improves the performance of the distributed protocols employed. To demonstrate that Excalibur is practical, we incorporated it in the Eucalyptus open-source cloud platform. Policy-sealed data can provide greater confidence to Eucalyptus customers that their data is not being mismanaged.

1 Introduction

Managing cloud computing services is complex and error-prone. Cloud providers therefore delegate this task to skilled cloud administrators who manage the cloud infrastructure software. However, it is difficult to assure that their actions are error-free. In particular, an accidental or, in some cases, intentional action from a cloud administrator could leak, corrupt, or lose customer data. The threat of potential violations to the integrity and confidentiality of customer data is often cited as a key barrier to the adoption of cloud services [2, 15]. Furthermore, publicized incidents involving the loss of confidentiality or integrity of customer data [1, 4, 7, 23, 25] and the growing amount of security-sensitive data outsourced to the cloud [3, 6] only heightens these concerns.

Recently, several proposals [22, 39, 45, 53] have advocated leveraging trusted computing technology to make cloud services more resilient to integrity and confidentiality concerns. This technology relies on a secure coprocessor – typically a Trusted Platform Module (TPM) chip [17] – deployed on every node in the cloud. Each

TPM chip would store a strong identity (unique key) and a fingerprint (hash) of the software stack that booted on the cloud node. TPMs could then restrict the upload of customer data to cloud nodes whose identities or fingerprints are considered *trusted*. This capability offers a building block in the design of trusted cloud services by securing data confidentiality and integrity against insiders, or confining the data location to a desired geographical or jurisdictional boundary.

Despite their benefits, current trusted computing abstractions are ill-suited to the requirements of cloud services for three main reasons. First, TPM abstractions were designed to protect data and secrets on a standalone machine; they are thus cumbersome to use in a multi-node datacenter environment where data migrates across multiple nodes with potentially different configurations. Second, TPM abstractions over-expose the cloud infrastructure by revealing the identity and software fingerprint of individual cloud nodes; external agents could use this information to exploit vulnerabilities in the cloud infrastructure or gain business advantage [40]. Third, the current implementation of TPM abstractions is inefficient and can introduce scalability bottlenecks to cloud services.

This paper presents Excalibur, a system that provides cloud service designers with new trusted computing abstractions that overcome these barriers. These abstractions provide another critical building block for constructing services that offer better guarantees regarding data integrity, confidentiality, or location. Excalibur's design includes two main innovations crucial to overcoming the concerns posed by using TPMs in the cloud.

First, Excalibur provides a new trusted computing abstraction, called *policy-sealed data*, that allows customer data to be encrypted according to a customer-chosen policy and guarantees that only the cloud nodes whose configuration satisfies that policy can decrypt and retrieve the data. We devised this abstraction to address the first two limitations of current TPM abstractions; the abstraction permits multiple nodes with or without identical configurations to flexibly access data as long as they satisfy the customer policies. Moreover, since it allows policies to be specified using human-readable attributes, policy-sealed data hides the low-level identities and software fingerprints of nodes.

Second, Excalibur implements the policy-sealed data abstraction in a way that overcomes the inefficiency hur-

dles of current TPMs and scales to the demand of cloud services. To do this, we designed a centralized *monitor* that checks the integrity of cloud nodes and acts as a single point-of-contact for customers to bootstrap trust in the cloud infrastructure. To prevent the potential scalability challenges associated with a centralized monitor, we designed a set of distributed protocols to efficiently implement the new abstractions. Our protocols use the Ciphertext Policy Attribute-Based Encryption (CPABE) encryption scheme [11], which drastically reduces the communication needs between the monitor and production nodes by requiring each node contact the monitor only once during a boot cycle, a relatively infrequent operation. We validated the correctness of Excalibur's cryptographic protocols using a protocol verifier [12].

To demonstrate the practicality of Excalibur, we built a proof-of-concept compute service akin to EC2. Based on the Eucalyptus open source cloud management platform [36], our service leveraged Excalibur to give users better guarantees regarding the type of hypervisor or the location where their VM instances run. Our experience shows that Excalibur's primitive is simple and versatile: our changes required minimal modifications to the Eucalyptus codebase.

Our evaluation suggests that Excalibur scales well. Due to CPABE, the monitor's load scales independent of the workload. In addition, according to our simulations, one server acting as a monitor was sufficient to manage a large cluster; for example, a server took ~ 15 seconds to check the node configurations of a cluster with 10K nodes that all rebooted simultaneously. Finally, offering trusted computing guarantees to the EC2-like service added modest overhead during VM management operations only.

2 Trusted Computing Concepts

The success of a cloud provider hinges on its customers being willing to entrust the provider with their data [2, 15]. A key factor in strengthening customers' trust is providing strong assurances about the integrity of the cloud infrastructure. TPMs can play a fundamental role in providing these assurances.

The integrity of the cloud infrastructure depends on the security of its hardware and software components. For hardware security, cloud providers already rely on surveillance devices and physical access control that severely restrict physical access to cloud nodes, even by cloud provider staff [19]. In certain cases, by deploying cloud nodes in sealed containers, they ensure that physical access is fully disallowed [19]. For software security, providers could take advantage of techniques that reduce the size of the TCB [53], narrow the management interfaces [34], and verify the TCB code [24]. These techniques help designers build secure software

platforms (e.g., secure hypervisors) to host customers' data and computations.

However, current cloud architectures provide scant assurances that the data that customers ship to the cloud is being handled by integrity-protected nodes running secure software platforms. Insecure software platforms (e.g., ones that have been tampered with or that run unpatched software versions) put at risk cloud service integrity and thus customer data. Trusted computing technology addresses this problem by providing customers with integrity guarantees of the cloud nodes themselves.

Trusted computing technology provides the hardware support needed to bootstrap trust in a computer [38]. To do so, it offers system designers four main abstractions. First, *strong identities* let the computer be uniquely identified without having to trust the OS or the software running on the computer. Second, *trusted boot* produces a unique fingerprint of the software platform running on the computer; the fingerprint consists of hashes of software platform components (e.g., BIOS, firmware controlling the computer's devices, bootloader, OS) computed at boot time. Third, this fingerprint can be securely reported to a remote party using a *remote attestation* protocol; this protocol lets the remote party authenticate both the computer and the software platform so it can assess whether the computer is trustworthy, e.g., if it is a trusted platform that is designed to protect the confidentiality and integrity of data [20, 32]. Fourth, *sealed storage* allows the system to protect persistent secrets (e.g., encryption keys) from an attacker with the ability to reboot the machine and install a malicious OS that can inspect the disk; the secrets are encrypted so that they can be decrypted only by the same computer running the trusted software platform specified upon encryption.

An important instance of trusted computing hardware is the Trusted Platform Module (TPM) [17], a secure co-processor widely deployed on desktops, laptops and increasingly on servers. To offer a strong identity, the TPM uses an Attestation Identity Key (AIK). To track the hash values that constitute a fingerprint, the TPM uses special registers called Platform Configuration Registers (PCRs). Whenever a reboot occurs, the PCRs are reset and updated with new hash values. To perform remote attestation, the TPM can issue a *quote*, which includes the PCR values signed by the TPM with an AIK. For sealed storage, the TPM offers two primitives, called *seal* and *unseal*, to encrypt and decrypt secrets, respectively. Seal encrypts the input data and binds it to the current set of PCR values. Unseal validates the identity and fingerprint of the software platform before decrypting sealed data.

3 Threat Model

Our premise is that the attacker seeks to compromise customer data by extracting it from integrity-protected cloud nodes. An attack is successful if either the data is accessible on a machine running an insecure software platform or is moved outside the provider's premises.

The attacker is assumed to be an agent with privileged access to the cloud nodes' management interface. Such an agent is typically a cloud provider's employee who manages cloud software and behaves inappropriately due either to negligence (e.g., misconfiguring the nodes where a computation should run) or to malice (e.g., desire to steal customer data). The management interface is accessible only from a *remote site*. Therefore, we assume the attacker cannot launch physical attacks. In fact, software and hardware management roles are usually differentiated and assigned to different teams.

The management interface grants the attacker privileges to the software platform running on the node (e.g., access to the root account) and to a dedicated hardware component for power cycling the nodes. These privileges empower him to access customer data on the nodes: he can reboot any node, access its local disk after rebooting, install arbitrary software on the node, and eavesdrop the network. However, whenever cloud nodes boot a secure software platform whose TCB we assume to be correct, the attacker can no longer exploit vulnerabilities through the software platform's interface.

Multiple trusted parties perform all other management tasks in the cloud provider's infrastructure. These tasks include, e.g., procuring and deploying the hardware, securing the premises, developing the software platforms, managing the provider's private keys, endorsing whether a software platform is secure, certifying the software and hardware, etc. Trusted parties can be employees of the cloud provider or external trusted organizations. Due to the nature of their roles, however, trusted parties *do not* have access to the cloud nodes' management interface.

We assume that the TPMs are correct, and we do not consider side-channel attacks.

4 Policy-sealed Data

This section makes the case for our new trusted computing abstraction, called *policy-sealed* data. We first discuss the limitations of existing TPM abstractions in the context of the design of a strawman trusted cloud service. We then describe how policy-sealed data addresses these limitations.

4.1 Strawman Design of a Trusted Cloud Service

Our strawman trusted cloud service offers features similar to Amazon's EC2 but aims to provide better pro-

tection against the inspection or corruption of customer VMs by a cloud administrator.

The first step in designing the strawman is to protect the state of customer VMs running on cloud nodes. To do this, we use recent proposals from research and industry that offer such guarantees but *on a single node only*. For example, CloudVisor [53] retrofits Xen so that the hypervisor guarantees the integrity and confidentiality of data and software running in guest VMs even in the presence of a malicious system administrator. Customers can leverage the TPM's remote attestation capability to verify that a cloud node is running CloudVisor before uploading data to it.

However, this verification step checks these guarantees only for the cloud node on which the data is first uploaded. Once in the cloud, the customer's data and VMs often migrate from one node to another, or are suspended to disk and resumed at a later time. To offer end-to-end protection, the checks must be repeated upon such events.

Thus, to accommodate VM migration, the strawman design of a trusted EC2 must perform remote attestation each time a customer's VM migrates to verify that: (1) the destination node's identity is signed by the cloud provider, and (2) the fingerprint matches that of CloudVisor. To protect the VM upon suspension to disk, the VM state must be encrypted using sealed storage before suspension occurs.

4.2 Limitations of TPM Abstractions

The strawman design highlights some shortcomings of current TPM abstractions stemming from a fundamental principle upon which TPMs were built: they were designed to offer guarantees about one single computer. In particular, TPMs suffer from three major problems when they are used to build trusted cloud services.

First, the sealed storage abstraction was not designed for a distributed and dynamic environment like the datacenters where cloud services operate. It precludes the application developer from encrypting and storing sensitive data in an untrusted medium (e.g., a local hard drive, or the Amazon S3 service) and retrieving it from a different node or from the same node running a software configuration that differs from that in place when the data was encrypted. However, developers might be interested in suspending the VM to disk and resuming it later on a different node (e.g., if, in the interim, the original node was shut down to save power) or on the same node running a different configuration (e.g., if, in the interim, the hypervisor was upgraded to a more recent version).

Second, today's TPMs are not built for high performance. TPMs can execute only one command at a time, and many TPM commands, such as remote attestation,

Attribute	Value	Description
service	"EC2"	service name
version	"1"	version of the service
vmm	"Xen", "CloudVisor"	virtual machine monitor
type	"small", "large"	resources of a VM
country	"US", "DE"	country of deployment
zone	"Z1", "Z2", "Z3", "Z4"	availability zone

Table 1: Example of service attributes. In this case, EC2 supports two types of VM instances, two types of VMMs, and four availability zones (datacenters) in the US and Germany.

Node	Configuration
N	service : "EC2" ; version : "1" ; type : "small" ; country : "DE" ; zone : "Z2" ; vmm : "CloudVisor"

Table 2: Example of a node configuration. This configuration contains the values for the attributes that characterize the hardware and software of a specific node N .

Policy	Policy Specification
P_1	service = "EC2" and vmm = "CloudVisor" and version \geq "1" and instance = "large"
P_2	service = "EC2" and vmm = "CloudVisor" and (zone = "Z1" or zone = "Z3")
P_3	service = "EC2" and vmm = "CloudVisor" and country = "DE"

Table 3: Examples of policies. P_1 expresses version and VM instance type requirements, P_2 specifies a zone preference for different sites, and P_3 expresses a regional preference.

take approximately one second to complete. This inefficiency hampers the scalability of cloud services that use the TPM and can even open avenues for denial of service attacks if the TPM abstractions were invoked by customer-accessible operations.

Finally, the cloud infrastructure may be overexposed. By revealing TPM node identities and allowing customers to remotely attest the nodes, any outsider could learn, for instance: (1) the number of cloud nodes that constitute the infrastructure of the cloud provider, and (2) the distribution of different platforms they run. This information could be used by external attackers to trace vulnerabilities in the infrastructure, or by competitors to learn business secrets. Handing over such information is often unacceptable to cloud providers.

Recent proposals for TPMs in the cloud do not completely address these TPM limitations. Systems like Nexus [50] or CloudVisor [53] use TPMs to allow customers to remotely attest only a single cloud node and therefore do not address the preceding issues. Essentially, these systems address the complementary problem of securing the platform running on a single node. Our previous workshop paper [45] took preliminary steps to address some of these issues, but its solution did not handle situations where sensitive data needed to be secured persistently, which is unrealistic to assume on real-world cloud services; our prior solution also suffered from scalability limitations.

4.3 The Policy-sealed Data Abstraction

To overcome these limitations, we propose the new *policy-sealed data* abstraction. This abstraction allows customer data to be bound to cloud nodes whose configuration is specified by a customer-defined policy. Policy-sealed data offers two primitives for securing customer data: *seal* and *unseal*. Seal can be invoked anywhere – either on the customer’s computer or on the cloud nodes. It takes as input the customer’s data and a *policy* and outputs ciphertext. The reverse operation, *unseal*, can be invoked only on the cloud nodes that need to decrypt the data. *Unseal* takes as input the sealed data and decrypts it *if and only if* the node’s configuration satisfies the policy specified upon seal; otherwise, decryption fails.

With our abstraction, each cloud node has a configuration, which is a set of human-readable *attributes*. Attributes express features that refer to the node’s software (e.g., “vmm”, “version”) or hardware (e.g., “location”). A policy expresses a logical condition over the attributes supported by the provider (e.g., “vmm=Xen and location=US”). Table 1 shows an example of the attributes of a hypothetical deployment of a service akin to EC2. Table 2 illustrates the configuration of a particular node, and Table 3 lists example policies over node configurations in that deployment.

Our primitive can replace the existing remote attestation and sealed storage calls for securing customer data on the cloud. In particular, to protect data upon upload or migration, the customer needs only to seal the data to a policy: if the destination cannot unseal the data, then its configuration does not match the policy; therefore, the node is not trusted from the perspective of the customer who originally specified the policy.

5 Excalibur Design

This section presents Excalibur, a system that provides policy-sealed data support for building trusted cloud services.

5.1 Design Goals & Assumptions

Our central goal is to design and implement a system that offers the policy-sealed data primitive by making use of commodity TPMs. Furthermore, the system design must overcome the preceding limitations of the interface offered by current TPMs.

We focus on the design of the primitive used by the cloud platforms running on individual nodes. Therefore, we are not concerned with securing these platforms themselves. In particular, our goal is not to prevent the management interface exposed to cloud administrators from leaking or corrupting sensitive data (e.g., direct memory inspection of VM memory). Similarly, we require that the individual cloud platforms pro-

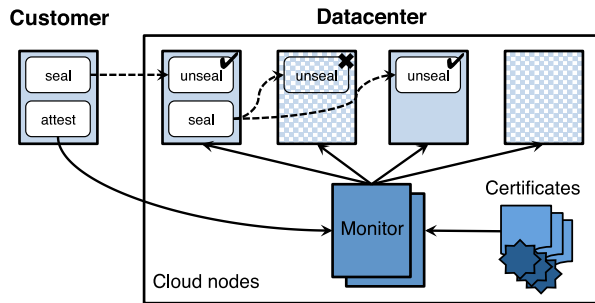


Figure 1: Excalibur deployment. The dashed lines show the flow of policy-sealed data, and the solid lines represent interactions between clients and the monitor. The monitor checks the configuration of cloud nodes. After a one-time monitor attestation step, clients can seal data. Data can be unsealed only on nodes that satisfy the policy (unshaded boxes).

protect certain key material used to seal and unseal data, and that the system interface does not allow the fingerprint stored in the TPM to be changed so that it becomes inconsistent with the current system state. To address these complementary goals, applications must make use of a series of existing systems and hardening techniques [20, 24, 33, 53].

5.2 System Overview

The design of Excalibur is based on a centralized component, called a *monitor*. The monitor is a dedicated service running on a single cloud node (or, as we will explain, on a small set of nodes for fault tolerance and scalability). It coordinates the enforcement of policy-sealed data on the entire cloud infrastructure by mapping TPM identities and fingerprints of the cloud nodes to policy-sealed data attributes. Only the monitor can trigger TPM primitives on the cloud nodes, minimizing the negative performance impact of TPM operations and preventing the exposure of infrastructure details.

Figure 1 illustrates a deployment of Excalibur, highlighting the separation between the two main system components: the *client* and the *monitor*. The client consists of a library that allows the implementation of a trusted cloud service to use the policy-sealed data primitives. This library can be used on both the customer side (e.g., before uploading data) and by the software platforms running on the cloud nodes (e.g., before migrating data between nodes). The customer-side client does not expose the unseal primitive since the notion of a configuration applies to cloud nodes only.

Whenever a cloud node reboots, the monitor runs a special remote attestation protocol to obtain the fingerprint and identity of the node and translates these to a node configuration by consulting an internal database. The node configuration — which expresses physical characteristics, like hardware or location, and software features as a set of attributes — is then encoded as cre-

dentials that are sent to the node. These credentials are required by cloud nodes to unseal policy-sealed data and are destroyed whenever the nodes reboot.

The monitor exposes a narrow management interface that lets the cloud administrator configure the mappings between attributes and identities (i.e., fingerprints). This is necessary for routing system maintenance as new software platforms and cloud nodes are deployed on the infrastructure. The management interface also allows multiple clones of the monitor to be securely spawned in order to scale up the system. To assure customers that it is properly maintained, the monitor accepts only mappings that are vouched for by special *certificates*; customers can directly attest the monitor in order to check its authenticity and integrity.

Though our high-level design is simple, we still need to overcome two main challenges: 1) to cryptographically enforce policies in a scalable, fault tolerant and efficient way, and 2) to assure customers that the monitor operates correctly despite the fact that it is managed by untrusted cloud administrators. To address these challenges, we: 1) use CPABE cryptography to enforce policies, and 2) devise certificates and a scalable monitor attestation mechanism to ensure that the monitor is trustworthy. We next explain these design choices in more detail.

5.3 Cryptographic Enforcement of Policies

The main challenge in implementing the seal and unseal primitives is avoiding scalability bottlenecks. A possible design is for the monitor itself to evaluate the policies: upon sealing, the client encrypts the data with a symmetric key and sends this key and the policy to the monitor; the monitor then encrypts this key and the policy with a secret key and returns the outcome to the client. To unseal, the encrypted key is sent to the monitor, which internally recovers the original symmetric key and policy, evaluates the policy, and releases the symmetric key if the node satisfies the policy. Although this solution implements the necessary functionality, it involves the monitor in every seal and unseal operation and thereby introduces a scalability bottleneck.

An alternative design is to evaluate the policies on the client side using public-key encryption. Each cloud node receives from the monitor a set of private keys that match its configuration; in this scheme, each key corresponds to an attribute-value pair of the configuration. Sealing is done by encrypting the data with the corresponding public keys according to the attributes defined in the policies. This solution avoids the bottlenecks of the first approach because all cryptographic operations take place on the client side, without involving the monitor. Its main shortcoming is complicated key manage-

ment due to the number of key-pairs that nodes must handle to reflect all possible attribute combinations usable by policies.

The solution we chose uses a cryptographic scheme called Ciphertext Policy Attribute-Based Encryption (CPABE) [11]. This scheme first generates a pair of keys: a public *encryption key* and a secret *master key*. Unlike traditional public key schemes, the encryption key allows a piece of data to be encrypted and bound to a *policy*. A policy is a logical expression that uses conjunction and disjunction operations over a set of terms. Each term tests a condition over an attribute, which can be a string or a number; both types support the equality operation, but the numeric type also supports inequalities (e.g., $a = x$ or $b > y$). CPABE can then create an arbitrary number of *decryption keys* from the same master key, each of which can embed a set of attributes specified at creation time. The encrypted data can be decrypted only by a decryption key whose attributes satisfy the policy (e.g., keys embedding the attribute $a = x$ can decrypt a piece of data encrypted with the preceding example policy).

Excalibur uses CPABE to encode the runtime configurations of the cloud nodes into decryption keys. At setup time, the monitor generates a CPABE encryption and master key pair and secures the master key. Whenever it checks the identity and software fingerprint of a cloud node, the monitor sends the appropriate credentials to the node, which include a CPABE decryption key embedding the attributes that correspond to the configuration of the node; the decryption key is created from the master key and forwarded to all the nodes featuring the same configuration. Sealing is done by encrypting the data using the encryption key and a policy, and unsealing is done by decrypting the sealed data using the decryption key. Policies are expressed in the CPABE policy language used to specify the examples in Table 3 as well as more elaborate policies.

The security of the system then depends on the security of the CPABE keys. The monitor protects the master key by: 1) ensuring that it cannot be released through the monitor's management interface, and 2) encrypting it before storing it on disk, as described in Section 6.3. Additionally, cloud platforms must protect decryption keys. A software platform must prevent leakage or corruption of key material through its management interface (e.g., by direct memory inspection of VM memory); it must hold the key in volatile memory so that key material is destroyed upon reboot. Moreover, the software platform must force a reboot after changing TCB components that get measured during a trusted boot (e.g., subsequent to upgrading the hypervisor). These properties ensure that the CPABE decryption keys of cloud nodes remain consistent with their

TPM fingerprints and therefore reflect current node configurations.

The benefits of using CPABE are twofold. First, it lets the system scale independently of the workload since the seal and unseal primitives do not interact with the monitor (and run entirely on the client side). Second, it permits the creation of expressive policies directly supported by the CPABE policy specification language while only requiring two keys – the CPABE encryption and decryption keys – to be sent to the nodes.

The cost using CPABE is a performance hit when compared to traditional cryptographic schemes. Section 6 explains how this impact can be minimized. A second cost of using CPABE is key revocation, which is typically difficult in identity- and attribute-based cryptosystems. Since Excalibur assumes that the TCB of nodes' software platforms is secure, any TCB vulnerability accessible through the administrator's interface will invalidate the guarantees provided by our system. To handle revocation of decryption keys, our current design requires that all sealed data whose original policy satisfies the attributes of the compromised keys be resealed. This operation can be done efficiently by re-encrypting only a symmetric key, not the data itself.

5.4 Trusting the Monitor

Since the monitor is managed by the cloud administrator, mismanagement threats that affect any cloud node could also affect the monitor. Thus, another challenge is to ensure that the monitor operates correctly and to efficiently convey this guarantee to customers.

To meet this challenge, we must first prevent the monitor from accepting flawed attribute mappings. For example, a mapping would be flawed if the attribute "location=DE" were mapped to the identity of a node located in the US, or if the attribute "vmm=Xen" were mapped to the fingerprint of CloudVisor. To prevent this, the monitor only accepts attribute mappings that are vouched for by a *certificate*. A certificate is issued by one or multiple *certifiers*, which validate the correctness of mappings. For example, a certifier checks the location of nodes and the fingerprints of software platforms. This role could be played by the provider itself, or by external trusted parties akin to Certification Authorities.

Since anyone can issue certificates, the monitor must let customers know the certifier's identity so they can judge the certifier's trustworthiness and thereby be confident that the attribute mappings are correct. Furthermore, even if the certifier were judged trustworthy, the system must nevertheless provide additional guarantees about the authenticity and integrity of the monitor: only in this case can the customer be sure that the certificate-based protections and the security proto-

cols implemented by the monitor are correct. To provide these guarantees, customers must directly attest the monitor when first using the system.

5.5 Monitor Scalability and Fault Tolerance

To improve scalability and make Excalibur resilient to faults, we enable several monitor replicas (*clones*) to be spawned, and we optimize the monitor attestation protocol.

Monitor clones can be elastically launched and terminated by the administrator, using the protocol described in Section 6.7. The cloud provider can then use standard load balancers to evenly distribute client attestation requests from clients among clones. Each clone can serve requests without communicating with other clones.

To eliminate critical bottlenecks within a clone, we introduce two optimizations. The first improves the throughput of clone attestations triggered by customers. Due to TPM inefficiencies, the maximum throughput of a monitor clone using a standard attestation protocol is close to one attestation per second, clearly insufficient even when spawning a reasonable number of clones. We therefore enhance the attestation protocol with a technique based on Merkle trees that can batch a large number of attestation requests into a single TPM quote (see Section 6).

A second optimization improves the throughput of decryption key requests issued by the cloud nodes. The algorithm for decryption key generation is also inefficient, which could significantly slow down servicing keys to the cloud nodes if a new key were to be generated per request. Since many machines in the datacenter share the same configuration (e.g., machines that belong to the same cluster), the monitor clone can instead securely cache the decryption keys and send them to all the nodes with the same profile.

6 Detailed Design

This section presents the design of Excalibur in more detail. We first introduce certificates, which constitute the root-of-trust of the system. We then describe the interfaces offered by Excalibur for building cloud services and managing the system. Finally, we present the security protocols that enforce policy-sealed data.

Notation. For CPABE keys, K^M , K^E and K^D denote master, encryption, and decryption keys, respectively. For asymmetric cryptography, K and K^P denote private and public keys, respectively. For symmetric keys, we drop the superscript. Notation $\langle x \rangle_K$ indicates data x encrypted with key K , and $\{y\}_K$ indicates data y signed with key K . We represent nonces as n . Session keys and nonces are randomly generated. Notation D , P , E , and

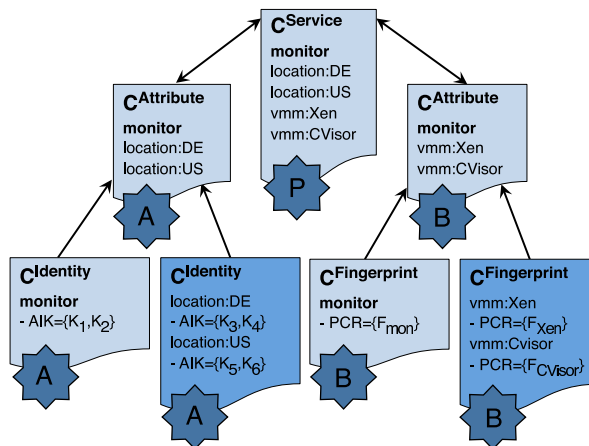


Figure 2: Example certificate tree. The certificates in light colored boxes form the *manifest* that validates the monitor’s authenticity and integrity.

M denote data, policy, envelope, and manifest; these terms are clarified in Section 6.2.

6.1 Certificate Specification

Excalibur uses certificates to validate mappings between attributes specific to a trusted cloud service and identities, i.e., fingerprints of cloud nodes. Certificates are used both by the monitor, to check the configuration of cloud nodes and attest new monitor clones, and by the customer-side client, to attest the monitor. Our certificate specification supports multiple certifiers since a single certifier may not have the expertise to assess all the attributes of the cloud service, or simply to increase customer trust. Therefore, certificates form a hierarchical tree. Figure 2 shows how a provider P can use the certificates that correspond to the internal nodes in the tree to delegate the certification of different attributes to two certifiers, A and B . Additionally, each leaf in the certificate tree vouches for a mapping between the attributes that appear in node configurations and low-level measurements, namely software fingerprints (PCRs) or hardware identities (AIK keys).

Due to space limitations, we defer a discussion of the details regarding the certification procedure, certificate expiration, certificate revocation, and certificate management to a separate technical report [46].

6.2 System Interfaces

Excalibur’s interface has two parts: a *service interface*, which supports the implementation of cloud services, and a *management interface*, which lets cloud administrators maintain the system.

The service interface exported by the client library supports three operations, summarized in Table 4. Before the data can be sealed on the customer-side, *attest-monitor* must be invoked to check the monitor’s authenticity and integrity. It returns the encryption key K^E

<code>attest-monitor(mon-addr)</code>	$\rightarrow (K^{\mathbb{E}}, M)$ or FAIL
<code>seal($K^{\mathbb{E}}, P, D$)</code>	$\rightarrow E = \langle P, D \rangle K, \langle K \rangle K^{\mathbb{E}}$
<code>unseal($K^{\mathbb{E}}, K^{\mathbb{D}}, E$)</code>	$\rightarrow (D, P)$ or FAIL

Table 4: Excalibur service interface.

needed for sealing and a *manifest* M , which contains the certificates needed to validate the monitor’s identity and fingerprint (see Figure 2). The manifest is passed to the customer, who learns from it which attributes can be used in policies and identifies the provider and certifier identities needed to decide whether the service is trustworthy. Since the client saves the manifest and encryption key for sealing, this operation needs to be performed only when the cloud service is first used.

The core primitives are *seal* and *unseal*. Seal can be invoked by both cloud nodes and customers; it takes as arguments the encryption key $K^{\mathbb{E}}$, a policy P , and the data D and produces an envelope E . This envelope is passed to *unseal*, which returns the decrypted data D or fails if its caller does not satisfy the policy. In addition to the decryption key $K^{\mathbb{D}}$, *unseal* receives as an argument the encryption key $K^{\mathbb{E}}$, which is required by CPABE decryption; the cloud node that invokes *unseal* must obtain this key from the monitor. *Unseal* also returns the original policy P so that a cloud node can re-seal the data with the customer’s policy. The CPABE policy language is used to express policies.

The management interface lets the cloud administrator remotely maintain the monitor using a console. Its main operations permit the administrator to initialize the system, manage certificates, and spawn monitor clones.

6.3 System Initialization

Before the system can be used, the monitor must be initialized by binding a unique CPABE key pair to the service. To do this, the cloud administrator loads the certificates that validate the service attributes into the monitor and instructs the monitor to generate the key pair. If these certificates form a consistent certificate tree, the monitor creates unique encryption and master keys and binds them to the tree’s root certificate (see Figure 2). To permit for system maintenance, the administrator can remove or add certificates as long as they form a valid certificate tree.

The monitor maintains its persistent state in a *certificate* store and a *key* store. Both stores keep their contents in XML files on a local disk. The certificate store contains the certificates loaded into the monitor. The key store contains all the CPABE keys. To secure the key material, the key store is sealed using the TPM seal primitive, which ensures that the key store can be accessed only under a trusted monitor configuration in case the monitor reboots.

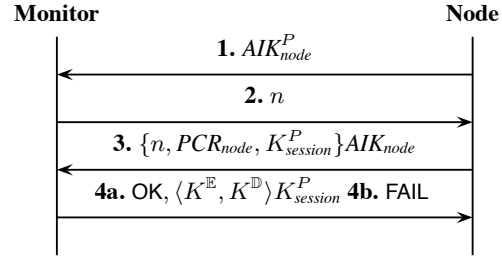


Figure 3: Node attestation protocol.

6.4 Node Attestation Protocol

Once the setup is complete, the monitor delivers to each cloud node a credential that reflects the boot time configuration of that node, which will allow the node to unseal and re-seal data. The goal of the node attestation protocol is to deliver these credentials securely. Recall that, under our assumptions, when a cloud node reboots, the credentials kept by the node in volatile memory are lost. Therefore, this protocol must be executed each time a cloud node reboots so it can obtain a fresh credential.

The monitor first obtains a quote from the node that is signed by the node’s AIK and contains the current PCRs. Then, the monitor looks in the certificate database for certificates that match the node’s PCRs and AIK. If any are found, the monitor obtains the node configuration by combining all the attributes of the matching certificates into a list like that shown in Table 2. Next, the monitor sends the credentials to the node; these include the encryption and decryption keys embedding these attributes. Since generating a new decryption key is expensive, the monitor caches these keys in the key store so they can be resent to nodes with the same configuration.

Figure 3 shows the precise messages exchanged between the monitor and the customer-side client. The protocol is based on a standard remote attestation in which a nonce n is sent to the node (message 2), and the node replies with a quote (message 3); the nonce is used to check the freshness of the attestation request. Message 3 includes a session key $K_{session}^P$ that is used in message 4 to securely send credentials $K^{\mathbb{E}}$ and $K^{\mathbb{D}}$ to the node. Since the session key is ephemeral, an adversary could not perform a TOCTOU attack by rebooting the machine after finishing attestation (message 3) but before receiving the decryption key (message 4).

Note that the node does not need to authenticate the monitor to preserve the security of policy-sealed data. In the worst case, a node may receive a compromised decryption key from an attacker. However, given that customers seal their data with the encryption key obtained from the legitimate monitor, *unseal* would fail in such a scenario, and this attack would fail to compromise customer data.

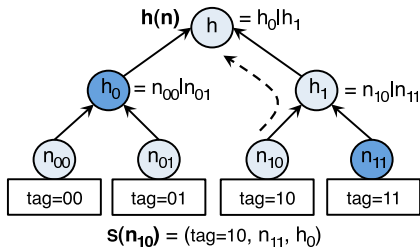


Figure 4: Batch attestation example. The tree is built from 4 nonces. A summary for nonce n_{10} comprises its tag and the hashes in the path to the root.

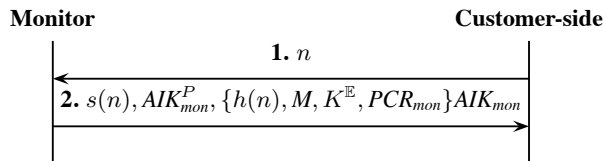


Figure 5: Monitor attestation protocol.

6.5 Monitor Attestation Protocol

The monitor attestation protocol is triggered by the *attest-monitor* operation, which lets customers detect if the monitor is legitimate by checking its authenticity and integrity. In addition, this protocol obtains: 1) the encryption key, which is used for sealing data, and 2) the set of certificates that form the manifest, which let the customer check the identity of certifiers and learn the attributes that are available. The monitor is legitimate if its identity and fingerprint are validated by the manifest.

The main challenge in designing this protocol is scalability. If every customer-side client were to run a standard remote attestation, then the throughput of the monitor would be extremely low due to TPM inefficiency.

To overcome this scalability problem, we batch multiple attestation requests into a single quote operation using a Merkle tree, as shown in Figure 4. The Merkle tree lets the monitor quote a batch of N nonces n_i expressed as an aggregate hash $h(n_{i=0}^N)$ and send an evidence – summary $s(n_i)$ – to each customer-side client that its nonce n_i is included in the aggregate hash in a network-efficient manner (i.e., instead of sending all N nonces, it sends just a summary of size $O(\log(N))$).

The detailed monitor attestation protocol is shown in Figure 5. In the first message, the customer-side client sends nonce n for freshness and then uses the information returned in message 2 to validate the monitor in two steps. First, it checks in the manifest M for the certificates with attribute “monitor”; it uses them to authenticate the monitor key AIK_{mon}^P and to validate the fingerprint of the monitor’s software platform PCR_{mon} (see Figure 2). Second, to validate the freshness of the received messages, it compares nonce n and the summary $s(n)$ against the aggregate hash $h(n)$ produced by batch attestation. If all tests pass, the monitor is trustworthy,

and the encryption key K^S is authentic. The customer can then seal data securely.

6.6 Seal and Unseal Protocols

The use of CPABE lets seal and unseal execute without contacting the monitor. In implementing these primitives, we take into account two aspects of CPABE related to performance and functionality. First, since CPABE is significantly more inefficient than symmetric encryption, seal encrypts the data with a randomly generated symmetric key and uses CPABE to encrypt the symmetric key. Second, given that CPABE decryption does not return the original policy (which unseal must return to let cloud nodes re-seal the data), we include in the envelope the original policy and a digest for integrity protection (see Table 4).

6.7 Clone Attestation Protocol

To scale the monitor elastically, the cloud administrator can create multiple monitor clones. To do so, an existing monitor instance must share the CPABE master key with the new clone so the latter can generate and distribute decryption keys to the cloud nodes. However, this can be done only if the new clone can be trusted to secure the key and to comply with the specification of Excalibur protocols.

To enforce this condition, the existing monitor instance and the clone candidate run a clone attestation protocol analogous to that shown in Figure 3, but with two differences. First, after message 3, the monitor assesses if the candidate is trustworthy by checking whether its AIK and PCR values map to the “monitor” attribute contained in the manifest; if not, cloning is aborted. Second, if the test passes, the monitor authorizes cloning and sends the master key, the encryption key, and a digest to the candidate. The digest identifies the head of the certificate tree associated with the keys. The new clone refrains from using the keys until the administrator uploads the corresponding certificates to it.

7 Implementation

We implemented Excalibur in about 22,000 lines of C. This included the monitor, a client-side library providing the service interface, a client-side daemon for securing the CPABE decryption key on the cloud nodes, a management console, and a certificate toolkit for issuing certificates. The console communicates with the monitor over SSL, and all other protocols used UDP messages. We used the OpenSSL crypto library [37] and the CPABE toolkit [8] for all cryptographic operations, and we used the Trousers software stack and its related tools [51] to interact with TPMs.

We extended a cloud service so it could use Excalibur to help us understand the effort needed to adapt services

```

1324 sock.send("receive\n")
1325 sock.recv(80)
1326
1327 pipe = subprocess.Popen("/xen-bin/seal",
1328     stdin=subprocess.PIPE,
1329     stdout=sock_FILENO())
1330 fd_pipe = pipe.stdin_FILENO()
1331
1332 XendCheckpoint.save(fd_pipe, dominfo, True,
1333     live, dst)
1334 os.close(fd_pipe)
1335 sock.close()

```

Figure 6: Hook to intercept migration (from file *XendDomain.py*.) We redirect the state of the VM through a process that seals the data before it proceeds to the destination on socket *sock* (lines 1327-1330).

for Excalibur and to estimate the performance impact of Excalibur on cloud services.

The example cloud service we adapted is an elastic VM service where customer VMs can be deployed in compute clusters in multiple locations, similar to Amazon’s EC2 service. Our extension used Excalibur to better assure customers that their VMs would not be accidentally or intentionally moved outside of a cluster in a certain area (e.g., the EU).

Our base platform was Eucalyptus [36], an open source system that provides an elastic VM service with an EC2-compatible interface. Eucalyptus supports various VMMs; we used Xen [9] because it is open source.

Our implementation modified Xen to invoke seal and unseal when the customer’s VM was created on a new node, migrated from one node to another, or suspended on one node and resumed on another. An attempt to migrate the VM to a node outside the specified locations would fail because the node would lack the credentials to unseal the policy-sealed VM.

Implementing these changes was straightforward. Integration with Excalibur required modifications to Xen, in particular to a Xen daemon called *xend*, which manages guest VMs on the machine and communicates with the hypervisor through the OS kernel of Domain 0. In particular, the VM operations *create*, *save*, *restore*, and *migrate* sealed or unsealed the VM memory footprint whenever the VM was unloaded from or loaded to physical memory, respectively. To streamline this implementation, we took advantage of the fact that *xend* always transfers VM state between memory and the disk or the network in a uniform manner using file descriptors. Therefore, we located the relevant file descriptors and redirected their operations through an OS process that sealed or unsealed according to the transfer direction. Figure 6 shows a snippet of *xend* that illustrates this technique applied to migration. Overall, our code changes were minimal: we added/modified 52 lines of Python code to *xend*.

The other two changes we made included: (1) hardening the software interfaces to prevent the system ad-

ministrators from invoking any VM operations other than the four noted above, and (2) using a TPM-aware boot-loader [5] to measure software integrity and to extend a TPM register with the Xen configuration fingerprint.

8 Evaluation

This section evaluates the correctness of Excalibur protocols using an automated tool. We also assess the performance of Excalibur and our example service.

8.1 Protocol Verification

We verified the correctness of our protocols using an automated theorem prover. We used a state-of-the-art tool, ProVerif [12], which supports the specification of security protocols for distributed systems in concurrent process calculus (pi-calculus).

To use the tool, we specified all protocols used by our system, which included all cryptographic operations (including CPABE operations), a simplified model of the TPM identity and fingerprint, the format of all certificate types in the system, the monitor protocols, and seal and unseal operations. In total, the specification contained approximately 250 lines of code in pi-calculus.

ProVerif proved the semantics of policy-sealed data in the presence of an attacker with unrestricted network access. The attacker could listen to messages, shuffle them, decompose them, and inject new messages into the network; this model covers, for example, eavesdropping, replay, and man-in-the-middle attacks. ProVerif proved that whenever a customer sealed data, the resulting envelope could be unsealed only by a node whose configuration matched the policy. We provide the specification and proof online [35].

8.2 Performance Evaluation

To evaluate Excalibur’s performance, we first evaluated the monitor’s scalability by measuring its performance overhead as well as its throughput for its three main activities: generating CPABE decryption keys, delivering these keys to nodes, and serving monitor attestation requests. We then measured the performance overhead of seal and unseal on the client side.

8.2.1 Setup and Methodology

We used two different experimental setups. The first used a two-node testbed; one node acted as a monitor, and the other acted as a regular cloud node making requests to the monitor. The second setup was used to evaluate the monitor throughput for attesting cloud nodes and serving customer attestation requests. For attesting cloud nodes, we simulated 1,000 nodes by using one machine acting as the monitor and five machines acting as cloud nodes, all running parallel instances of the node attestation protocol. For monitor attestations, we used a single machine acting as customers running

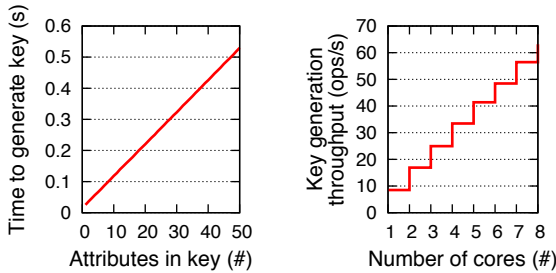


Figure 7: Performance of decryption key generation. Time to generate key as we vary the number of attributes (left), and throughput for 10 attributes as we vary the number of cores (right).

parallel instances of the monitor attestation protocol. This number of nodes was sufficient to exhaust monitor resources and ensure that there were no bottlenecks in the client nodes.

Both setups used Intel Xeon machines, each one equipped with 2.83GHz 8-core CPUs, 1.6GB of RAM, and TPM version 1.2 manufactured by Winbond. All machines ran Linux 2.6.29 and were connected to a 10Gbps network. We repeated each experiment ten times and reported median results; the standard deviation was negligible.

8.2.2 Decryption Key Generation

The overhead of generating a CPABE decryption key depends on the number of attributes embedded in the key. We measured the time to generate a decryption key stemming from the same master key, in which we varied the number of attributes from one to 50. This range seemed reasonable to characterize a node configuration.

Figure 7 shows the results, which confirm two relevant findings of the original authors of CPABE. First, the overhead of generating keys grows linearly with the number of attributes present in the key. Second, generating CPABE keys is expensive, e.g., a key with ten attributes took 0.12 seconds to create, which corresponds to a maximum rate of 8.33 keys/sec on a single core.

Although CPABE key generation is inherently inefficient, we consider that its performance is acceptable when throughput pressure on the monitor is relatively low because large groups of machines are likely to have the same configuration. The latency to generate a key is experienced only by the first node that reboots with a configuration new to the monitor. Since the key is cached, it is reused in future identical requests without additional costs.

8.2.3 Node Attestation

The latency of the node attestation protocol took 0.82 seconds. The bulk of the attestation cost (96%) was due to the node’s performing a TPM quote operation necessary for remote attestation. This result is not surprising since such operations are known to be inefficient [31].

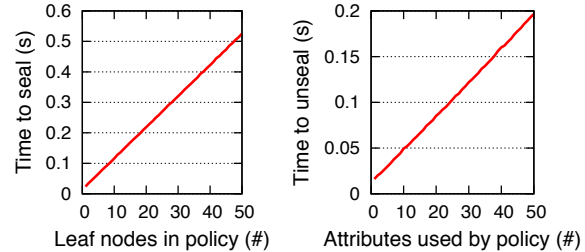


Figure 8: Performance overhead of sealing and unsealing data as a function of the complexity of the policy, with input data of constant size (1K bytes).

Most of the work required by this protocol is carried out by cloud nodes. Therefore, the attestation latency should not represent a bottleneck to the coordinator. To confirm this, we evaluated the monitor’s throughput when running multiple parallel instances of this protocol. Results showed that the monitor could deliver up to 632.91 keys per second, which is efficient and would allow a single monitor machine to scale to serve a large number of nodes.

8.2.4 Monitor Attestation

We measured the performance of the monitor attestation protocol. This protocol had a latency of 1.21 seconds and a throughput of approx. 4800 reqs/sec on a single node. The quote operation performed by the monitor’s local TPM accounted for the bulk of the latency (0.82 seconds), and the remaining time was due to cryptographic operations and network latency. The high peak throughput we observed was enabled by batch attestation. When we disabled batching, the throughput dropped sharply to 0.82 reqs/sec. Thus, this technique is crucial to the scalability of the monitor and delivered a throughput speedup of over 5000x.

8.2.5 Sealing and Unsealing

The performance overhead of the seal and unseal operations performed by Excalibur clients was dominated by the two cryptographic primitives: CPABE and symmetric cryptography (which uses AES with a 256-bit key size). We study their effects in turn.

To understand the overall performance overhead of CPABE, we set the input data to a small, constant size. Figure 8 shows the performance overhead of sealing and unsealing 1KB of data as a function of policy complexity. On the left is the cost of a seal operation as a function of the number of tests contained in the policy. For instance, policy $A=x$ and $(B=y \text{ or } B=z)$ contains three comparisons. Our findings show that the sealing cost grows linearly with the number of attributes. The cost of sealing for a policy with 10 attributes was about 128 milliseconds.

On the right, Figure 8 shows the cost of an unseal operation. Unlike encryption, CPABE decryption depends

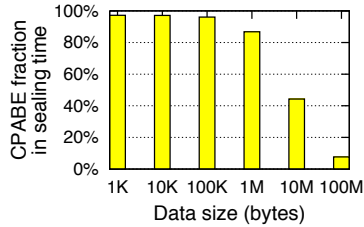


Figure 9: CPABE fraction in the performance overhead of sealing, varying the size of the input data.

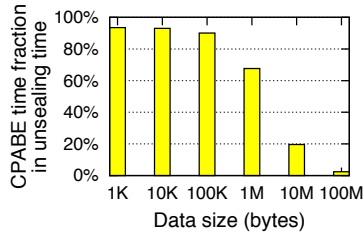


Figure 10: CPABE fraction in the performance overhead of unsealing, varying the size of the input data.

on the number of attributes in the decryption key that are used to satisfy the policy. For example, consider a decryption key with attributes $A:x$ and $B:y$, and policies $P_1 : A=x$, and $P_2 : A=x \text{ and } B=y$. Policy P_1 uses one attribute, whereas P_2 uses two. As before, the performance overhead of unseal grows linearly with the size of the policy. The time required to unseal a policy with 10 attributes was 51 milliseconds.

To study the relative effect of CPABE on the overall performance of Excalibur primitives, we varied the size of the input data. Figures 9 and 10 show the fraction of overhead due to CPABE, and Table 5 lists the absolute operation times. Our findings show that CPABE accounts for the most significant fraction of performance overhead. Sealing 1 MB of data with a policy containing 10 leaf nodes took 134 milliseconds, and 87% of the total cost of sealing was due to CPABE encryption. For unsealing, the fraction of CPABE was slightly lower than for sealing, but it was still very significant. Unsealing 1 MB of data with a policy satisfying 10 attributes of the private key took 68 milliseconds, where 68% of the latency was due to CPABE.

In summary, our evaluation of Excalibur showed these results: the costs of generating decryption keys and the node attestation protocol were reasonable when taking into account how infrequently they are required; the monitor scaled well with the number of cloud customers that used the service for the first time and with the number of cloud nodes that were attested upon re-boot; the monitor could be further scaled up using cloning, and the latency of seal and unseal was reasonable and dominated by the cost of symmetric key encryption for large data items.

Data (bytes)	Latency (ms)	
	Sealing	Unsealing
1K	120	50
10K	120	49
100K	121	51
1M	134	68
10M	264	243
100M	1522	1765

Table 5: Performance overhead of sealing and unsealing data, varying the size of the input data.

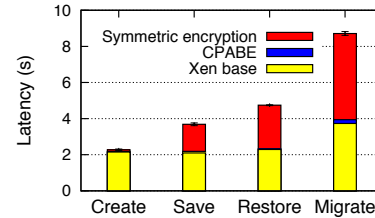


Figure 11: Latency of VM operations in Xen. Encrypting the VM state accounts for the largest fraction of the overhead, while the execution time of CPABE is relatively small. Encryption runs AES with 256-bit key size.

8.3 Cloud Compute Service

We now evaluate the performance overhead that the changes to Xen incur on its VM management operations, namely *create*, *save*, *restore* and *migrate*. We measured the time to complete each operation using an example VM for 10 trials. The example VM ran a Debian Lenny distribution, with Linux-xen 2.6.26, used a 4GB disk image, and its memory footprint was 128MB.

Figure 11 shows the results of our experiments. The performance impact is noticeable, especially for the *save*, *restore*, and *migrate* operations, where the completion time roughly doubled. The overhead, however, came from encrypting the VM’s entire memory footprint; using Excalibur to secure or recover the encryption key added a small delay. Unlike the other operations, *create* experienced a small overhead increase of only 4%. This is because the system only decrypted the kernel image, which occupied 4.6MB, instead of the larger VM footprint as it did for the other operations.

As the results show, seal and unseal introduced noticeable overhead to the VM operations due to the symmetric encryption of the VM image. However, given that these operations occur infrequently, and considering the additional benefits to data security, we argue that these results reflect an acceptable trade-off between security and performance.

9 Related Work

Over the past several years, there has been considerable work on trusted computing [38]. Most of this work targets single computers with the goal of enforcing application runtime protection [16,20,26,30,31], virtualizing

trusted computing hardware [10], and devising remote attestation solutions based on both software [18,48] and hardware [13,21,42–44,49]. Other work, focusing on distributed environments, provides integrity protection on shared testbeds [14] or distributed mandatory access control [29]. More recently, trusted computing primitives have been adapted to mobile scenarios to provide increased assurances about the authenticity of data generated by sensor-equipped smartphones [27]. Our work concentrates on the specific challenges of cloud computing environments, which fall outside the scope of these prior efforts.

Excalibur shares some ideas with property-based attestation [42], whose goal is to make hash-based software fingerprints more meaningful to humans. Like Excalibur, property-based attestation maps low-level fingerprints to high level attributes (properties) and relies on a monitor (controller) to perform this mapping. However, this prior work offers an abstract model without an associated system. Moreover, Excalibur extends this work by proposing new trusted computing primitives.

Nexus [50], a new operation system for trustworthy computing, introduces active attestation, which allows attesting a program’s application-specific runtime properties and supports access control policies per application. Both Nexus policies and policy-sealed data can bind data based on attributes. However, the two systems target complementary problems: Nexus policies focus on nodes running Nexus and restrict the applications that can access the data; Excalibur policies focus on multi-node settings and restrict the cloud nodes that can access the data, supporting multiple software platforms. Thus, Nexus would be a good candidate to use as an attribute in an Excalibur policy.

The work by Schiffman et al. [47] aims to improve the transparency of IaaS cloud services by providing customers with integrity proofs of their VMs and underlying VMMs. Like Excalibur, a central component, called cloud verifier (CV), mediates attestations of nodes and uses high-level properties (attributes) for reasoning about node configurations. However, the scope of this work is narrower than ours: while the CV provides only integrity proofs, Excalibur builds on these proofs to enforce policy-sealed data, which is a general, data-centric abstraction for protecting customer data in the cloud. In addition, the CV administrator is assumed to be trustworthy, representing a weaker threat model; in our view, this assumption does not address an important class of problems that occur in cloud services today. Finally, their system does not address the shortcomings of sealed storage TPM primitives, which could raise concerns of data management inflexibility and isolation crippling if these primitives need to be used by cloud services to secure persistent data.

Multiple software systems have been proposed to increase the security of sensitive data. At the OS layer, hypervisors and OSES can protect the confidentiality and integrity of data using isolation [24,30,39,53] or information flow control [52] techniques. At the middleware layer, frameworks that build Web services to offer their users strict control over their data hosted at the provider site [22] enable controlled sharing of sensitive data using differential privacy [41] or provide general-purpose encapsulation mechanisms for data [28]. These proposals are complementary to our work: despite their potential to increase security and control over data in the cloud, these proposals lack a scalable mechanism for bootstrapping trust in the multi-node cloud environment. By combining these platforms with Excalibur, cloud providers could build new trusted cloud services.

10 Conclusion

This paper presented Excalibur, a system that implements policy-sealed data. This new abstraction addresses the limitations of trusted computing when used in the cloud and enables the design of trusted cloud services. Excalibur leverages TPMs, a novel architecture with a set of associated protocols, and CPABE to offer developers two new primitives, seal and unseal, for constructing cloud services with stronger protection over how data is managed. We demonstrated the simplicity and flexibility of policy-sealed data by using Excalibur to build an elastic VM cloud computing service based on Xen and Eucalyptus, which accesses customer’s data only on customer-approved platform configurations.

Acknowledgements: We would like to thank Peter Drushel, Pedro Fonseca, Aniket Kate, Jay Lorch, Massimiliano Marcon, Bryan Parno, Himanshu Raj, and Alec Wolman for their valuable comments and conversations that improved our work. We are also grateful to the anonymous reviewers and Mihai Christodorescu, our shepherd, for their feedback.

References

- [1] Blippy Users Credit Card Numbers Exposed in Google Search Results. <http://mashable.com/2010/04/23/blippy-credit-card-numbers>.
- [2] Cloudcamp: Five key concerns raised about cloud computing. http://www.itnews.com.au/News/223980_cloudcamp-five-key-concerns-raised-about-cloud-computing.aspx.
- [3] Federal Government’s Cloud Plans: A \$20 Billion Shift. http://www.cio.com/article/671013/Federal_Government_s_Cloud_Plans_A_20_Billion_Shift.
- [4] T-mobile: All your sidekick data has been lost forever. <http://mashable.com/2009/10/10/t-mobile-sidekick-data>.
- [5] Trusted GRUB. <http://trousers.sourceforge.net/grub.html>.

- [6] Verizon to Put Medical Records in the Cloud. <http://www.networkcomputing.com/cloud-computing/229501444>.
- [7] Insecurity of Privileged Users: Global Survey of IT Practitioners. Technical report, Ponem Institute and HP, 2011. <http://h30507.www3.hp.com/hpblogs/attachments/hpblogs/666/62/1/HP%20Privileged%20User%20Study%20FINAL%20December%202011.pdf>.
- [8] Advanced Crypto Software Collection. <http://acsc.cs.utexas.edu>.
- [9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.
- [10] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: virtualizing the trusted platform module. In *USENIX Security Symposium*, 2006.
- [11] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-policy attribute-based encryption. In *Symposium on Security and Privacy*, 2007.
- [12] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *CSFW*, 2001.
- [13] E. Brickell, J. Camenisch, and L. Chen. Direct Anonymous Attestation. In *CCS*, 2004.
- [14] C. Cutler, M. Hibler, E. Eide, and R. Ricci. Trusted disk loading in the Emulab network testbed. In *WCSET*, 2010.
- [15] ENISA. Cloud Computing - SME Survey, 2009. <http://www.enisa.europa.eu/act/rm/files/deliverables/cloud-computing-sme-survey/>.
- [16] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *SOSP*, 2003.
- [17] T. C. Group. TPM Main Specification Level 2 Version 1.2, Revision 130, 2006.
- [18] V. Haldar, D. Chandra, and M. Franz. Semantic Remote Attestation - A Virtual Machine directed approach to Trusted Computing. In *VM*, 2004.
- [19] J. Hamilton. An Architecture for Modular Data Centers. In *CIDR*, 2007.
- [20] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The Nizza secure-system architecture. *CollaborateCom*, 2005.
- [21] T. Jaeger, R. Sailer, and U. Shankar. PRIMA: policy-reduced integrity measurement architecture. In *SACMAT*, 2006.
- [22] J. Kannan, P. Maniatis, and B.-G. Chun. Secure data preservers for web services. In *WebApps*, 2011.
- [23] M. Keeney. Insider Threat Study: Computer System Sabotage in Critical Infrastructure Sectors. Technical report, U.S. Secret Service and CMU, 2005. http://www.secretservice.gov/ntac/its_report_050516.pdf.
- [24] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, 2009.
- [25] E. Kowalski. Insider Threat Study: Illicit Cyber Activity in the Information Technology and Telecommunications Sector. Technical report, U.S. Secret Service and CMU, 2008. http://www.secretservice.gov/ntac/final_it_sector_2008_0109.pdf.
- [26] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *ASPLOS*, 2000.
- [27] H. Liu, S. Saroiu, A. Wolman, and H. Raj. Software abstractions for trusted sensors. In *MobiSys*, 2012.
- [28] P. Maniatis, D. Akhawe, K. Fall, E. Shi, S. McCamant, and D. Song. Do You Know Where Your Data Are? Secure Data Capsules for Deployable Data Protection. In *HotOS*, 2011.
- [29] J. M. McCune, T. Jaeger, S. Berger, R. Cáceres, and R. Sailer. Shamon: A System for Distributed Mandatory Access Control. In *ACSAC*, 2006.
- [30] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. D. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*, 2010.
- [31] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *EuroSys*, 2008.
- [32] Microsoft. BitLocker Drive Encryption. <http://www.microsoft.com/whdc/system/platform/hwsecurity/default.mspx>.
- [33] A. G. Miklas, S. Saroiu, A. Wolman, and A. D. Brown. Bunker: a privacy-oriented platform for network tracing. In *NSDI*, 2009.
- [34] D. G. Murray, G. Milos, and S. Hand. Improving Xen security through disaggregation. In *VEE*, 2008.
- [35] N. Santos. ProVerif scripts for the Excalibur protocols, 2011. <http://www.mpi-sws.org/~nsantos/excalibur/xcproverif.zip>.
- [36] D. Nurmi, R. Wolski, C. Grzegorzczak, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. Eucalyptus: A Technical Report on an Elastic Utility Computing Architecture Linking Your Programs to Useful Systems. Technical Report 2008-10, UCSB.
- [37] OpenSSL. <http://www.openssl.org>.
- [38] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping trust in commodity computers. In *IEEE Symposium on Security and Privacy*, 2010.
- [39] H. Raj, D. Robinson, T. B. Tariq, P. England, S. Saroiu, and A. Wolman. Credo: Trusted Computing for Guest VMs with a Commodity Hypervisor. Technical Report MSR-TR-2011-130, Microsoft Research, 2011.
- [40] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get Off of My Cloud! Exploring Information Leakage in Third-Party Compute Clouds. In *CCS*, 2009.
- [41] I. Roy, S. T. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for mapreduce. In *NSDI*, 2010.
- [42] A.-R. Sadeghi and C. Stübke. Property-based attestation for computing platforms: caring about properties, not mechanisms. In *NSPW*, 2004.
- [43] R. Sailer, T. Jaeger, X. Zhang, and L. van Doorn. Attestation-based policy enforcement for remote access. In *CCS*, 2004.
- [44] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *USENIX Security Symposium*, 2004.
- [45] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards trusted cloud computing. In *HotCloud*, 2009.
- [46] N. Santos, R. Rodrigues, K. Gummadi, and S. Saroiu. Excalibur: Building Trustworthy Cloud Services. Technical Report MPI-SWS-2011-004, MPI-SWS, 2011.
- [47] J. Schiffman, T. Moyer, H. Vijayakumar, T. Jaeger, and P. McDaniel. Seeding clouds with trust anchors. In *WCCS*, 2010.
- [48] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. Swatt: Software-based attestation for embedded devices. *IEEE Symposium on Security and Privacy*, 2004.
- [49] E. Shi, A. Perrig, and L. V. Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *IEEE Symposium on Security and Privacy*, 2005.
- [50] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical Attestation: An Authorization Architecture for Trustworthy Computing. In *SOSP*, 2011.
- [51] TrouSerS. <http://trousers.sourceforge.net>.
- [52] S. Vandeboogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the asbestos operating system. *ACM Trans. Comput. Syst.*, 2007.
- [53] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *SOSP*, 2011.

STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud

Taesoo Kim
MIT CSAIL

Marcus Peinado
Microsoft Research

Gloria Mainar-Ruiz
Microsoft Research

Abstract

Cloud services are rapidly gaining adoption due to the promises of cost efficiency, availability, and on-demand scaling. To achieve these promises, cloud providers share physical resources to support multi-tenancy of cloud platforms. However, the possibility of sharing the same hardware with potential attackers makes users reluctant to off-load sensitive data into the cloud. Worse yet, researchers have demonstrated side channel attacks via shared memory caches to break full encryption keys of AES, DES, and RSA.

We present STEALTHMEM, a system-level protection mechanism against cache-based side channel attacks in the cloud. STEALTHMEM manages a set of locked cache lines per core, which are never evicted from the cache, and efficiently multiplexes them so that each VM can load its own sensitive data into the locked cache lines. Thus, any VM can hide memory access patterns on confidential data from other VMs. Unlike existing state-of-the-art mitigation methods, STEALTHMEM works with existing commodity hardware and does not require profound changes to application software. We also present a novel idea and prototype for isolating cache lines while fully utilizing memory by exploiting architectural properties of set-associative caches. STEALTHMEM imposes 5.9% of performance overhead on the SPEC 2006 CPU benchmark, and between 2% and 5% overhead on secured AES, DES and Blowfish, requiring only between 3 and 34 lines of code changes from the original implementations.

1 Introduction

Cloud services like Amazon's Elastic Compute Cloud (EC2) [5] and Microsoft's Azure Service Platform (Azure) [26] are rapidly gaining adoption because they offer cost-efficient, scalable and highly available computing services to their users. These benefits are made possible by sharing large-scale computing resources among a large

number of users. However, security and privacy concerns over off-loading sensitive data make many end-users, enterprises and government organizations reluctant to adopt cloud services [18, 20, 25].

To offer cost reductions and efficiencies, cloud providers multiplex physical resources among multiple tenants of their cloud platforms. However, such sharing exposes multiple side channels that exist in commodity hardware and that may enable attacks even in the absence of software vulnerabilities. By exploiting side channels that arise from shared CPU caches, researchers have demonstrated attacks extracting encryption keys of popular cryptographic algorithms such as AES, DES, and RSA. Table 1 summarizes some of these attacks.

Unfortunately, the problem is not limited to cryptography. Any algorithm whose memory access pattern depends on confidential information is at risk of leaking this information through cache-based side channels. For example, attackers can detect the existence of `sshd` and `apache2` via a side channel that results from memory deduplication in the cloud [38].

There is a large body of work on countermeasures against cache-based side channel attacks. The main directions include the design of new hardware [12, 23, 24, 41–43], application specific defense mechanisms [17, 28, 30, 39] and compiler-based techniques [11]. Unfortunately, we see little evidence of general hardware-based defenses being adopted in mainstream processors. The remaining proposals often lack generality or have poor performance.

We solve the problem by designing and implementing a system-level defense mechanism, called STEALTHMEM, against cache-based side channel attacks. The system (hypervisor or operating system) provides each user (virtual machine or application) with small amounts of memory that is largely free from cache-based side channels. We first design an efficient software method for locking the pages of a virtual machine (VM) into the shared cache, thus guaranteeing that they cannot be evicted by other VMs. Since different processor cores might be running

Type	Enc.	Year	Attack description	Victim machine		Samples	Crypt. key
Active Time-driven [9]	AES	2006	Final Round Analysis	UP	Pentium III	$2^{13.0}$	Full 128-bit key
Active Time-driven [30]	AES	2005	Prime+Evict (Synchronous Attack)	SMP	Athlon 64	$2^{18.9}$	Full 128-bit key
Active Time-driven [40]	DES	2003	Prime+Evict (Synchronous Attack)	UP	Pentium III	$2^{26.0}$	Full 56-bit key
Passive Time-driven [4]	AES	2007	Statistical Timing Attack (Remote)	SMT	Pentium 4 with HT	$2^{20.0}$	Full 128-bit key
Passive Time-driven [8]	AES	2005	Statistical Timing Attack (Remote)	UP	Pentium III	$2^{27.5}$	Full 128-bit key
Trace-driven [14]	AES	2011	Asynchronous Probe	UP	Pentium 4 M	$2^{6.6}$	Full 128-bit key
Trace-driven [29]	AES	2007	Final Round Analysis	UP	Pentium III	$2^{4.3}$	Full 128-bit key
Trace-driven [3]	AES	2006	First/Second Round Analysis	-	-	$2^{3.9}$	Full 128-bit key
Trace-driven [30]	AES	2005	Prime+Probe (Synchronous Attack)	SMP	Pentium 4 with HT	$2^{13.0}$	Full 128-bit key
Trace-driven [32]	RSA	2005	Asynchronous Probe	SMT	Xeon with HT	-	310-bit of 512-bit key

Table 1: Overview of cache-based side channel attacks: UP, SMT and SMP stand for uniprocessor, simultaneous multithreading and symmetric multiprocessing, respectively.

different VMs at the same time, we assign a set of locked cache lines to each core, and keep the pages of the currently running VMs on those cache lines. Therefore each VM can use its own special pages to store sensitive data without revealing its usage patterns. Whenever a VM is scheduled, STEALTHMEM ensures the VM’s special pages are loaded into the locked cache lines of the current core. Furthermore, we describe a method for locking pages without sacrificing utilization of cache and memory by exploiting an architectural property of caches (set associativity) and the cache replacement policy (pseudo-LRU) in commodity hardware.

We apply this locking technique to the last level caches (LLC) of modern x64-based processors (usually the L2 or L3 cache). These caches are particularly critical as they are typically shared among several cores, enabling one core to monitor the memory accesses of other cores. STEALTHMEM prevents this for the locked pages. The LLC is typically so large that the fraction of addresses that maps to a single cache line is very small, making it possible to set aside cache lines without introducing much overhead. In contrast, the L1 cache of a typical x64 processor is not shared and spans only a single 4 kB page. Thus, we do not attempt to lock it.

We use the term “locking” in a conceptual sense. We have no hardware mechanism for locking cache lines on mass market x64 processors. Instead, we use a hypervisor to control memory mappings such that the protected memory addresses are guaranteed to stay in the cache, irrespective of the sequence of memory accesses made by software. While the cloud was our main motivation, our techniques are not limited to the cloud and can be used to defend against cache-based side channel attacks in a general setting.

Our experiments show that our prototype of the idea on Windows Hyper-V efficiently mitigates cache-based side channel attacks. It imposes a 5.9% performance overhead on the SPEC 2006 CPU benchmark running with 6 VMs. We also adapted standard implementations of three common block ciphers to take advantage of STEALTHMEM. The code changes amounted to 3 lines for Blowfish, 5 lines for DES and 34 lines for AES. The overheads of the secured versions were 3% for DES, 2% for Blowfish and

Level	Shared	Type	Line size	Assoc.	Size
L1	No	Inst./Data	64 Bytes	4/8	32 kB/32 kB
L2	No	Unified	64 Bytes	8	256 kB
L3	Yes	Unified	64 Bytes	16	8 MB

Table 2: Caches in a Xeon W3520 processor

5% for AES.

2 Background

This section provides background on the systems STEALTHMEM is intended to protect, focusing on CPU caches and the channels through which cache information can be leaked. It also provides an overview of known cache-based side channel attacks.

2.1 System Model

We target modern virtualized server systems. The hardware is a shared memory multiprocessor whose processing cores share a cache (usually the last level cache). The CPUs may support simultaneous multi-threading (Hyper-Threading). The system software includes a hypervisor that partitions the hardware resources among multiple tenants, running in separate virtual machines (VMs). The tenants are not trusted and may not trust each other.

2.1.1 Cache Structure

The following short summary of caches is specific to typical x64-based CPUs, which are the target of our work. The CPU maps physical memory addresses to cache addresses (called *cache indices*) in n -byte aligned units. These units are called *cache lines*, and mapped physical addresses are called *pre-image sets* of each cache line as in Figure 1. A typical value of n is 64. We call the number of possible cache indices the *index range*. We call the index range times the line size, the *address range of the cache*.

On x64 systems, caches are typically *set associative*. Every cache index is backed by cache storage for some number $w > 1$ of cache lines. Thus, up to w different lines of memory that map to the same cache index can

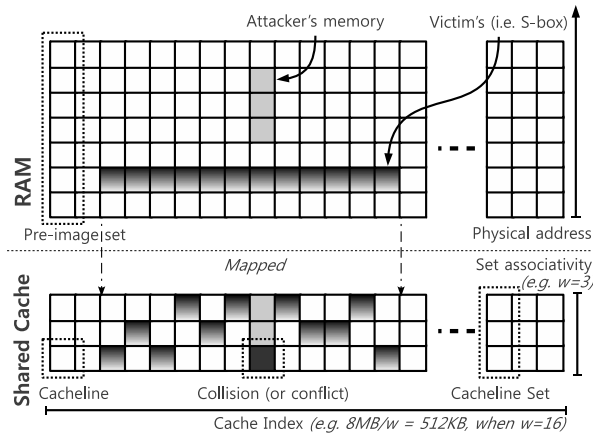


Figure 1: Cache structure and terminology

be retained in the cache simultaneously (see Figure 1). The number w is called the *wayness* or *set associativity*, and typical values are 8 and 16, as in Table 2. Since w cache lines have the same *pre-image sets* (correspondingly mapped physical memory), we refer to all w cache lines as a *cache line set*.

CPUs typically implement a logical hierarchy of caches, called L1, L2 and L3 depending on where they are located. L1 is physically closest to CPU, so it is the fastest (about 4 cycles), but has the smallest capacity (e.g., 32 kB). In multi-core architectures (e.g., Xeon), each core has its own L1 and backed L2 cache. The L3 cache, usually the last level cache, is the slowest (about 40 cycles) and largest cache (e.g., 8 MB). It is shared by all cores of a processor. The L3 is particularly interesting because it can be shared among virtual machines running concurrently on different cores.

2.1.2 Cache Properties

This section lists two well-known properties of caches that our algorithms rely on. The first condition is the basis for our main algorithm. We will also describe an optimization that is possible if the cache has the second property.

Inertia No cache line of a cache line set will be evicted unless there is an attempt to add another item to the cache line set. In other words, the current contents of each cache line set stay in the cache until an address is accessed that is not in the cache and that maps to the same cache line set. That is, cache lines are not spontaneously forgotten. The only exceptions are CPU instructions to flush the cache such as `invd` or `wbinvd` on x64 CPUs. However, such instructions are privileged and can be controlled by a trusted hypervisor.

k -LRU Cache lines are typically evicted according to a pseudo-LRU cache replacement policy. Under an LRU replacement policy, the least recently used cache line is evicted, assuming that cache line is not likely to be utilized in the near future. Pseudo-LRU is an approximation to LRU which is cheaper to implement in hardware. We say that an associative cache has the *k -LRU property* if the replacement algorithm will never evict the k most recently used copies. The k is not officially documented by major CPU vendors and may also differ by micro architectures and their implementations. We will perform an experiment to find the proper k for our Xeon W3520 in Section 5.

2.1.3 Leakage Channels

This section summarizes the different ways in which information can leak through caches (see Figure 2). These leakage channels form the basis for *active time-driven attacks* and *trace-driven attacks* that we will define in the next section.

Preemptive scheduling An attacker's VM and a victim's VM may share a single CPU core (and its cache). The system uses preemptive scheduling to switch the CPU between the different VMs. Upon each context switch from the victim to the attacker, the attacker can observe the cache state as the victim had left it.

Hyper-Threading Hyper-Threading is a hardware technology that allows multiple (typically two) hardware threads to run on a single CPU core. The threads share a number of CPU resources, including the ALU and all of the core's caches. This gives rise to a number of side channels, and scheduling potentially adversarial VMs on Hyper-Threading of the same core is generally considered to be unsafe.

Multicore The attacker and the victim may be running concurrently on separate CPU cores with a shared L3 cache. In this case, the attacker can try to probe the L3 cache for accesses by the victim while the victim is running.

2.2 Cache-based Side Channel Attacks

In this section, we summarize and classify well-known cache-based side channel attacks. Following Page [31], we distinguish between time-driven and trace-driven cache attacks, based on the information that is leaked in the attacks. Furthermore, we classify time-driven attacks as *passive* or *active*, depending on the scope of the attacks.

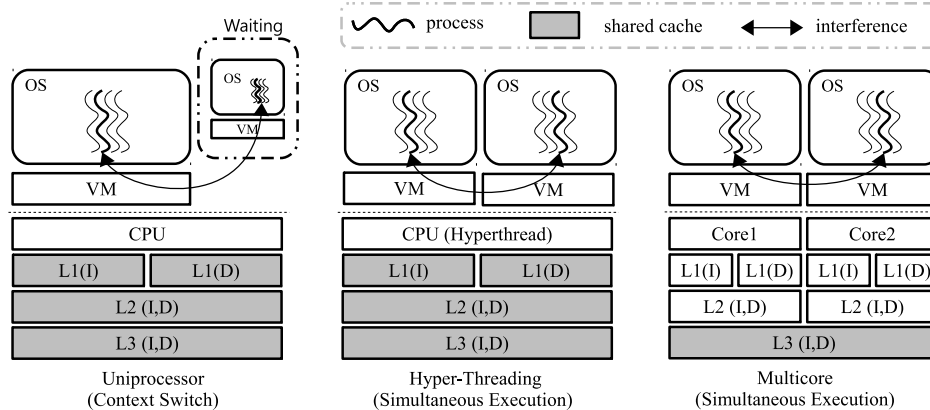


Figure 2: Leakage channels in three VM settings—uniprocessor, Hyper-Threading and multicore architectures. Modern commodity multicore machines suffer from all of three types of cache-based side channels. The letters (I) and (D) indicate instruction-cache and data-cache, respectively.

2.2.1 Time-driven Cache Attacks

The first class of attacks are time-driven cache attacks, also known as timing attacks. Memory access times depend on the state of the cache. This can result in measurable differences in execution times for different inputs. Such timing differences could be converted into meaningful attacks such as inferring cryptographic keys. For example, the number of cache lines accessed by a block cipher during encryption may depend on the key and on the plaintext, resulting in differences in execution times. Such differences may allow an attacker to derive the key directly or to reduce the possible key space, making it possible to extract the complete key within a feasible amount of time by brute force search.

Depending on the location of the attacker, the time-driven cache attacks fall into two categories: passive and active attacks. A passive attacker has no direct access to the victim’s machine. Thus the attacker cannot manipulate or probe the victim’s cache directly. Furthermore, he does not have access to precise timers on the victim’s machine. An active attacker, on the other hand, can run code on the same machine as the victim. Thus, the attacker can directly manipulate the cache on the victim’s machine. He can also access precise timers on that machine.

Passive time-driven cache attacks The time measurements in passive attacks are subject to two sources of noise. The initial state of the cache, which passive attackers cannot directly manipulate or observe, may influence the running time. Furthermore, since the victim’s running time cannot be measured locally with a high precision timer, the measurement itself is subject to noise (e.g. due to network delays). Passive attacks, therefore, generally require more samples and try to reduce the noise by means of statistical methods.

For example, Bernstein’s AES attack [8] exploits the

fact that the execution time of AES encryption varies with the number of cache misses caused by S-box table lookups during encryption. The indices of the S-box lookups depend on the cryptographic key and the plaintext chosen by the attacker. After measuring the execution times for a sufficiently large number of carefully chosen plaintexts, the attacker can infer the key after performing further offline analysis.

Active time-driven cache attacks Active attackers can directly manipulate the cache state, and thus can induce collisions with the victim’s cache lines. They can also measure the victim’s running time directly using a high precision timer of the victim. This eliminates much of the noise faced by passive attackers, and makes active attacks more efficient. For example, Osvik et al. [30] describe an active timing attack on AES which can recover the complete 128-bit AES key from only 500,000 measurements. In contrast, Bernstein’s passive timing attack required $2^{27.5}$ measurements.

2.2.2 Trace-driven Cache Attacks

The second type of cache-based side channel attacks are trace-driven attacks. These attacks try to observe which cache lines the victim has accessed by probing and manipulating the cache. Thus, like active timing attacks, trace-driven attacks require attackers to access the same machine as the victim. Given the additional information about access patterns of cache lines, trace-driven attacks have the potential of being more efficient and sophisticated than time-driven attacks.

A typical attack strategy (Prime+Probe) is for the attacker to access certain memory addresses, thus filling the cache with its own memory contents (Prime). Later, the attacker measures the time required to access the same memory addresses again (Probe). A large access time

indicates a cache miss which, in turn, may indicate that the victim accessed a pre-image of the same cache line.

Trace-driven attacks were considered harmful especially with simultaneous multi-threading technologies, such as Hyper-Threading, that enable one CPU to execute multiple hardware threads at the same time without a context switch. By exploiting the fact that both threads share the same processor resources, such as caches, Percival [32] experimentally demonstrated a trace-driven cache attack against RSA. The attacker's process monitoring L1 activity of RSA encryption can easily distinguish the footprints of modular squaring and modular multiplications based on the Chinese Remainder Theorem, which is used by various RSA implementations to compute modular operations on the private key of RSA [32].

More severely, Neve [29] introduced another trace-driven attack even without requiring multi-threading technologies. Within a single-threaded processor, Neve analyzed the last round of AES encryption with multiple footprints of the AES process. To gain a footprint, Neve's attack exploits the preemptive scheduling policy of commodity operating systems. Gullasch et al. similarly used the Completely Fair Scheduler of Linux to extract full AES encryption keys. This is the first fully functional asynchronous attack in a real-world setting.

More quantitative research on trace-driven cache-based side channel attacks was conducted by Osvik, Shamir and Tromer [30, 39]. They demonstrated two interesting AES attacks by analyzing the first and second round of AES. The first attack (Prime+Probe) was able to recover a complete 128-bit AES key after only 8,000 encryptions. The second attack is asynchronous and allows an attacker to recover parts of an AES key when the victim is running concurrently on the same machine. The attack was applied to a Hyper-Threading processor. However, it is in principle also applicable to modern multicore CPUs with a shared last level cache.

3 Threat Model and Goals

With the move from private computing hardware toward cloud computing, the dangers of cache-based side channels become more acute. The sharing of hardware resources, especially CPU caches, exposes cloud tenants to both *active* time-driven and trace-driven cache attacks by co-located attackers. Neither of these attack types is typically a concern in a private computing environment which does not admit arbitrary code of unknown origin.

In contrast, *passive* time-driven attacks do not require the adversary to execute code on the victim's machine and thus apply equally to both environments. This class of attacks depends on the design, implementation, and behavior of the victim's algorithms.

The goal of this paper is to reduce the exposure of cloud

systems to cache-based side channels to that of private computing environments. This requires defenses against *active* time-driven and trace-driven attacks.

We aim to design a practical system-level mechanism that provides such defenses. The design should be practical in the sense that it is compatible with existing commodity server hardware. Furthermore, its impact on system performance should be minimal, and it should not require significant changes to tenant software.

4 Design

We have designed the STEALTHMEM system to meet the aforementioned goals. The high-level idea is to provide users with a limited amount of private memory that can be accessed as if caches were not shared with other tenants. We call this abstraction *stealth memory* [13]. Tenants can use stealth memory to store data, such as the S-boxes of block ciphers, that are known to be the target of cache-based side channel attacks.

We describe our design and implementation for virtualized systems that are commonly used in public clouds. However, our design could also be applied to regular operating systems running directly on a physical machine. STEALTHMEM extends a hypervisor, such that each VM can access small amounts of memory whose cache lines are not shared.

Let p be the maximum number of CPU cores that can share a cache. This number depends on the CPU model. However, it is generally a small constant, such as $p = 4$ or $p = 6$. In particular, systems with larger numbers of processors typically consist of independent CPUs without shared caches among them.

The hypervisor selects p pre-image sets arbitrarily and assigns one page (or a few pages) from each set to one of the cores such that any two cores that share a cache are assigned pages from different pre-image sets and such that no page is assigned to more than one core. These pages are the cores' *stealth pages*, and they will be exposed to virtual machines running on the cores. At boot or initialization time, the hypervisor sets up the page tables for each core, such that each stealth page is mapped only to the core to which it was assigned. We will call the p pre-image sets from which the stealth pages were chosen the *collision sets* of the stealth pages.

Figure 3 shows an example of a CPU with four cores sharing an L3 cache. Thus, $p = 4$. STEALTHMEM would pick four pages from four different pre-image sets and set the page tables such that the i -th core has exclusive access to the i -th page.

In the rest of this section, we will refine the design and describe how STEALTHMEM disables the three leakage channels of Section 2.

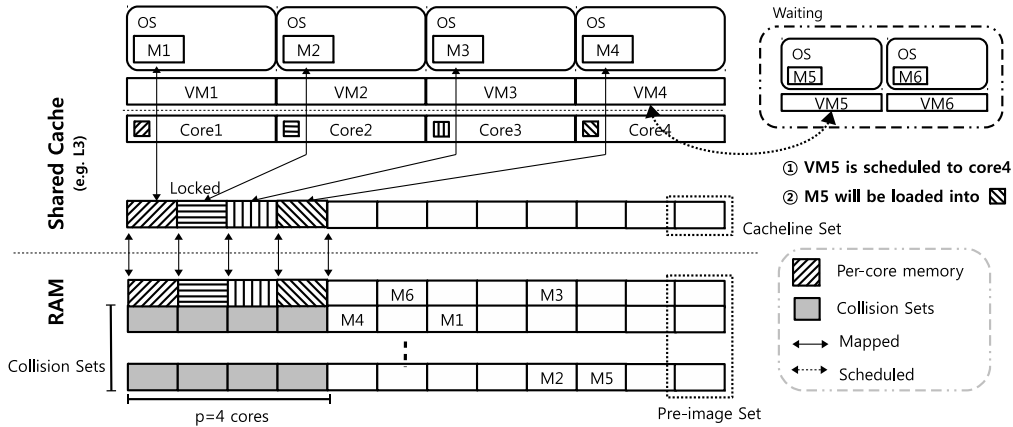


Figure 3: STEALTHMEM on a typical multicore machine: Each VM has its own stealth page. When a VM is scheduled on a core, the core will lock the VM’s stealth page into the shared cache. In one version, the hypervisor will not use the collision sets in order to avoid cache collisions.

4.1 Context Switching

In general, cores are not assigned exclusively to a single VM, but are time-shared among multiple VMs. STEALTHMEM will save and restore stealth pages of VMs during context switches. In the notation of Figure 3, when VM5 is scheduled to a core currently executing VM4, the STEALTHMEM hypervisor will save the stealth pages of the core into VM4’s context, and restore them from VM5’s context. STEALTHMEM will thus ensure that all of VM4’s stealth pages are removed from the cache and all of VM5’s stealth pages are loaded into the cache. STEALTHMEM performs this step at the very end of the context switch—right before control is transferred from VM4 to VM5. This way, all of VM5’s stealth pages will be in the L1 cache (in addition to being in L2 and L3) when VM5 starts executing.

Guest operating systems can use the same technique to multiplex their stealth memory to an arbitrary number of applications.

4.2 Hyper-Threading

In order to avoid asynchronous cache side channels between hyperthreads on the same CPU core, STEALTHMEM gang schedules them. In other words, the hyperthreads of a core are never simultaneously assigned to different VMs. Some widely used hypervisors such as Hyper-V already implement this policy. Given the tight coupling of hyperthreads through shared CPU components, it is hard to envision how the hyperthreads of a core could be simultaneously assigned to multiple VMs without giving rise to a multitude of side channels. Another option is to disable Hyper-Threading.

4.3 Multicore

STEALTHMEM has to prevent an attacker running on one core from using the shared cache to gain information about the stealth memory accesses of a victim running concurrently on another core. For this purpose, STEALTHMEM has to remove or tightly control access to any page that maps to the same cache lines as the stealth pages; i.e., to the p pre-image sets from which the stealth pages were originally chosen. We consider two options: a) STEALTHMEM makes these pages inaccessible and b) STEALTHMEM makes the pages available to VMs, but mediates access to them carefully.

Under the first option, STEALTHMEM ensures at the hypervisor level that, beyond the stealth pages, no pages from the p pre-image sets from which the stealth pages were taken are mapped in the hardware page tables. Thus, these pages are not used and are physically inaccessible to any VM. There is no accessible page in the system that maps to the same cache lines as the stealth pages. Code running on one core cannot probe or manipulate the cache lines of another core’s stealth page because it cannot access any page that maps to the same cache lines.

The total amount of memory that is sacrificed in this way depends on the shared cache configuration of the processor. It is about 3% for all CPU models we have examined. For example, the Xeon W3520 of Table 2 has an 8 MB 16-way set associative L3 cache that is shared among 4 cores ($p = 4$). Dividing 8 MB by the wayness (16) and the page size (4096 bytes), yields 128 page-granular pre-image sets. Removing $p = 4$ of them corresponds to a memory overhead of $4/128 = 3.125\%$. The available shared cache is reduced by the same amount.

One could consider the option of reducing the overhead by letting trusted system software (e.g. the hypervisor, or root partition) use the reserved pages, rather than not

assigning them to guest VMs. However, this would make it hard to argue about the security of the resulting system. For example, if the pages were used to store system code, one would have to ensure that attackers could not access the cache lines of stealth pages indirectly by causing the execution of certain system functions.

4.4 Page Table Alerts

The second option is to use the memory from the p pre-image sets, but to carefully mediate access to them. This option eliminates the memory and cache overhead at the expense of maintenance cost.

STEALTHMEM maintains the invariant that the stealth pages never leave the shared cache. The shared cache is w -way set associative. Intuitively, STEALTHMEM tries to reserve one of the w slots for the stealth cache line, while the remaining $w - 1$ slots can be used by other pages. STEALTHMEM interposes itself on accesses that might cause stealth cache lines to be evicted by setting up the hardware page mappings for most of the colliding pages, such that attempts to access them result in page faults and, thus, invocation of the hypervisor. We call this mechanism a page table alert (PTA).

Rather than simply not using the pre-image sets, the hypervisor maps all their pages to VMs like regular pages. However, the hypervisor sets up PTAs in the hardware page mappings for most of these pages.

More precisely, the hypervisor ensures that there will never be more than $w - 1$ pages (other than one stealth page) from any of the p pre-image sets without a PTA. The $w - 1$ pages without PTAs are effectively a cache of pages that can be accessed directly without incurring the overhead of a PTA.

At initialization, the hypervisor places a PTA on every page of each of the p pre-image sets. Upon a page fault, the handler in the hypervisor will determine if the page fault was caused by a PTA. If so, it will determine the pre-image set of the page that triggered the page fault and perform the following steps: (a) If the pre-image set already contains $w - 1$ pages without a PTA then one of these pages is chosen (according to some replacement strategy), and a PTA is placed on it. (b) The hypervisor ensures that all cache lines of the stealth page and of the up to $w - 1$ colliding pages without PTAs are in the cache. This can be done by accessing these cache lines—possibly repeatedly. On most modern processors, the hypervisor can verify that the lines are indeed in the cache by querying the CPU performance counters for the number of L3 cache misses that occurred while accessing the w pages. If this number is zero then all required lines are in the cache. (c) The hypervisor removes the PTA from the page that caused the page fault. (d) The hypervisor resumes execution of the virtual processor that caused

the page fault. The hypervisor executes steps (b) and (c) atomically—preemption is disabled.

The critical property of these steps is that all accesses to the w pages without PTAs will always hit the cache and, by the inertia property, not cause any cache evictions. Any accesses to other pages from the same pre-image set are guarded by PTAs and will be mediated by STEALTHMEM.

In order to improve scalability, we maintain a separate set of PTAs for each group of p processors that share the cache. Steps (a) to (d) are performed only locally for the set of PTAs of the processor group that contains the processor on which the page fault occurred. Thus, only the local group of p processors needs to be involved in the TLB shutdown, and different processor groups can have different sets of pages on which the PTAs are disabled. This comes at the expense of additional memory for page tables.

k -LRU If the CPU's cache replacement algorithm has the k -LRU property (see Section 2) for some $k > 1$, the following simplification is possible in step (b). Rather than loading the cache lines from all pages without PTAs from the pre-image set, STEALTHMEM only needs to access once each cache line of the stealth page. This reduces the overhead per PTA.

Furthermore, the maximum number of pages without PTAs must now be set to $k - 1$, which may be smaller than $w - 1$. This may lead to more PTAs in this variant of the algorithm.

The critical property of this variant of the algorithm is that, at any time, the only pages in the stealth page's pre-image set that could have been accessed more recently than the stealth page are the $k - 1$ pages without PTAs. Thus, by the k -LRU property, the stealth page will never be evicted from the cache. Figure 4 illustrates this for $k = 4$.

4.5 Optimizations

Our design to expose stealth pages to arbitrary numbers of VMs adds work to context switches. Early experiments showed that this overhead can be significant. We use the following optimizations to minimize this cost.

We associate physical stealth pages with cores, rather than VMs, in order to minimize the need for shared data structures and the resulting lock contention. STEALTHMEM virtualizes these physical stealth pages and exposes a (virtual) stealth page associated with each virtual processor of a guest. This requires copying the contents of a virtual processor's stealth page and acquiring inter-processor locks whenever the hypervisor's scheduler decides to move a virtual processor to a different core. This event, however, is relatively rare and costly in itself. Thus, the work we add is only a small fraction of the total cost.

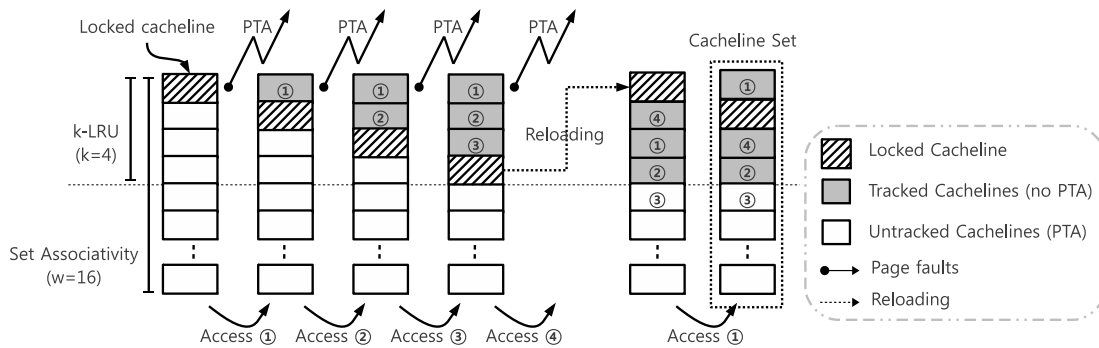


Figure 4: Page table alerts on accessing pages 1, 2, 3, 4 and 1, which are the pre-images of the same cache line set. When getting a page fault on accessing page 4, STEALTHMEM_{PTA} reloads the stealth page to lock its cache lines. The k -LRU policy ($k = 4$) guarantees that the stealth page will not be evicted from the cache. Extra page faults come from accessing PTA-guarded pages. Accessing the tracked cache lines (pages without PTAs) will not generate extra page faults and, thus, no extra performance penalty.

With this optimization, each guest still has its own private stealth pages (one per virtual processor). A potential difficulty of this approach is that guest code sees different stealth pages, depending on which virtual processor it runs on. However, this problem is immaterial for the standard application of STEALTHMEM, in which the stealth pages store S-box tables that never change.

Furthermore, we use several optimizations to minimize the cost of copying stealth pages and flushing their cache lines during context switches. Rather than backing the contents of a core's stealth page to a regular VM context page, we give each VM a separate set of stealth pages. Each VM has its own stealth page from pre-image set i for core i . Thus, if a VM is preempted and later resumes execution on the same set of cores, it is only necessary to refresh the cache lines of its stealth pages. The contents of a stealth page only have to be saved and restored if a virtual processor moves to a different core.

A frequent special case are transitions between a VM and the root partition. When a VM requires a service, such as access to the disk or the network, the root partition needs to be invoked. After the requested service is complete, control is returned to the VM—typically on the same cores on which it was originally running. Thus, it is not necessary to copy the stealth page contents on either transition. Furthermore, since we do not assign stealth pages to the root partition, it is not even necessary to flush caches.

4.6 Extensions

As long as the machine has sufficient memory, we do not use the pages from the collision sets. This will help STEALTHMEM to avoid the performance overhead of maintaining PTAs. If, at some point, the machine is short of memory, STEALTHMEM can start assigning PTA-guarded pages to VMs, making all memory accessible.

STEALTHMEM can, in principle, provide more than one page of stealth memory per core. In order to ensure that stealth pages are not evicted from the cache, the number of stealth pages per core can be at most $k - 1$ for variants that rely on the k -LRU property and at most $w - 1$ for other variants, where w is the wayness of the cache.

4.7 API

VM level STEALTHMEM exposes stealth pages as architectural features of virtual processors. The guest operating system can find out the physical address of a virtual processor's stealth page by making a hypercall, which is a common interface to communicate with the hypervisor.

Application level Application code has to be modified in order to place critical data on stealth pages. STEALTHMEM provides programmers with two simple APIs for requesting and releasing stealth memory as shown in Table 3: *sm_alloc()* and *sm_free()*. Programmers can protect important data structures, such as the S-boxes of encryption algorithms, by requesting stealth memory and then copying the S-boxes to the allocated space. In Section 6, we will evaluate the API design by modifying popular cryptographic algorithms, such as DES, AES and Blowfish, in order to protect their S-boxes with STEALTHMEM.

5 Implementation

We have implemented the STEALTHMEM design on Windows Server 2008 R2 using Hyper-V for virtualization. The STEALTHMEM implementation consists of 5,000 lines of C code that we added to the Hyper-V hypervisor. We also added 500 lines of C code to the Windows boot loader modules (bootmgr and winloader).

API	Description
<code>void *sm_alloc(size_t size)</code>	Allocate dynamic memory of size bytes and return a corresponding pointer
<code>void sm_free(void *ptr)</code>	Free allocated memory pointed to by the given pointer, <i>ptr</i>

Table 3: APIs to allocate and free stealth memory

STEALTHMEM exposes stealth pages to applications through a driver that runs in the VMs and that produces the user mode mappings necessary for `sm_alloc()` and `sm_free()`. We did not have to modify the guest operating system to use STEALTHMEM.

We implemented two versions of STEALTHMEM. In the first implementation, Hyper-V makes the unused pages from the p pre-image sets inaccessible. We will refer to this implementation as STEALTHMEM. The second implementation maps those pages to VMs, but guards them with PTAs. We will explicitly call this version STEALTHMEM_{PTA}.

Hyper-V configures the hardware virtualization extensions to trap into the hypervisor when VM code executes `invd` instructions. We extended the handler to reload the stealth cache lines immediately after executing `invd`. We proceeded similarly with `wbinvd`.

5.1 Root Partition Isolation

Hyper-V relies on Windows to boot the machine. First, Windows boots on the physical machine. Hyper-V is launched only after that. The Windows instance that booted the machine becomes the root partition (equivalent to dom0 in Xen). In general, by the time Hyper-V is launched, the root partition will be using physical pages from all pre-image sets. It would be hard or impossible to free up complete pre-image sets by evicting the root partition from selected physical pages. The reasons include the use of large pages which span all pre-image sets or the use of pages by hardware devices that operate on physical addresses.

We obtain pre-image sets that are not used by the system by marking all pages in these sets as *bad pages* in the boot configuration data using `bcdedit`. This causes the system to ignore these pages and cuts physical memory into many small chunks. We had to adapt the Windows boot loader to enable Windows to boot under this unusual memory configuration.

As a result of this change there are no contiguous large (2 MB or 4 MB) pages on the machine. Both the Windows kernel and Hyper-V attempt to use large pages to improve performance. Large page mappings reduce the translation depth from virtual to physical addresses. Furthermore, they reduce pressure on the TLB. We will evaluate the impact of not using large pages on the performance of STEALTHMEM in Section 6).

5.2 k -LRU

Major CPU vendors implement pseudo-LRU replacement policies as an approximation of the LRU policy [14]. However, this is neither officially documented nor explicitly stated in CPU developer manuals [6, 16]. We conducted the following experiment to find a k value for which our target Xeon W3520 CPU has the k -LRU property.

We selected a set of pages that mapped to the same cache lines. Then, we loaded one page into the L3 cache by reading the contents of the page. After that, we loaded k' other pages of the same pre-image set. Then, we turned on the performance counter and checked L3 cache misses after reading the first page again. We ran this experiment in a device driver (`ring0`) on one core, while the other cores were spinning on a shared lock. Interrupts were disabled. We varied k' from 1 to 16 (set associativity). We started seeing L3 misses at $k' = 15$ and concluded that our CPU has the 14-LRU property.

6 Evaluation

We ask three questions to evaluate STEALTHMEM. First, how effective is STEALTHMEM against cache-based side channel attacks? Second, what is the performance overhead of STEALTHMEM and its characteristics? And finally, how easy is it to adopt STEALTHMEM in existing applications?

6.1 Security

6.1.1 Basic Algorithm

We consider the basic algorithm (without the optimizations of Section 4.5) first. STEALTHMEM guarantees that all cache lines of stealth pages are always in the shared (L3) cache. In the version that makes colliding pages inaccessible, this is the case simply because on each group of cores that share a cache, the only accessible pages from the collision sets of the stealth pages are the stealth pages themselves. We load all stealth pages into the shared cache at initialization. Since Section 4.6 limits the number of stealth pages per collision set to $w - 1$, this will result in all stealth pages being in the cache simultaneously. It is impossible to generate collisions. Thus, by the inertia property, these cache lines will never be evicted.

In the PTA version, it is theoretically possible for stealth cache lines to be evicted very briefly from the

cache during PTA handling while the $w - 1$ colliding pages without PTAs are loaded into the cache. The stealth cache line would be reloaded immediately as part of the same operation, and the time outside the shared cache could be limited to one instruction by accessing the stealth cache line immediately after accessing a colliding line.

Leakage channels This property together with other properties of STEALTHMEM prevents trace-driven and active time-driven attacks on stealth pages. We consider each of the three leakage channels in turn:

Multicore: Attackers running concurrently on other cores cannot directly manipulate (prime) or probe stealth cache lines of the victim's core. This holds for the shared cache because, as observed above, all stealth lines always remain in the shared (L3) cache irrespective of the actions of victims or attackers. It also holds for the other caches (L1 and L2) because they are not shared.

Time sharing: Attackers who time-share a core with a victim cannot directly manipulate or probe stealth cache lines either because we load all stealth cache lines into the cache (including L1 and L2) at the very end of a context switch. Thus, no matter what the adversary or the victim did before the context switch, all stealth lines will be in all caches after a context switch. Thus, direct priming and probing the cache should yield no information.

Hyper-Threading: STEALTHMEM gang schedules hyperthreads to prevent side channels across them.

Limitations While STEALTHMEM locks stealth lines into the last level shared (L3) cache, it has no such control over the upper level caches (L1 and L2) other than reloading stealth pages while context switching. Accordingly, STEALTHMEM cannot hide the timing differences coming out of L1 and L2 cache. Passive timing attacks may arise by exploiting the timing differences between L1 and L3 from a different VM. As stated earlier, passive timing attacks are not our focus since they are not a new threat that results from hardware sharing in the cloud.

6.1.2 Extensions and Optimizations

Per-VM stealth pages Section 4.5 describes an optimization that maintains a separate set of per-core stealth pages for each VM. With this optimization, stealth cache lines are not guaranteed to stay in the shared cache permanently. However, by loading the stealth page contents into the cache at the end of context switches, STEALTHMEM guarantees that the contents of a VM's per-core stealth pages are reloaded in the shared cache, *whenever the core executes the VM*. Thus, the situation for attackers running concurrently on different cores is the same as for the basic algorithm. Our observations regarding context switches and Hyper-Threading also carry over directly.

k -LRU In the PTA variant that relies on the k -LRU property, the stealth page is kept in the cache because at most $k - 1$ colliding pages can be accessed without PTAs. Since STEALTHMEM accesses the stealth page at the end of every page fault that results in a PTA update, the stealth cache lines are always at least the k -least recently used lines in their associative set. Thus, on a CPU with the k -LRU property, they will not be evicted.

6.1.3 Denial of Service

VMs do not have to (and cannot) request or release stealth pages. Instead, STEALTHMEM provides every VM with its own set of stealth pages as part of the virtual machine interface. This set is fixed from the point of view of the VM. Accesses by a VM to its stealth pages do not affect other VMs. Thus, there should be no denial of service attacks involving stealth pages at the VM interface level.

Guest operating systems running inside VMs may have to provide stealth pages to multiple processes. The details of this lie outside the scope of this paper. As noted above, the techniques used in STEALTHMEM can also be applied to operating systems. Operating systems that choose to follow the STEALTHMEM approach virtualize their VM-level stealth pages and provide a fixed independent set of stealth pages to each process. Again, this type of stealth memory should not give rise to denial of service attacks. The APIs of Table 3 would be merely convenient syntax for a process to obtain a pointer to its stealth pages.

6.2 Performance

We have measured the performance of our STEALTHMEM implementation to assess the efficiency and practicality of STEALTHMEM. The experiments ran on an HP Z400 workstation with a 2.67 GHz 4 core Intel Xeon W3520 CPU with 16 GB of DDR3 RAM. The cores were running at 2.8 GHz. Each CPU core has a 32 kB 8-way L1 D-cache, a 32 kB 4-way L1 I-cache and a 256 kB 8-way L2 cache. In addition, the four cores share an 8 MB 16-way L3 cache. The machine ran a 64-bit version of Windows Server 2008 R2 HPC Edition (no service pack). We configured the power settings to run the CPU always at full speed in order to reduce measurement noise. The virtual machines used in the experiments ran the 64-bit version of Windows 7 Enterprise Edition and had 2 GB of RAM. This was the recommended minimum amount of memory for running the SPEC 2006 CPU benchmark [37].

6.2.1 Performance Overhead

Our first goal was to estimate the overhead of STEALTHMEM and STEALTHMEM_{PTA}. We have measured execution times for three configurations: Baseline—an un-

Benchmark	Baseline		Stealth			Stealth PTA			BaselineNLP		
	time	st.dev.	time	st.dev.	overhead	time	st.dev.	overhead	time	st.dev.	overhead
perlbench	508	0.1%	537	0.3%	5.7%	538	0.5%	5.9%	532	0.5%	4.7%
bzip2	610	2.0%	618	0.2%	1.3%	624	1.8%	2.3%	617	2.0%	1.1%
gcc	430	0.1%	466	0.3%	8.4%	476	0.2%	10.7%	462	0.3%	7.4%
milc	257	0.1%	289	0.7%	12.5%	298	0.5%	16.0%	284	1.6%	10.5%
namd	498	0.0%	500	0.1%	0.4%	500	0.1%	0.4%	499	0.1%	0.2%
dealII	478	0.1%	492	0.3%	2.9%	495	0.2%	3.6%	490	0.1%	2.5%
soplex	361	1.9%	401	0.4%	11.1%	412	0.3%	14.1%	394	0.2%	9.1%
povray	228	0.1%	229	0.6%	0.4%	229	0.1%	0.4%	228	0.2%	0.0%
calculix	360	0.2%	366	0.3%	1.7%	366	0.3%	1.7%	363	0.8%	0.8%
astar	454	0.1%	501	0.3%	10.4%	508	1.3%	11.9%	495	0.2%	9.0%
wrf	307	1.9%	331	0.8%	7.8%	336	1.2%	9.4%	329	0.6%	7.2%
sphinx3	602	0.1%	654	0.4%	8.6%	662	0.7%	10.0%	639	0.2%	6.1%
xalancbmk	307	0.2%	324	0.2%	5.5%	329	0.3%	7.2%	321	0.0%	4.6%
average					5.9%			7.2%			4.9%

Table 4: Running time in seconds (time), error bound (st.dev.) and overhead on 13 SPEC2006 CPU benchmarks for Baseline, STEALTHMEM, STEALTHMEM_{PTA} and BaselineNLP.

modified version of Windows with an unmodified version of Hyper-V—and our respective implementations of STEALTHMEM and STEALTHMEM_{PTA}.

In the first experiment, we ran each configuration with two VMs. One VM ran the SPEC 2006 CPU benchmark [37]. Another VM was idle. Table 4 displays the execution times for 13 applications from the SPEC benchmark suite. We repeated each run ten times, obtaining ten samples for each time measurement. The running times in the table are the sample medians. The table also displays the sample standard deviation as a percentage of the sample average as an indication of the noise in the sample. The sample standard deviation is typically less than one percent of the sample average.

The overhead of STEALTHMEM varies between close to zero for about one third of the SPEC applications and 12.5% for *milc*. The average overhead is 5.9%. As expected, the overhead of STEALTHMEM_{PTA} (7.2%) is larger than that of STEALTHMEM because of the extra cost of handling PTA page faults. Server operators can choose either variant, depending on the memory usage of their servers.

We also attempted to find the source of the overhead of STEALTHMEM. Possible sources are the cost of virtualizing stealth pages, the 3% reduction in the size of the available cache and the cost of not being able to use large pages. We repeated the experiment with a configuration that is identical to the Baseline configuration, except that it does not use large pages. It is labeled BaselineNLP (for ‘no large pages’) in Table 4. The overheads for BaselineNLP across the different SPEC applications correlate with the overheads of STEALTHMEM. The overhead due to not using large pages (4.9% on average) accounts for more than 80% of the overhead of STEALTHMEM.

We constructed BaselineNLP using the same binaries as Baseline. However, at hypervisor startup, we disabled

one Hyper-V function by using the debugger to overwrite its first instruction with a *ret*. This function is responsible for replacing regular mappings by large mappings in the extended page tables. Without it, Hyper-V will not use large page mappings irrespective of the actions of the root partition or other guests.

6.2.2 Comparison with Page Coloring

Page coloring [33] isolates VMs from cache-related dependencies by partitioning physical memory pages among VMs such that no VM shares cache lines with any other VM. We modified one of the Hyper-V support drivers in the root partition (*vid.sys*) to assign physical memory to VMs accordingly.

In this simple implementation of Page Coloring, the VMs still share cache lines with the root partition. The same holds for the system in [33]. In contrast, our STEALTHMEM implementation isolates stealth pages also from the root partition. While this difference makes the Page Coloring configuration less secure, it should work to its advantage in the performance comparison.

The next experiment compares the overheads of STEALTHMEM and Page Coloring as the number of VMs increases. We ran BaselineNLP, STEALTHMEM and Page Coloring with between 2 and 7 VMs, running the SPEC workload in one VM and leaving the remaining VMs idle. The root partition is not included in the VM count. Again, each time measurement is the median of ten SPEC runs. The sample standard deviation was typically less than 1% and in no case more than 2.5% of the sample mean.

Figure 5 displays the overheads over BaselineNLP of STEALTHMEM (left) and Page Coloring (right) as a function of the number of VMs. We chose to display the overhead over BaselineNLP, rather than Baseline, in order to eliminate the constant cost of not using large pages,

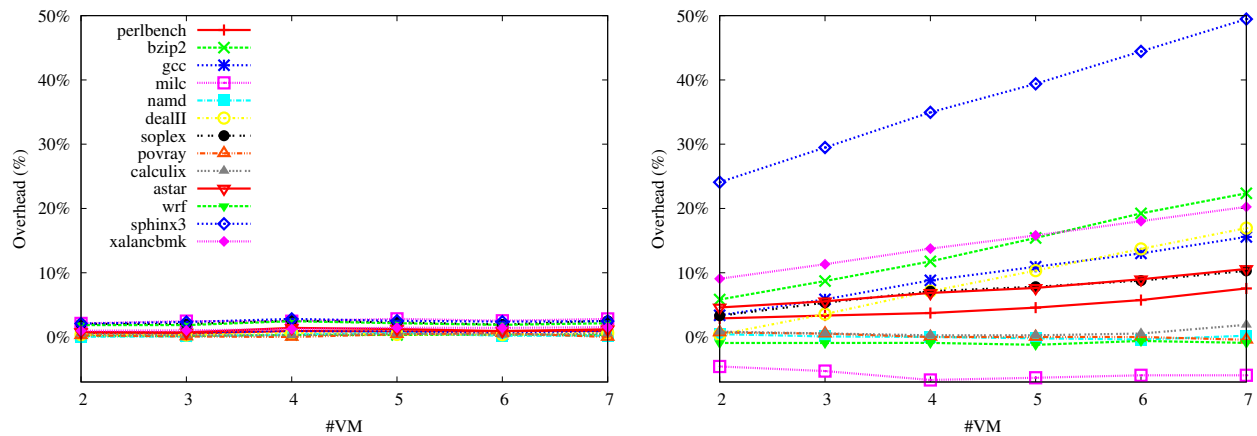


Figure 5: Overhead of STEALTHMEM (left) and Page Coloring (right) over BaselineNLP. The x -axis is the number of VMs.

which affects STEALTHMEM and Page Coloring similarly. Using Baseline adds an application dependent constant to each curve.

Overall, the overhead of STEALTHMEM is significantly smaller than the overhead of Page Coloring. The latter grows with the number of VMs, as each VM gets a smaller fraction of the cache. In contrast, the overhead of STEALTHMEM remains largely constant as the number of VMs increases.

Figure 5 also shows significant differences between the individual benchmarks. For eight benchmarks, Page Coloring shows a large and rising overhead. The most extreme case of this is *sphinx3* with a maximum overhead of almost 50%. For four benchmarks, the overhead of Page Coloring is close to zero. Finally, the *milc* benchmark stands out, as Page Coloring runs it consistently faster than BaselineNLP and STEALTHMEM.

These observations are roughly consistent with the cache sensitivity analysis of Jaleel [19]. The applications with low overhead (*namd*, *povray* and *calculix*) appear to have very small working sets that fit into the L3 cache of all configurations we used in the experiment (including Page Coloring with 7 VMs). For the eight benchmarks with higher overhead, the number of cache misses appears to be sensitive to lower cache sizes in the range covered by our Page Coloring experiment (8/7 MB to 8 MB). For the *milc* application, the data reported by Jaleel indicate a working set size of more than 64 MB. This suggests that *milc* may be thrashing the L3 cache as well as the TLB even when given the entire cache of the machine under BaselineNLP. The performance improvement under Page Coloring may be the result of the CPU being able to resolve certain events (such as page table walks) faster when a large part of the cache is not being thrashed by *milc*.

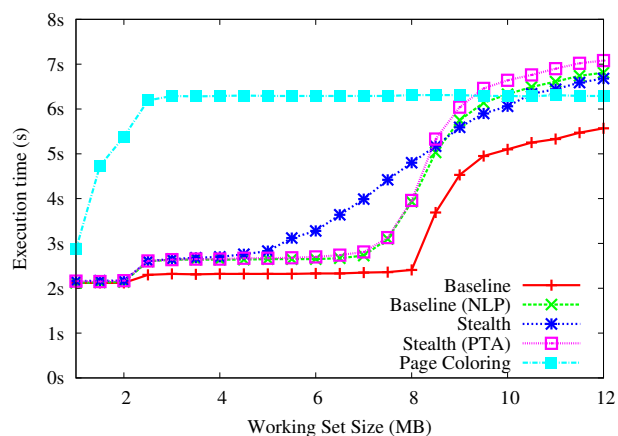


Figure 6: Running times of a micro-benchmark as a function of its working set size.

6.2.3 Overhead With Various Working Set Sizes

The following experiment shows overhead as a function of working set size. Given the working set of an application, developers can estimate the expected performance overhead when they modify an application to use STEALTHMEM.

In the experiment, we used a synthetic application that makes a large number of accesses to an array whose size we varied (the working set size). The working set size is the input to the application. It allocates an array of that size and reads memory from the array in a tight loop. The memory accesses start at offset zero and move up the array in a quasi-linear pattern of increasing the offset for the next read operation by 192 bytes (three cache line sizes) and reducing the following offset by 64 bytes (one cache line size). This is followed by another 192 byte increase and another 64 byte reduction etc. When the end of the array is reached, the process is repeated, starting

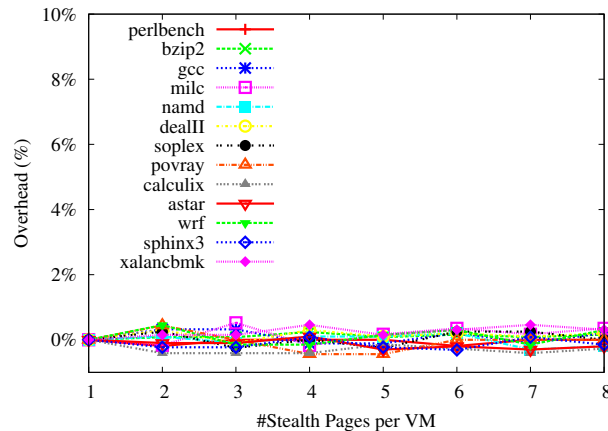


Figure 7: Overhead of STEALTHMEM as a function of the number of stealth pages

again at offset zero.

We ran the application for several configurations. In each case, we ran seven VMs. One VM was running our application. The remaining six VMs were idle. We varied the working set sizes from 100 kB to 12.5 MB and measured for each run the time needed by the application to make three billion memory accesses. The results are displayed in Figure 6. The time measurements in the figure are the medians over five runs. The sample standard deviations were less than 0.5% of the sample means for most working set sizes. However, where the slope of a curve was very steep, the sample standard deviations could be up to 5% of the sample means.

Most configurations show a sharp rise in the running times as the working set size increases past the size of the L3 cache (8 MB). For Page Coloring, this jump occurs for much smaller working sets since the VM can access only one seventh of the CPU’s cache. Most configurations also display a second, smaller increase around 2 MB. This may be the result of TLB misses. The processor’s L2 TLB has 512 entries which can address up to 2 MB based on regular 4 kB page mappings.

For very large workload sizes, BaselineNLP and STEALTHMEM become slower than Page Coloring. This appears to be the same phenomenon that caused Page Coloring to outperform BaselineNLP and STEALTHMEM on the *milc* benchmark.

6.2.4 Overhead With Various Stealth Pages

This experiment attempts to estimate how the overhead of STEALTHMEM depends on the number of stealth pages that the hypervisor provides to each VM. We ran STEALTHMEM with one VM running the SPEC benchmarks and varied the number of stealth pages per VM. As before, the times we report are the medians over ten runs.

The sample standard deviations were less than 0.4% of the sample means in all cases.

Figure 7 displays the overhead with respect to STEALTHMEM with one stealth page per VM. There is no noticeable increase in the running time as the number of stealth pages increases. This is the result of the optimizations described earlier that eliminate the need to copy the contents of stealth pages or to load them into the cache frequently.

6.3 Block Ciphers

The goal of this experiment is to evaluate performance for real-world applications that heavily use stealth pages. We choose three popular block ciphers: AES [2], DES [1] and Blowfish [35]. Efficient implementations of each of these ciphers perform a number of lookups in a table during encryption and decryption. We picked Bruce Schneier’s implementation of Blowfish [36], and standard commercial implementations of AES and DES and adapted them to use stealth pages (as described in Section 6.4).

We measured the encryption speeds of each of the ciphers for (a) the baseline configuration (unmodified Windows 7, Hyper-V and cipher implementation), (b) our STEALTHMEM configuration using the modified versions of the cipher implementations just described and (c) an uncached configuration, which places the S-box tables on a page that is not cached. Configuration (c) runs the modified version of the block cipher implementations on an unmodified version of Windows and an essentially unmodified version of the hypervisor. We added a driver in the Windows 7 guest that creates an uncached user mode mapping to a page. We also had to add one hypercall to Hyper-V to ensure that this page was indeed mapped as uncached in the physical page tables. We included this configuration in our experiments since using an uncached page is the simplest way to eliminate cache side channels.

We measured the time required to encrypt 5 million bytes for each configuration. In order to reduce measurement noise, we raised the scheduling priority of the encryption process to the `HIGH_PRIORITY_CLASS` of the Windows scheduler. We ran the experiment in a small buffer configuration (50,000 byte buffer encrypted 1,000 times) and a large buffer configuration (5 million byte buffer encrypted once) to show performance overheads with different workloads.

The numbers in Table 5 are averaged over 1,000 runs. The sample standard deviation lies between 1 and 4 percent of the sample averages. The overhead of using a stealth page with respect to baseline performance lies between 2% and 5%, while the overhead of the uncached version lies between 97.9% and 99.9%.

Cipher	A small buffer (50,000 bytes)					A large buffer (5,000,000 bytes)				
	Baseline	Stealth	Uncached	Baseline	Stealth	Uncached	Baseline	Stealth	Uncached	
DES	60	58	-3%	0.83	-99%	59	57	-3%	0.83	-99%
AES	150	143	-5%	1.33	-99%	142	135	-5%	1.32	-99%
Blowfish	77	75	-2%	1.65	-98%	75	74	-2%	1.64	-98%

Table 5: Block cipher encryption speeds in MB/s for small and large buffers. We mapped the S-box of each encryption algorithm to cached, stealth and uncached pages.

	Source code
Original	<code>static unsigned long S[4][256];</code>
Modified	<code>typedef unsigned long UlongArray[256]; static UlongArray *S; // in the initialization function S = sm_alloc(4*256);</code>

Table 6: Modified Blowfish to use STEALTHMEM

Encryption	Size of S-box	LoC Changes
DES	256 B * 8 = 2 kB	5 lines
AES	1024 B * 4 = 4 kB	34 lines
Blowfish	1024 B * 4 = 4 kB	3 lines

Table 7: Size of S-box in various encryption algorithms, and corresponding changes to use STEALTHMEM

6.4 Ease-of-use

We had to make only minor changes to the block cipher implementations to adapt them to STEALTHMEM. These changes amounted to replacing the global array variables that hold the encryption tables by pointers to the stealth page. In the case of Blowfish, this change required only 3 lines. We replaced the global array declaration by a pointer and assigned the base of the stealth page to it in the initialization function (see Table 6).

Adapting DES required us to change a total of 5 lines. In addition to a change of the form just described, we had to copy the table contents (constants in the source code) to the stealth page. This was not necessary for Blowfish which read these data from a file. Adapting AES required a total of 34 lines. This large number is the result of the fact that our AES implementation declares its table as 8 different variables, which forced us to repeat 8 times the simple adaptation we did for DES. Table 7 summarizes the S-box layouts and the required code changes for the three ciphers.

7 Related Work

Kocher [22] presented the initial idea of exploiting timing differences to break popular cryptosystems. Even though Kocher speculated about the possibility of exploiting cache side channels, the first theoretical model of cache attacks was described by Page [31] in 2002.

Around that time, researchers started investigating cache-based side channels against actual cryptosystems and broke popular cryptosystems such as AES [4, 8, 9, 30], and DES [40]. With the emergence of simultaneous multi-threading, researchers discovered a new type of cache attacks, classified as trace-driven attacks in our paper, against AES [3, 30] and RSA [32] by exploiting the new architectural feature of an L1 cache that is shared by two hyperthreads. Recently, Osvik et al. [30, 39] executed more quantitative research on cache attacks and classified possible attack methods. The new cloud computing environments have also gained the attention of researchers who have explored the possibility of cache-based side channel attacks in the cloud [7, 34, 44], or inversely their use in verifying co-residency of VMs [45].

Mitigation methods against cache attacks have been studied in three directions: suggesting new cache hardware with security in mind, designing software-only defense mechanism, and developing application specific mitigation methods.

Hardware-based mitigation methods focus on reducing or obfuscating cache accesses [23, 24, 41–43] by designing new caches, or partitioning caches with dynamic or other efficient methods [12, 21, 27, 42, 43]. Wang and Lee [42, 43] proposed PLcache to hide cache access patterns by locking cache lines, and RPcache to obfuscate patterns by randomizing cache mappings. These hardware-based approaches, however, will not provide practical defenses until CPU makers integrate them into mainstream CPUs and cloud providers purchase them. Our defense mechanism not only provides similar security guarantee as these methods, but also allows cloud providers to utilize existing commodity hardware.

Software-only defenses [7, 11, 13, 15, 33] also have been actively proposed. Against time-drive attacks, Coppens et al. [11] demonstrated a mitigation method by modifying a compiler to remove control-flow dependencies on confidential data, such as secret keys. This compiler technique, however, leaves applications still vulnerable to trace-driven cache attacks in the cloud. Against trace-driven attacks, static partitioning techniques, such as page coloring [33], provide a general mitigation solution by partitioning pre-image sets among VMs. Since static partitioning divides the cache by the number of VMs, its performance overhead becomes significantly larger when cloud providers run more VMs, as we demonstrated in

Section 6. Our solution, however, assigns unique cache line sets to virtual processors and flexibly loads stealth pages of each VM if necessary, and thus demonstrates better performance.

Erlingsson and Abadi [13] proposed the abstraction of “stealth memory” and sketched techniques for implementing it. We have realized the abstraction in a virtualized multiprocessor environment by designing and implementing a complete defense system against cache side channel attacks and evaluating it across system layers (from the hypervisor to cryptographic applications) in a concrete security model.

Since existing hardware-based and software-only defenses are not practical because they require new CPU hardware or because of their performance overhead, researchers have been exploring mitigation methods for particular algorithms or applications. The design and implementation of AES has been actively revisited by [8–10, 14, 30, 39], focusing on eliminating or controlling access patterns on S-Boxes, or not placing S-Boxes into memory [28], but into registers of x64 CPUs. Recently, Intel [17] introduced a special instruction for AES encryption and decryption. These approaches may secure AES from cache side channels, but it is not realistic to introduce new CPU instructions for every software algorithm that might be subject to leaking information via cache side channels. In contrast, STEALTHMEM provides a general system-level protection solution that every application can take advantage of if it wants to protect its confidential data in the cloud.

8 Conclusion

We design and implement STEALTHMEM, a system-level protection mechanism against cache-based side channel attacks, specifically against active time-driven and trace-driven cache attacks, which cloud platforms suffer from. STEALTHMEM helps cloud service providers offer better security against cache attacks, without requiring any hardware modifications.

With only a few lines of code changes, we can modify popular encryption schemes such as AES, DES and Blowfish to use STEALTHMEM. Running the SPEC 2006 CPU benchmark shows an overhead of 5.9%, and our micro-benchmark shows that the secured AES, DES, and Blowfish have between 2% and 5% performance overhead, while making extensive use of STEALTHMEM.

Acknowledgments

We thank the anonymous reviewers, and our shepherd, David Lie, for their feedback. We would also like to thank Úfar Erlingsson and Martín Abadi for several valuable

conversations. Taesoo Kim is partially supported by the Samsung Scholarship Foundation.

References

- [1] Data Encryption Standard (DES). In *FIPS PUB 46, Federal Information Processing Standards Publication* (1977).
- [2] Advanced Encryption Standard (AES). In *FIPS PUB 197, Federal Information Processing Standards Publication* (2001).
- [3] ACHIÇMEZ, O., AND ÇETIN KAYA KOÇ. Trace-driven cache attacks on AES. *Cryptology ePrint Archive*, Report 2006/138, 2006.
- [4] ACHIÇMEZ, O., SCHINDLER, W., AND ÇETIN K. KOÇ. Cache based remote timing attack on the AES. In *Topics in Cryptology – CT-RSA 2007, The Cryptographers’ Track at the RSA Conference 2007* (2007), Springer-Verlag, pp. 271–286.
- [5] AMAZON, INC. Amazon Elastic Compute Cloud (EC2). <http://aws.amazon.com/ec2>, 2012.
- [6] AMD, INC. *AMD64 Architecture Programmer’s Manual*. No. 24594. December 2011.
- [7] AVIRAM, A., HU, S., FORD, B., AND GUMMADI, R. Determining timing channels in compute clouds. In *Proceedings of the 2010 ACM Cloud Computing Security Workshop* (2010), pp. 103–108.
- [8] BERNSTEIN, D. J. Cache-timing attacks on AES. Available at: <http://cr.yp.to/antiforgery/cachetiming-20050414.pdf>, 2005.
- [9] BONNEAU, J., AND MIRONOV, I. Cache-collision timing attacks against AES. In *Proceedings of the 8th International Workshop on Cryptographic Hardware and Embedded Systems* (2006), pp. 201–215.
- [10] BRICKELL, E., GRAUNKE, G., NEVE, M., AND SEIFERT, J.-P. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *IACR ePrint Archive*, Report 2006/052, 2006.
- [11] COPPENS, B., VERBAUWHEDE, I., BOSSCHERE, K. D., AND SUTTER, B. D. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy* (2009), pp. 45–60.
- [12] DOMNITSER, L., JALEEL, A., LOEW, J., ABU-GHAZALEH, N., AND PONOMAREV, D. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization* 8, 4 (Jan. 2012), 35:1–35:21.
- [13] ERLINGSSON, Ú., AND ABADI, M. Operating system protection against side-channel attacks that exploit memory latency. Tech. Rep. MSR-TR-2007-117, Microsoft Research, August 2007.
- [14] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache Games – bringing access-based cache attacks on AES to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (May 2011), pp. 490–505.
- [15] HU, W. M. Reducing timing channels with fuzzy time. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy* (1991), pp. 8–20.
- [16] INTEL, INC. *Intel® 64 and IA-32 Architectures Software Developer’s Manual*. No. 253669-033US. December 2009.
- [17] INTEL, INC. Advanced Encryption Standard (AES) Instructions Set. <http://software.intel.com/file/24917>, 2010.

- [18] ION, I., SACHDEVA, N., KUMARAGURU, P., AND ČAPKUN, S. Home is safer than the cloud! Privacy concerns for consumer cloud storage. In *Proceedings of the Seventh Symposium on Usable Privacy and Security* (2011), pp. 13:1–13:20.
- [19] JALEEL, A. Memory characterization of workloads using instrumentation-driven simulation – a pin-based memory characterization of the SPEC CPU2000 and SPEC CPU2006 benchmark suites. Tech. rep., VSSAD, 2007.
- [20] JANSEN, W., AND GRANCE, T. Guidelines on security and privacy in public cloud computing. NIST Special Publication 800-144, December 2011.
- [21] KIM, S., CHANDRA, D., AND SOLIHIN, Y. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques* (2004), pp. 111–122.
- [22] KOCHER, P. C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology* (1996), pp. 104–113.
- [23] KONG, J., ACIİÇMEZ, O., SEIFERT, J.-P., AND ZHOU, H. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 2nd ACM Workshop on Computer Security Architectures* (2008), pp. 25–34.
- [24] KONG, J., ACIİÇMEZ, O., SEIFERT, J.-P., AND ZHOU, H. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *Proceedings of the 15th International Conference on High Performance Computer Architecture* (2009), pp. 393–404.
- [25] MANGALINDAN, J. Is user data safe in the cloud? <http://tech.fortune.cnn.com/2010/09/24/is-user-data-safe-in-the-cloud>, September 2010.
- [26] MICROSOFT, INC. Microsoft Azure Services Platform. <http://www.microsoft.com/azure/>.
- [27] MOSCIBRODA, T., AND MUTLU, O. Memory performance attacks: denial of memory service in multi-core systems. In *Proceedings of the 16th USENIX Security Symposium* (2007), pp. 257–274.
- [28] MÜLLER, T., DEWALD, A., AND FREILING, F. C. AESSE: a cold-boot resistant implementation of AES. In *Proceedings of the Third European Workshop on System Security* (2010), pp. 42–47.
- [29] NEVE, M., AND SEIFERT, J.-P. Advances on access-driven cache attacks on AES. In *Selected Areas in Cryptography*, vol. 4356. 2007, pp. 147–162.
- [30] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers Track at the RSA Conference 2006* (2005), pp. 1–20.
- [31] PAGE, D. Theoretical use of cache memory as a cryptanalytic side-channel. Tech. Rep. CSTR-02-003, Department of Computer Science, University of Bristol, June 2002.
- [32] PERCIVAL, C. Cache missing for fun and profit. In *BSDCan 2005* (Ottawa, 2005).
- [33] RAJ, H., NATHUJI, R., SINGH, A., AND ENGLAND, P. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM Cloud Computing Security Workshop* (2009), pp. 77–84.
- [34] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (2009), pp. 199–212.
- [35] SCHNEIER, B. The Blowfish encryption algorithm. <http://www.schneier.com/blowfish.html>.
- [36] SCHNEIER, B. The Blowfish source code. <http://www.schneier.com/blowfish-download.html>.
- [37] (SPEC), S. P. E. C. The SPEC CPU 2006 Benchmark Suite. <http://www.specbench.org>.
- [38] SUZAKI, K., IJIMA, K., YAGI, T., AND ARTHO, C. Memory deduplication as a threat to the guest OS. In *Proceedings of the Fourth European Workshop on System Security (EUROSEC '11)* (2011), pp. 1:1–1:6.
- [39] TROMER, E., OSVIK, D. A., AND SHAMIR, A. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology* 23, 2 (2010), 37–71.
- [40] TSUNOO, Y., SAITO, T., SUZAKI, T., AND SHIGERI, M. Cryptanalysis of DES implemented on computers with cache. In *Proceedings of the 2003 Cryptographic Hardware and Embedded Systems* (2003), pp. 62–76.
- [41] WANG, Z., AND LEE, R. B. Covert and side channels due to processor architecture. In *Proceedings of the 22nd Annual Computer Security Applications Conference* (December 2006), pp. 473–482.
- [42] WANG, Z., AND LEE, R. B. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th International Symposium on Computer Architecture* (2007), pp. 494–505.
- [43] WANG, Z., AND LEE, R. B. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture* (2008), pp. 83–93.
- [44] XU, Y., BAILEY, M., JAHANIAN, F., JOSHI, K., HILTUNEN, M., AND SCHLICHTING, R. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the 2011 ACM Cloud Computing Security Workshop* (2011), pp. 29–40.
- [45] ZHANG, Y., JUELS, A., OPREA, A., AND REITER, M. K. Home-Along: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (2011), pp. 313–328.

Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices

Nadia Heninger^{†*} Zakir Durumeric^{‡*} Eric Wustrow[‡] J. Alex Halderman[‡]

[†] *University of California, San Diego*
nadiah@cs.ucsd.edu

[‡] *The University of Michigan*
{zakir, ewust, jhalderm}@umich.edu

Abstract

RSA and DSA can fail catastrophically when used with malfunctioning random number generators, but the extent to which these problems arise in practice has never been comprehensively studied at Internet scale. We perform the largest ever network survey of TLS and SSH servers and present evidence that vulnerable keys are surprisingly widespread. We find that 0.75% of TLS certificates share keys due to insufficient entropy during key generation, and we suspect that another 1.70% come from the same faulty implementations and may be susceptible to compromise. Even more alarmingly, we are able to obtain RSA private keys for 0.50% of TLS hosts and 0.03% of SSH hosts, because their public keys shared nontrivial common factors due to entropy problems, and DSA private keys for 1.03% of SSH hosts, because of insufficient signature randomness. We cluster and investigate the vulnerable hosts, finding that the vast majority appear to be headless or embedded devices. In experiments with three software components commonly used by these devices, we are able to reproduce the vulnerabilities and identify specific software behaviors that induce them, including a boot-time entropy hole in the Linux random number generator. Finally, we suggest defenses and draw lessons for developers, users, and the security community.

1 Introduction and Roadmap

Randomness is essential for modern cryptography, where security often depends on keys being chosen uniformly at random. Researchers have long studied random number generation, from both practical and theoretical perspectives (e.g., [8, 13, 15, 17, 21, 23]), and a handful of major vulnerabilities (e.g., [5, 19]) have attracted considerable scrutiny to some of the most critical implementations. Given the importance of this problem and the effort and attention spent improving the state of the art, one might

expect that today's widely used operating systems and server software generate random numbers securely. In this paper, we test that proposition empirically by examining the public keys in use on the Internet.

The first component of our study is the most comprehensive Internet-wide survey to date of two of the most important cryptographic protocols, TLS and SSH (Section 3.1). By scanning the public IPv4 address space, we collected 5.8 million unique TLS certificates from 12.8 million hosts and 6.2 million unique SSH host keys from 10.2 million hosts. This is 67% more TLS hosts than the latest released EFF SSL Observatory dataset [18]. Our techniques take less than 24 hours to scan the entire address space for listening hosts and less than 96 hours to retrieve keys from them. The results give us a macroscopic perspective of the universe of keys.

Next, we analyze this dataset to find evidence of several kinds of problems related to inadequate randomness. To our surprise, at least 5.57% of TLS hosts and 9.60% of SSH hosts use the same keys as other hosts in an apparently vulnerable manner (Section 4.1). In the case of TLS, at least 5.23% of hosts use manufacturer default keys that were never changed by the owner, and another 0.34% appear to have generated the same keys as one or more other hosts due to malfunctioning random number generators. Only a handful of the vulnerable TLS certificates are signed by browser-trusted certificate authorities.

Even more alarmingly, we are able to compute the private keys for 64,000 (0.50%) of the TLS hosts and 108,000 (1.06%) of the SSH hosts from our scan data alone by exploiting known weaknesses of RSA and DSA when used with insufficient randomness. In the case of RSA, distinct moduli that share exactly one prime factor will result in public keys that appear distinct but whose private keys are efficiently computable by calculating the greatest common divisor (GCD). We implemented an algorithm that can compute the GCDs of all pairs of 11 million distinct public RSA moduli in less than 2 hours (Section 3.3). Using the resulting factors, we are able to

*The first two authors both made substantial contributions.

obtain the private keys for 0.50% of TLS hosts and 0.03% of SSH hosts (Section 4.2). In the case of DSA, if a DSA key is used to sign two different messages with the same ephemeral key, an attacker can efficiently compute the signer's long-term private key. We find that our SSH scan data contain numerous DSA signatures that used the same ephemeral keys during signing, allowing us to compute the private keys for 1.6% of SSH DSA hosts (Section 4.3).

To understand why these problems are occurring, we manually investigated hundreds of the vulnerable hosts, which were representative of the most commonly repeated keys as well as each of the private keys we obtained (Section 3.2). Nearly all served information identifying them as headless or embedded systems, including routers, server management cards, firewalls, and other network devices. Such devices typically generate keys automatically on first boot, and may have limited entropy sources compared to traditional PCs. Furthermore, when we examined clusters of hosts that shared a key or factor, in nearly all cases these appeared to be linked by a manufacturer or device model. These observations lead us to conclude that the problems are caused by specific defective implementations that generate keys without having collected sufficient entropy. We identified vulnerable devices and software from dozens of manufacturers, including some of the largest names in the technology industry, and worked to notify the responsible parties.

In the final component of our study, we experimentally explore the root causes of these vulnerabilities by investigating several of the most common open-source software components from the population of vulnerable devices (Section 5). Based on the devices we identified, it is clear that no one implementation is solely responsible, but we are able to reproduce the vulnerabilities in plausible software configurations. Every software package we examined relies on `/dev/urandom` to generate cryptographic keys; however, we find that Linux's random number generator (RNG) can exhibit a boot-time entropy hole that causes `urandom` to produce deterministic output under conditions likely to occur in headless and embedded devices. In experiments with OpenSSL and Dropbear SSH, we show how repeated output from the system RNG can lead not only to repeated long-term keys but also to factorable RSA keys and repeated DSA ephemeral keys due to the behavior of application-specific entropy pools.

Given the diversity of the devices and software implementations involved, mitigating these problems will require action by many different parties. We draw lessons and recommendations for developers of operating systems, cryptographic libraries, and applications, and for device manufacturers, certificate authorities, end users, and the security and cryptography communities (Section 7). We have also created an online key-check service to allow users to test whether their keys are vulnerable.

It is natural to wonder whether these results should call into question the security of every RSA or DSA key. Based on our analysis, the margin of safety is slimmer than we might like, but we have no reason to doubt the security of most keys generated interactively by users on traditional PCs. While we took advantage of the details of specific cryptographic algorithms in this paper, we conclude that the blame for these vulnerabilities lies chiefly with the implementations. Ultimately, the results of our study should serve as a wake-up call that secure random number generation continues to be an unsolved problem in important areas of practice.

Online resources For an extended version of this paper, partial source code, and our online key-check service, visit <https://factorable.net>.

2 Background

In this section, we review the RSA and DSA public-key cryptosystems and discuss the known weaknesses of each that we used to compromise private keys. We then discuss how an adversary might exploit compromised keys to attack SSH and TLS in practice.

2.1 RSA review

An RSA [35] public key consists of two integers: an exponent e and a modulus N . The modulus N is the product of two randomly chosen prime numbers p and q . The private key is the decryption exponent

$$d = e^{-1} \bmod (p-1)(q-1).$$

Anyone who knows the factorization of N can efficiently compute the private key for any public key (e, N) using the preceding equation. When p and q are unknown, the most efficient known method to calculate the private key is to factor N into p and q and use the above equation to calculate d [9].

Factorable RSA keys No one has been publicly known to factor a well-generated 1024-bit RSA modulus; the largest known factored modulus is 768 bits, which was announced in December 2009 after a multi-year distributed-computing effort [28]. In contrast, the greatest common divisor (GCD) of two 1024-bit integers can be computed in microseconds. This asymmetry leads to a well-known vulnerability: if an attacker can find two distinct RSA moduli N_1 and N_2 that share a prime factor p but have different second prime factors q_1 and q_2 , then the attacker can easily factor both moduli by computing their GCD, p , and dividing to find q_1 and q_2 . The attacker can then compute both private keys as explained above.

2.2 DSA review

A DSA [32] public key consists of three so-called domain parameters (two prime moduli p and q and a generator g of the subgroup of order $q \bmod p$) and an integer $y = g^x \bmod p$, where x is the private key. The domain parameters may be shared among multiple public keys without compromising security. A DSA signature consists of a pair of integers (r, s) : $r = g^k \bmod p \bmod q$ and $s = (k^{-1}(H(m) + xr)) \bmod q$, where k is a randomly chosen ephemeral private key and $H(m)$ is the hash of the message.

Low-entropy DSA signatures DSA is known to fail catastrophically if the ephemeral key k used in the signing operation is generated with insufficient entropy [4]. (Elliptic curve DSA (ECDSA) is similarly vulnerable. [11]) If k is known for a signature (r, s) , then the private key x can be computed from the signature and public key as follows:

$$x = r^{-1}(ks - H(m)) \bmod q.$$

If a DSA private key is used to sign two different messages with the same k , then an attacker can efficiently compute the value k from the public key and signatures and use the above equation to compute the private key x [29]. If two messages m_1 and m_2 were signed using the same ephemeral key k to obtain signatures (r_1, s_1) and (r_2, s_2) , then this will be immediately clear as r_1 and r_2 will be equal. The ephemeral key k can be computed as:

$$k = (H(m_1) - H(m_2))(s_1 - s_2)^{-1} \bmod q.$$

2.3 Attack scenarios

The weak key vulnerabilities we describe in this paper can be exploited to compromise two of the most important cryptographic transport protocols used on the Internet, TLS and SSH, both of which commonly use RSA or DSA to authenticate servers to clients.

TLS In TLS [16], the server sends its public key in a TLS certificate during the protocol handshake. The key is used either to provide a signature on the handshake (when Diffie-Hellman key exchange is negotiated) or to encrypt session key material chosen by the client (when RSA-encrypted key exchange is negotiated).

If the key exchange is RSA encrypted, a passive eavesdropper with the server's private key can decrypt the message containing the session key material and use it to decrypt the entire session. If the session key is negotiated using Diffie-Hellman key exchange, then a passive attacker will be unable to compromise the session key from just a connection transcript. However, in both cases, an active attacker who can intercept and modify traffic between the client and server can man-in-the-middle the connection in order to decrypt or modify the traffic.

SSH In SSH, host keys allow a server to authenticate itself to a client by providing a signature during the protocol handshake. There are two major versions of the protocol. In SSH-1 [38], the client encrypts session key material using the server's public key. SSH-2 [39] uses a Diffie-Hellman key exchange to establish a session key. The user manually verifies the host key fingerprint the first time she connects to an SSH server. Most clients then store the key locally in a `known_hosts` file and automatically trust it for all subsequent connections.

As in TLS, a passive eavesdropper with a server's private key can decrypt an entire SSH-1 session. However, because SSH-2 uses Diffie-Hellman, it is vulnerable only to an active man-in-the-middle attack. In the SSH user authentication protocol, the user-supplied password is sent in plaintext over the encrypted channel. An attacker who knows a server's private key can use the above attacks to learn a user's password and escalate an attack to the system.

3 Methodology

In this section, we explain how we performed our Internet-wide survey of public keys, how we attributed vulnerable keys to devices, and how we efficiently factored poorly generated RSA keys.

3.1 Internet-wide scanning

We performed our data collection in three phases: discovering IP addresses accepting connections on TCP port 443 (HTTPS) or 22 (SSH); performing a TLS or SSH handshake and storing the presented certificate chain or host key; and parsing the collected certificates and host keys into a relational database. Table 1 summarizes the results.

Host discovery In the first phase, we scanned the public IPv4 address space to find hosts with port 443 or 22 open. We used the Nmap 5 network exploration tool [33]. We executed our first host discovery scan beginning on October 6, 2011 from 25 Amazon EC2 Micro instances spread across five EC2 regions (Virginia, California, Japan, Singapore, and Ireland). The scan ran at an average of 40,566 IPs/second and finished in 25 hours.

Certificate and host-key retrieval For TLS, we implemented a certificate fetcher in Python using the Twisted event-driven network framework. We fetched TLS certificates using an EC2 Large instance with five processes each maintaining 800 concurrent connections. We started fetching certificates on October 11, 2011.

To efficiently collect SSH host keys, we implemented a simple SSH client in C, which is able to process upwards of 1200 hosts/second by concurrently performing

	SSL Observatory (12/2010)	Our TLS scan (10/2011)	Our SSH scans (2-4/2012)
Hosts with open port 443 or 22	≈16,200,000	28,923,800	23,237,081
Completed protocol handshakes	7,704,837	12,828,613	10,216,363
Distinct RSA public keys	3,933,366	5,656,519	3,821,639
Distinct DSA public keys	1,906	6,241	2,789,662
Distinct TLS certificates	4,021,766	5,847,957	—
Trusted by major browsers	1,455,391	1,956,267	—

Table 1: **Internet-wide scan results**— We exhaustively scanned the public IPv4 address space for TLS and SSH servers listening on ports 443 and 22, respectively. Our results constitute the largest such network survey reported to date. For comparison, we also show statistics for the EFF SSL Observatory’s most recent public dataset [18].

protocol handshakes using *libevent*. Initially, we ran the fetcher from an EC2 Large instance in a run that started on February 12, 2012. This run targeted only RSA-based host keys. In two later runs, we targeted DSA-based host keys, and rescanned those hosts that had offered DSA keys in the first SSH scan. For these, we also stored the authentication signature provided by the server; we varied the client string to ensure that each signature would be distinct. The first DSA run started on March 26, 2012 from a host at UCSD. The second run, from a host at the University of Michigan, started on April 1, 2012; it took 3 hours to complete.

TLS certificate processing For TLS, we performed a third processing stage in which we parsed the previously fetched certificate chains and generated a database from the X.509 fields. We implemented a certificate parser in Python and C primarily based on the M2Crypto SWIG interface to the OpenSSL library.

3.2 Identifying vulnerable device models

We attempted to determine what hardware and software generated or served the weak keys we identified using manual detective work. The most straightforward method was based on TLS certificate information—predominately the X.509 *subject* and *issuer* fields. In many cases, the certificate identified a specific manufacturer or device model. Other certificates contained less information; we attempted to identify these devices through Nmap host detection or by inspecting the public contents of HTTPS sites or other IP services hosted on the IP addresses.

When we could identify a pattern in vulnerable TLS certificates that appeared to belong to a device model or product line, we constructed regular expressions to find other similar devices in our scan results. Under the theory that the keys were vulnerable because of a problem with the design of the devices (where they were most likely generated), this allows us to estimate the total population of devices that might be potentially vulnerable, beyond those serving immediately compromised keys.

Identifying SSH devices was more problematic, as SSH keys do not include descriptive fields and the server identification string used in the protocol often indicated only a common build of a popular SSH server. We were able to classify many of the vulnerable SSH hosts using a combination of TCP/IP fingerprinting and examination of information served over HTTP and HTTPS.

The device names and manufacturers that we report here have been identified with moderate or high confidence given the available information. However, because we do not have physical access to the hosts, we cannot state with certainty that all our identifications are correct.

3.3 Efficiently computing all-pairs GCDs

We now describe how we efficiently computed the pairwise GCD of all distinct RSA moduli in our multimillion-key dataset. This allowed us to calculate RSA private keys for 66,540 vulnerable hosts that shared one of their RSA prime factors with another host in our survey.

The fastest known factoring method for general integers is the number field sieve, which has heuristic complexity $O(2^{n^{1/3}(\log n)^{2/3}})$ for n -bit numbers [30]. In contrast to factoring, the greatest common divisor (GCD) of two integers can be computed very efficiently using Euclid’s algorithm. Using fast integer arithmetic, the complexity of GCD can be improved to $O(n(\lg n)^2 \lg \lg n)$ [7]. Computing the GCD of two 1024-bit RSA moduli using the GMP library [20] takes approximately 15 μ s on a current mid-range computer.

The naïve way to compute the GCDs of every pair of integers in a large set would be to apply a GCD algorithm to each pair individually. There are 6×10^{13} distinct pairs of RSA moduli in our data; at 15 μ s per pair, this calculation would take 30 years. We can do much better by using a more efficient algorithm.

To accomplish this, we implemented a quasilinear-time algorithm for factoring a collection of integers into co-primes, based on Bernstein [6]. The relevant steps, illustrated in Figure 1, are as follows:

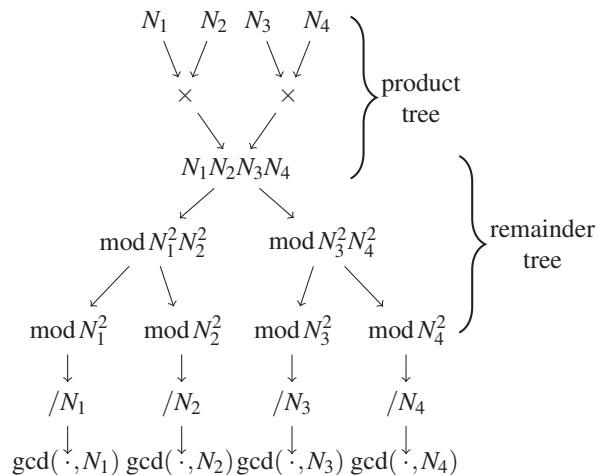


Figure 1: **Computing all-pairs GCDs efficiently** — We computed the GCD of every pair of RSA moduli in our dataset using an algorithm due to Bernstein [6].

Algorithm 1 Quasilinear GCD finding

Input: N_1, \dots, N_m RSA moduli

- 1: Compute $P = \prod N_i$ using a product tree.
- 2: Compute $z_i = (P \bmod N_i^2)$ for all i using a remainder tree.

Output: $\gcd(N_i, z_i/N_i)$ for all i .

A product tree computes the product of m numbers by constructing a binary tree of products. A remainder tree computes the remainder of an integer modulo many integers by successively computing remainders for each node in their product tree. For further discussion, see [7].

The final output of the algorithm is the GCD of each modulus with the product of all the other moduli. We are interested in the moduli for which this GCD is not 1. However, if a modulus shares both of its prime factors with two other distinct moduli, then the GCD will be the modulus itself rather than one of its prime factors. This occurred in a handful of instances in our dataset; we factored these moduli using the naïve quadratic algorithm for pairwise GCDs.

We implemented the algorithm in C using the GMP library for the arithmetic operations and ran it on the 11,170,883 distinct RSA moduli from our TLS and SSH datasets and the EFF SSL Observatory [18] dataset.

The entire computation finished in 5.5 hours using a single core on a machine with a 3.30 GHz Intel Core i5 processor and 32 GB of RAM. The remainder tree took approximately ten times as long to process as the product tree. Parallelized across sixteen cores on an EC2 Cluster Compute Eight Extra Large Instance with 60.5 GB of RAM and using EBS-backed storage for scratch data, the same computation took 1.3 hours at a cost of about \$5.

4 Vulnerabilities

We analyzed the data from our TLS and SSH scans and identified several patterns of vulnerability that would have been difficult to detect without a macroscopic view of the Internet. This section discusses the details of these problems, as summarized in Table 2.

4.1 Repeated keys

We found that 7,770,232 of the TLS hosts (61%) and 6,642,222 of the SSH hosts (65%) served the same key as another host in our scans. To understand why, we clustered certificates and host keys that shared the same public key and manually inspected representatives of the largest clusters. In all but a few cases, the TLS certificate subjects, SSH version strings, or WHOIS information were identical within a cluster, or pointed to a single manufacturer or organization. This sometimes suggested an explanation for the shared keys.

Not all of the repeated keys were due to vulnerabilities. For instance, many of the most commonly repeated keys appeared in shared hosting situations. Six of the ten most common DSA host keys and three of the ten most common RSA host keys were served by large hosting providers (see Figure 2). Another frequent reason for repeated keys was distinct TLS certificates all belonging to the same organization. For example, TLS hosts at google.com, appspot.com, and doubleclick.net all served distinct certificates with the same public key. We excluded these cases and attributed remaining clusters of shared keys to several classes of problems.

Default keys A common reason for hosts to share the same key that we *do* consider a vulnerability is

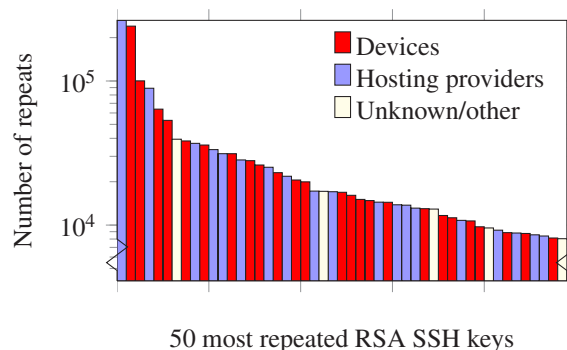


Figure 2: **Commonly repeated SSH keys** — We investigated the 50 most repeated SSH host keys for both RSA and DSA. Nearly all of the repeats appeared to be due either to hosting providers using a single key on many IP addresses or to devices that used a default key or generated keys using insufficient entropy. Note log scale.

	Our TLS Scan	Our SSH Scans
Number of live hosts	12,828,613 (100.00%)	10,216,363 (100.00%)
... using repeated keys	7,770,232 (60.50%)	6,642,222 (65.00%)
... using vulnerable repeated keys	714,243 (5.57%)	981,166 (9.60%)
... using default certificates or default keys	670,391 (5.23%)	
... using low-entropy repeated keys	43,852 (0.34%)	
... using RSA keys we could factor	64,081 (0.50%)	2,459 (0.03%)
... using DSA keys we could compromise		105,728 (1.03%)
... using Debian weak keys	4,147 (0.03%)	53,141 (0.52%)
... using 512-bit RSA keys	123,038 (0.96%)	8,459 (0.08%)
... identified as a vulnerable device model	985,031 (7.68%)	1,070,522 (10.48%)
... model using low-entropy repeated keys	314,640 (2.45%)	

Table 2: **Summary of vulnerabilities** — We analyzed our TLS and SSH scan results to measure the population of hosts exhibiting several entropy-related vulnerabilities. These include use of repeated keys, use of RSA keys that were factorable due to repeated primes, and use of DSA keys that were compromised by repeated signature randomness. Under the theory that vulnerable repeated keys were generated by embedded or headless devices with defective designs, we also report the number of hosts that we identified as these device models. Many of these hosts may be at risk even though we did not specifically observe repeats of their keys.

manufacturer-default keys. These are preconfigured in the firmware of many devices, such that every device of a given model shares the same key pair unless the user changes it. The private keys to these devices may be accessible through reverse engineering, and published databases of default keys such as littleblackbox [24] contain private keys for thousands of firmware releases.

At least 670,391 (5.23%) of the TLS hosts appeared to serve manufacturer-default certificates or keys. We classified a certificate as a manufacturer default if nearly all the devices of a given model used identical certificates, or if the certificate was labeled as a default certificate.

The most common default certificate that we could ascribe to a particular device belonged to a model of consumer router. Our scan uncovered 90,779 instances of this device model sharing a single certificate. We also found a variety of enterprise products serving default keys, including server management devices, network storage devices, routers, remote access devices, and VoIP devices.

For most of the repeated SSH keys, the lack of uniquely identifying host information prevents us from distinguishing default keys from keys generated with insufficient entropy, so we address these together in the next section.

Repeated keys due to low entropy Another common reason that hosts share the same key appears to be entropy problems during key generation. In these instances, when we investigated a key cluster, we would typically see thousands of hosts across many address ranges, and, when we checked the keys corresponding to other instances of the same model of device, we would see a long-tailed distribution in their frequencies. Intuitively, this type of

distribution suggests that the key generation process may be using insufficient entropy, with distinct keys due to relatively rare events. For TLS, our investigations began with the keys that occurred in at least 100 distinct self-signed certificates. For SSH, we started from the 50 most commonly repeated keys for each of RSA (appearing on more than 8000 hosts) and DSA (more than 4000 hosts).

With this process, we identified 43,852 TLS hosts (0.34%) that served repeated keys due apparently to low entropy during key generation. 27,545 certificates (98%) containing these repeated keys were self-signed; all 577 CA-signed certificates identified Iomega StorCenter network storage devices. For most SSH hosts we were unable to distinguish between default keys and keys repeated due to entropy problems, but 981,166 of the SSH hosts (9.60%) served keys repeated for one of these reasons.

We used the techniques described in Section 3.2 to identify apparently vulnerable devices from 27 manufacturers. These include enterprise-grade routers from Cisco and Juniper; server management cards from Dell, Hewlett-Packard, and IBM; virtual-private-network (VPN) devices; building security systems; network attached storage devices; and several kinds of consumer routers and VoIP products.

Duplicated keys are a red flag that calls the security of the device’s key generation process into question, and all keys generated with the same model device should be considered suspect. For 14 of the TLS devices generating repeated keys, we were able to infer a fingerprint for the device model and estimate the total population of the device in our scan. The prevalence of duplicated keys varied greatly for different device models, from as low as 0.2%

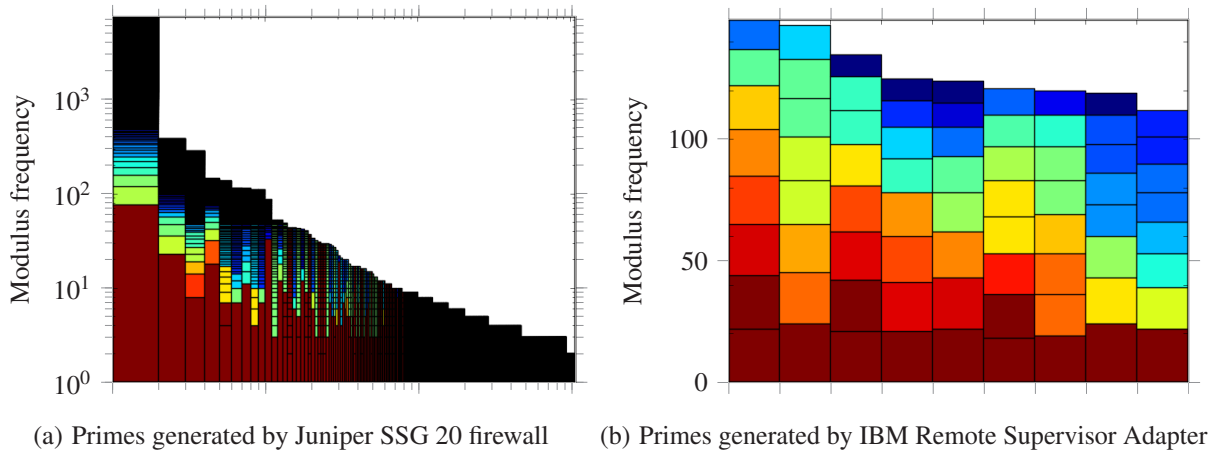


Figure 3: **Visualizing RSA common factors** — Different implementations displayed different patterns of vulnerable keys. These plots depict the distribution of vulnerable keys divisible by common factors generated by two different device models. Each column represents a collection of RSA moduli divisible by a single prime factor p . The color and internal rectangles show, for each p , the frequencies of each distinct prime factor q . The Juniper device (*left*; note log-log scale) follows a long-tailed distribution that is typical of many of the devices we identified. In contrast, the IBM remote access device (*right*) was unique among those we observed in that it generates RSA moduli roughly uniformly distributed among nine distinct prime factors.

in the case of one router to 98% for one thin client. The total population of these identified, potentially vulnerable TLS devices was 314,640 hosts, which represents 2.45% of the TLS hosts in our scan.

In the above analyses, we excluded repeated keys that were due to the infamous Debian weak-key vulnerability [5, 37], which we report separately in Table 2.

4.2 Factorable RSA keys

A second way that entropy problems might manifest themselves during key generation is if an RSA modulus shares one of its prime factors p or q with another key. As explained in Section 2.1, finding such a pair immediately allows an adversary to efficiently factor both moduli and obtain their private keys. In order to find such keys, we computed the GCD of all pairs of distinct RSA moduli by applying the algorithm described in Section 3.3.

The 11,170,883 distinct RSA moduli yielded 2,314 distinct prime divisors, which divided 16,717 distinct public keys. This allowed us to obtain private keys for 23,576 (0.40%) of the TLS certificates in our scan data, which were served on 64,081 (0.50%) of the TLS hosts, and 1,013 (0.02%) of the RSA SSH host keys, which were served on 2,459 (0.027%) of the RSA SSH hosts.

The vast majority of the vulnerable keys appeared to be system-generated certificates and SSH host keys used by routers, firewalls, remote administration cards, and other types of headless or embedded network devices. Only two of the factorable TLS certificates had been signed by

a browser trusted authority and both have expired. Some devices generated factorable keys both for TLS and SSH, and a handful of devices shared common factors across SSH and TLS keys.

We classified these factorable keys in a similar manner to the repeated keys. We found that, in all but a small number of cases, the TLS certificates and SSH host keys divisible by a common factor all belonged to a particular manufacturer, which we were able to identify in most cases using the techniques described in Section 3.2.

We identified devices from 41 manufacturers in this way, which constituted 99% of the hosts that generated RSA keys we could factor. The devices range from 100% (576 of 576 devices) vulnerable to 0.01% vulnerable (2 out of 10,932). As with repeated keys, we would not expect to see well-generated cofactorable keys; any device model observed generating factorable keys should be treated as potentially vulnerable.

The majority of the devices serving factorable keys were Juniper firewalls. We identified 46,993 of these devices in our dataset, and we factored the keys for 12,688 (27%). Of these keys, 7,510 (59%) share a single common divisor. The distribution of common factors among its moduli is shown in Figure 3a.

The most remarkable devices were IBM Remote Server Administration cards and BladeCenter devices, which displayed a distribution of factors unlike any of the other vulnerable devices we found. There were only 9 distinct prime factors that had been used to generate the keys for 576 devices. Each device’s key was the product of two

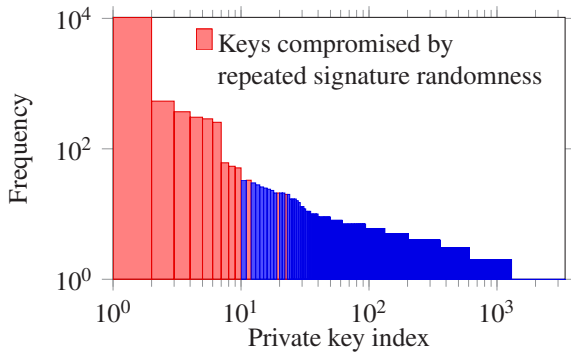


Figure 4: **Vulnerable DSA keys for one SSH device** — We identified 18,684 SSH DSA keys that appeared to have been generated by Gigaset DSL routers, of which 16,575 were repeated at least once. Shown in red in this log-log plot are 12,378 keys further compromised due to repeated DSA signature values.

different primes from this list. The 36 possible moduli that could be generated with this process were roughly uniformly distributed, as shown in Figure 3b. In addition, this was the only device we observed to generate RSA moduli where *both* prime factors were shared with other keys.

4.3 DSA signature weaknesses

The third class of entropy-related vulnerability that we searched for was repeated ephemeral keys in DSA signatures. As explained in Section 2.2, if a DSA key is used to sign two different messages using the same ephemeral key, then the long-term private key is immediately computable from the public key and the signatures. The presence of this vulnerability indicates entropy problems at later phases of operation, after initial key generation. We searched for signatures from identical keys containing repeated r values. Then we used the method in Section 2.2 to compute the corresponding private keys.

Our combined SSH DSA scans collected 9,114,925 signatures (in most cases, two from each SSH host serving a DSA-based key) of which 4,365 (0.05%) contained the same r value as at least one other signature. 4,094 of these signatures (94%) used the same r value and the same public key. This allowed us to compute the 281 distinct private keys used to generate these signatures. These compromised keys were served by 105,728 (1.6%) of the SSH DSA hosts in our combined scans.

We clustered the vulnerable signatures by r values and manufacturer. 2,026 (46%) of the colliding r values appeared to have been generated by Gigaset SX762 consumer-grade DSL routers and revealed private keys for 17,349 (66%) of the 26,374 hosts we identified as this device model (see Figure 4). Another 934 signa-

ture collisions appeared to be from ADTran Total Access business-grade phone/network routers and revealed private keys for 62,807 (73%) out of 86,301 such hosts. Several vulnerable device models, including the IBM RSA II remote administration cards and Juniper NetScreen, also generated factorable RSA keys.

While we collected multiple signatures from some SSH hosts, 3,917 (89.7%) out of 4,365 of the collisions were from *different* hosts that had generated the same long-term key and also used the same ephemeral key during the key exchange protocol. This problem compounds the danger of the repeated key vulnerability: a single signature collision between any pair of hosts sharing the same key at any point during runtime reveals the private key for every host using that key. In our dataset we observed tens of thousands of hosts using the same public key. While a single host may never repeat an ephemeral key, two hosts sharing a private key due to poor entropy can put each others' keys at risk.

We note that any estimation of vulnerability based on our data is an extreme lower bound, as we gathered at most two signatures from each host in our scans. It is likely that many more private keys would be revealed if we collected additional signatures.

5 Experimental Investigation

Based on the results the previous section, we conjectured that the problems we observed were an implementational phenomenon. To more deeply understand the causes, we augmented our data analysis with experimental investigation of specific implementations. While there are many independently vulnerable implementations, we chose to examine three open-source cryptographic software components that appeared frequently in the vulnerable populations.

5.1 Weak entropy and the Linux RNG

We conjectured that the cause for many of the entropy problems we observed began with insufficient randomness provided by the operating system. This led us to take an in-depth look at the Linux random number generator (RNG). We note that not every vulnerable key was generated on Linux; we also observed vulnerable keys on FreeBSD and Windows systems, and similar vulnerabilities to those we describe here have been reported with BSD's `arc4random` [36].

The Linux RNG maintains three entropy pools, each with an associated counter that estimates how much fresh entropy it has available. Fresh entropy from unpredictable kernel sources is periodically mixed into the *Input pool*. When processes read from `/dev/random` or `/dev/urandom`, the kernel extracts the requested amount

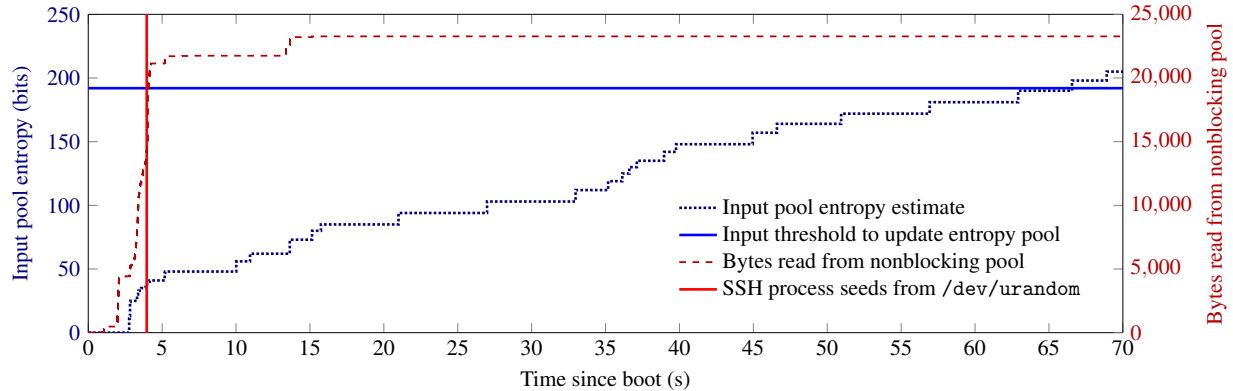


Figure 5: **Linux urandom boot-time entropy hole** — We instrumented an Ubuntu Server 10.04 system to record its estimate of the entropy contained in the Input entropy pool during a typical boot. Linux does not mix Input pool entropy into the Nonblocking pool that supplies `/dev/urandom` until the Input pool exceeds a threshold of 192 bits (blue horizontal line), which occurs here at 66 seconds post-boot. For comparison, we show the cumulative number of bytes generated from the Nonblocking entropy pool; the vertical red line marks the time when OpenSSH seeds its internal PRNG by reading from `urandom`, *well before* this facility is ready for secure use.

of entropy from the Input pool and mixes it into the *Blocking* or *Nonblocking* pool, respectively, and then extracts bytes from the respective pool to return. If the input pool does not contain enough entropy to satisfy the request, the read from the Blocking pool blocks; the Nonblocking pool read is satisfied immediately.

Entropy sources We experimented with the Linux 2.6.35 kernel to exhaustively determine the sources of entropy used by the RNG. To do this, we traced through the kernel source code and systematically disabled entropy sources until the RNG output was repeatable. All of the entropy sources we found are greatly weakened under certain operating conditions.

The explicit entropy sources we observed are the uninitialized contents of the pool buffers when the kernel starts, the startup clock time in nanosecond resolution, input event and disk access timings, and entropy saved across boots to a local file. Surprisingly, modern Linux systems no longer collect entropy from IRQ timings.

The final and most interesting entropy source was one that we have not seen documented elsewhere. The developers chose not to put a lock around the mixing procedure when entropy is extracted from the pool, and, as a result, if two threads extract entropy concurrently, the pool contents may change anywhere in the middle of the hash computation, resulting in the introduction of significant (but uncredited) entropy to the pool.

The removal of IRQs as an entropy source has likely exacerbated RNG problems in headless and embedded devices, which often lack human input devices, disks, and multiple cores. If they do, the only source of entropy—if there are any at all—may be the time of boot.

Experiment To test whether Linux’s `/dev/urandom` can produce repeatable output in conditions resembling the initial boot of a headless or embedded networked device, we modified the 2.6.35 kernel to add instrumentation to the RNG and disable certain entropy sources to simulate a cold boot on a low-end machine without a working clock.

We experimented with this kernel on a Dell Optiplex 980 system using a fresh installation of Ubuntu server 10.04.4. The machine was configured with a Core i7 CPU, 4 GB RAM, a 32 GB SSD, and a USB keyboard. It was attached to a university office LAN and obtained an IP address using DHCP. With this configuration, we performed 1,000 unattended boots. Each time, we read 32 bytes from `urandom` at the point in the initialization process where the SSH server would normally start. Under these conditions, we found that the output of `/dev/urandom` was entirely predictable and repeatable.

The kernel maintains a reserve threshold for the Input pool, and no data is copied into the Nonblocking pool until the Input pool has been credited with at least that much entropy (192 bits, for our kernel). Figure 5 shows the cumulative amount of entropy credited to the Input pool during a typical bootup from our tests. (Note that none of the entropy sources we disabled would have resulted in more entropy being *credited* to the pool.) The credited entropy does not cross this reserve threshold until more than a minute after boot, well after the SSH server and other startup processes have launched. Although Ubuntu tries to restore entropy saved during the last shutdown, this happens slightly *after* the point when `sshd` first reads from `urandom`. With no entropic inputs, `urandom` produces a deterministic output stream.

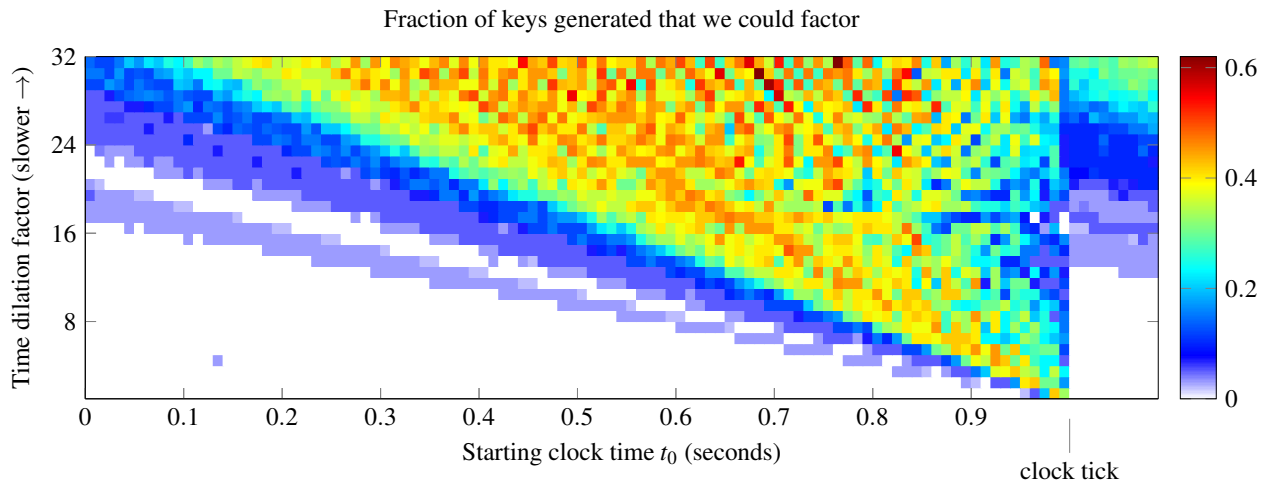


Figure 6: **OpenSSL generating factorable keys**— We hypothesized that OpenSSL can generate factorable keys under low-entropy conditions due to slight asynchronicity between the key generation process and the real-time clock, we generated 14 million RSA keys using controlled entropy sources for a range of starting clock times. Each square in the plot indicates the fraction of 100 generated keys that we could factor. In many cases (white), keys are repeated but never share primes. After a sudden phase change, factorable keys occur during a range leading up to the second boundary, and that range increases as we simulate machines with slower execution speeds.

Boot-time entropy hole The entropy sources we disabled would likely be missing in some headless and embedded systems, particularly on first boot. This means that there is a window of vulnerability—a boot-time entropy hole—during which Linux’s `urandom` may be entirely predictable, at least for single-core systems. If processes generate long-term cryptographic keys or maintain their own entropy pools seeded only with entropy gathered during this window, those keys are likely to be vulnerable. The risk is particularly high for unattended systems that ship with preconfigured operating systems and generate SSH or TLS keys the first time the respective daemons start during the initial boot.

On stock Ubuntu systems, these risks are somewhat mitigated: TLS keys must be generated manually, and OpenSSH host keys are generated during package installation, which is likely to be late in the install process, giving the system time to collect sufficient entropy. However, on the Fedora, Red Hat Enterprise Linux (RHEL), and CentOS Linux distributions, OpenSSH is installed by default, and host keys are generated on first boot. We experimented further with RHEL 5 and 6 to determine whether host keys on these systems might be compromised, and observed that sufficient entropy had been collected at the time of key generation by a slim margin. We believe that most server systems running these distributions are safe, particularly since they likely have multiple cores and gather additional entropy from physical concurrency. However, it is possible that other distributions and customized installations do not collect sufficient entropy on startup and generate weak keys on first boot.

5.2 Factorable RSA keys and OpenSSL

One interesting question raised by our vulnerability results is why factorable RSA keys occur at all. A naïve implementation of RSA key generation would simply seed a PRNG from the operating system’s entropy pool and then use it to generate p and q . In this approach, we would expect to see duplicate keys if the OS provided the same seed, but factorable keys would be extremely unlikely. What we see instead is that some devices seem prone to generating keys with common factors. Another curious feature is that although some of the most common prime factors divided hundreds of different moduli, in nearly all of these cases the second prime factor did not divide any other keys.

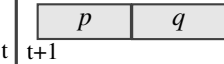
One explanation for this pattern is that implementations updated their entropy pools in the middle of key generation. In this case, the entropy pool states might be identical as distinct key generation processes generate the first prime p and diverge while generating the second prime q . In order to explore this theory, we studied the source code of OpenSSL [14], one of the most widely used open-source cryptographic libraries. OpenSSL is not the only software library responsible for the problems we observed, but we chose to examine it because the source code is freely available and because of its popularity.

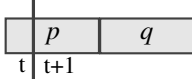
OpenSSL RSA key generation OpenSSL’s built-in RSA key generator relies on an internal entropy pool maintained by OpenSSL. The entropy pool is seeded on first use with (on Linux) 32 bytes read from `/dev/urandom`, the process ID, user ID, and the current time in seconds.

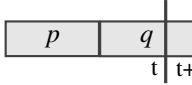
OpenSSL provides a function named `bnrand()` to generate cryptographically-sized integers from the entropy pool, which, on each call, mixes into the entropy pool the current time in seconds, the process ID, and the possibly uninitialized contents of a destination buffer.

The RSA key generation algorithm generates the primes p and q using a randomized algorithm. During this process, OpenSSL extracts entropy from the entropy pool dozens to hundreds of times. Since we observed many keys with one prime factor in common, we can conjecture that multiple systems are starting with `urandom` and the time in the same state and that the entropy pool states diverge due to the addition of time during intermediate steps of the key generation process.

We hypothesized that this process is hypersensitive to small variations in where the *boundary between seconds* falls. Slight variations in execution speed might cause the wall clock tick to fall between different calls to `bnrand()`, resulting in different execution paths. This can result in several different behaviors, with three simple cases:

If the second never changes while computing p and q , every execution will generate identical keys. 

If the clock ticks while generating p , both p and q diverge, yielding distinct keys with no shared factors. 

If instead the clock advances to the next second during the generation of the second prime q , then two executions will generate identical primes p but can generate distinct primes q based on exactly when the second changes. 

Experiment To test our hypothesis, we modified OpenSSL 1.0.0g to control all the entropy inputs used during key generation, generated a large number of RSA keys, and determined how many were identical or factorable. To simulate the effects of slower clock speeds, we dilated the clock time returned by `time()` and repeated the experiment using dilation multipliers of 1–32. In all, we generated 14 million keys. We checked for common factors within each batch of 100 keys.

The results we obtained, illustrated in Figure 6, are consistent with our hypothesis. No factorable keys are generated for low starting offsets, as both p and q are generated before the second changes. As the initial offset increases, there is a rapid phase change to generating factorable keys, as generation of q values begins to overlap the second boundary. Eventually, the fraction of factorable keys falls as the second boundary occurs during the generation of more p values, resulting in distinct moduli with no common factors.

5.3 DSA signature weaknesses and Dropbear

The DSA signature vulnerabilities we observed indicate that entropy problems impact not only key generation but also the continued runtime behavior of server software during normal usage. This is somewhat surprising, since we might expect the operating system to collect sufficient entropy *eventually*, even in embedded devices. We investigated Dropbear, a popular light-weight SSH implementation. It maintains its own entropy pools seeded from the operating system at launch, on Linux with 32 bytes read from `urandom`. This suggests a possible explanation for the observed problems: the operating system had not collected enough entropy when the SSH server started, and, from then on, even though the system entropy pool may have had further entropy, the running SSH daemon did not.

To better understand why these programs produce vulnerable DSA signatures, we examined the source code for the current version of Dropbear, 0.55. The ephemeral key is generated as output from an internal entropy pool. Whenever Dropbear extracts data from its entropy pool, it increments a static counter and hashes the result into the pool state. No additional randomness is added until the counter (a 32-bit integer) overflows. This implies that, if two Dropbear servers are initially seeded with the same value from `urandom`, they will provide identical signature randomness as long as their counters remain synchronized and do not overflow.

(We note that Dropbear contains a routine to generate k in a manner dependent on the message to be signed, which would ensure that distinct messages are always signed with distinct k values and protect against the vulnerability that we explore here. However, that code is disabled by default.)

We looked for evidence of synchronized sequences of ephemeral keys in the wild by making further SSH requests to a handful of the Dropbear hosts from our scan. We chose two hosts with the SSH version string `dropbear-0.39` that had used identical DSA public keys and r values and found that the signatures followed an identical sequence of r values. We could advance the sequence of one host by making several SSH requests, then cause the other host to catch up by making the same number of requests. When probed again an hour later, both hosts remained in sync. This suggests that the Dropbear code is causing vulnerabilities on real hosts in the manner we predicted.

Several other implementations, including hosts identifying OpenSSH and the Siemens Gigaset routers displayed similar behavior when rescanned. Because OpenSSL adds the current clock time to the entropy pool before extracting these random values, this suggests that some of these devices do not have a working clock at all.

6 Discussion

6.1 RSA vs. DSA in the face of low entropy

We believe that the DSA signature vulnerabilities pose more cause for concern than the RSA factorization vulnerability. The RSA key factorization vulnerability that we investigated occurs only for certain patterns of key generation implementations in the presence of low entropy. In contrast, the DSA signature vulnerability can compromise any DSA private key—no matter how well generated—if there is ever insufficient entropy at the time the key is used for signing. It is not necessary to search for a collision, as we did; it suffices for an attacker to be able to guess the ephemeral private key k . The most analogous attacks against RSA of which we are aware show that some types of padding schemes can allow an attacker to discover the encrypted plaintext or forge signatures [10]. We are unaware of any attacks that use compromised RSA signatures to recover the private key.

We note that our findings show a larger fraction of SSH hosts are compromised by the DSA vulnerability than by factorable RSA keys, even though our scanning techniques have likely only revealed a small fraction of the hosts prone to repeating DSA signature randomness. In contrast, the factoring algorithm we used has found all of the repeated RSA primes in our sample of keys.

There are specific countermeasures that implementations can use to protect against these attacks. If both prime factors of an RSA modulus are generated from a PRNG without adding additional randomness during key generation, then low entropy would result in repeated but not factorable keys. These are more readily observable, but may be trickier to exploit, because they do not immediately reveal the private key to a remote attacker. To prevent DSA randomness collisions, the randomness for each signature can be generated as a function of the message and the pseudorandom input. (It is very important to use a cryptographically secure PRNG for this process [4].) Of course, the most important countermeasure is for implementations to use sufficient entropy during cryptographic operations that require randomness, but defense-in-depth remains the prudent course.

6.2 /dev/(u)random as a usability failure

The Linux documentation states that “[a]s a general rule, `urandom` should be used for everything except long-lived GPG/SSL/SSH keys” [1]. However, all the open-source implementations we examined used `random` to generate keys by default. Based on a survey of developer mailing lists and forums, it appears that this choice is motivated by two factors: `random`’s extremely conservative behavior and the mistaken perception that `urandom`’s output is secure enough.

As others have noted, Linux is very conservative at crediting randomness added to the entropy pool [23], and `random` further insists on using *freshly collected* randomness that has not already been mixed into the output PRNG. The blocking behavior means that applications that read from `random` can hang unpredictably, and, in a headless device without human input or disk entropy, there may never be enough input for a read to complete. While blocking is intended to be an indicator that the system is running low on entropy, `random` often blocks even though the system has collected more than enough entropy to produce cryptographically strong PRNG output—in a sense, `random` is often “crying wolf” when it blocks.

Our experiments suggest that many of the vulnerabilities we observed may be due to the output of `urandom` being used to seed entropy pools before any entropic inputs have been mixed in. Unfortunately, the existing interface to `urandom` gives the operating system no means of alerting applications to this dangerous condition. Our recommendation is that Linux should add a secure RNG that blocks until it has collected adequate seed entropy and thereafter behaves like `urandom`.

6.3 Are we seeing only the tip of the iceberg?

Nearly all of the vulnerable hosts that we were able to identify were headless or embedded devices. This raises the question of whether the problems we found appear *only* in these types of devices, or if instead we are merely seeing the tip of a much larger iceberg.

Based on the experiments described in Section 5.1, we conjecture that there may exist further classes of vulnerable keys that were not visible to our methods, but could be compromised with targeted attacks. The first class is composed of embedded or headless devices with an accurate real-time clock. In these cases, keys generated during the boot-time entropy hole may appear distinct, but depend only on a configuration-specific state and the boot time. These keys would not appear vulnerable in our scanning, but an attacker may be able to enumerate some or all of such a reduced key space for targeted implementations.

A more speculative class of potential vulnerability consists of traditional PC systems that automatically generate cryptographic keys on first boot. We observed in Section 5.1 that an experimental machine running RHEL 5 and 6 did collect sufficient entropy in time for SSH key generation, but that the margin of safety was slim. It is conceivable that some lower-resource systems may be vulnerable.

Finally, our study was only able to detect vulnerable DSA ephemeral keys under specific circumstances where a large number of systems shared the same long-term key and were choosing ephemeral keys from the same small set. There may be a larger set of hosts using ephemeral

keys that do not repeat across different systems but are nonetheless vulnerable to a targeted attack.

We found no evidence suggesting that RSA keys from standard implementations that were generated interactively or subsequent to initial boot are vulnerable.

6.4 Directions for future work

In this work, we examined keys from two cryptographic algorithms on two protocols visible via Internet-wide scans of two ports. A natural direction for future work is to broaden the scope of all of these choices. Entropy problems can also affect the choice of Diffie-Hellman key parameters and keying material for symmetric ciphers. In addition, there are many more subtle attacks against RSA, DSA, and ECDSA that we did not search for. We focused on keys, but one might also try to search for evidence of repeated randomness in initialization vectors in ciphertext or salts in cryptographic hashes.

We also focused solely on services visible to our scans of the public Internet. Similar vulnerabilities might be found by applying this methodology to keys or other cryptographic data obtained from other resource-constrained devices that perform cryptographic operations, such as smart cards or mobile phones.

The observation that `urandom` can produce predictable output on some types of systems at boot may lead to attacks on other services that automatically begin at boot and depend on good randomness from the kernel. It warrants investigation to determine whether this behavior may undermine other security mechanisms such as address space layout randomization or TCP initial sequence numbers.

7 Defenses and Lessons

The vulnerabilities we have identified are a reminder that secure random number generation continues to be a challenging problem. There is a tendency for developers at each layer of the software stack to silently shift responsibility to other layers; a far better practice would be a defense-in-depth approach where developers at every layer apply careful security design and testing and make assumptions clear. We suggest defensive strategies and lessons for several important groups of stakeholders.

For OS developers:

Provide the RNG interface applications need. Typical security applications require a source of randomness that is guaranteed to be of high quality and has predictable performance; neither Linux's `/dev/random` nor `/dev/urandom` strikes this balance. The operating system should maintain a secure PRNG that refuses to return data until it has been seeded with a minimum amount

of true randomness and is continually seeded with fresh entropy collected during operation.

Communicate entropy conditions to applications. The problem with `/dev/urandom` is that it can return data even before it has been seeded with any entropy. The OS should provide an interface to indicate how much entropy it has mixed into its PRNG, so that applications can gauge whether the state is sufficiently secure for their needs.

Test RNGs thoroughly on diverse platforms. Many of the entropy sources that Linux supports are not available on headless or embedded devices. These behaviors may not be apparent to OS developers unless they routinely test the internals of the entropy collection subsystem across the full spectrum on platforms the system supports.

For library developers:

Default to the most secure configuration. Both OpenSSL and Dropbear default to using `/dev/urandom` instead of `/dev/random`, and Dropbear defaults to using a less secure DSA signature randomness technique even though a more secure technique is available as an option. In general, cryptographic libraries should default to using the most secure mechanisms available.

Use RSA and DSA defensively. Crypto libraries can take specific steps to prevent weak entropy from resulting in the immediate leak of private keys due to co-factorable RSA moduli and repeated DSA signature randomness (see Section 6.1).

For application developers:

Generate keys on first use, not on install or first boot. If keys must be generated automatically, it may be better to defer generation until the keys are needed.

Heed warnings from below. If the OS or cryptography library being used raises a signal that insufficient entropy is available (such as blocking), applications should detect this signal and refuse to perform security-critical operations until the system recovers from this potentially vulnerable state. Developers have been known to work around low-entropy states by ignoring or disabling such warnings, with extremely dangerous results [22].

For device manufacturers:

Avoid factory-default keys or certificates. While some defense is better than nothing, default keys and certificates provide only minimal protection.

Seed entropy at the factory. Devices could be initialized with truly random seeds at the factory. Sometimes it is already necessary to configure unique state on the assembly line (such as to set MAC addresses), and entropy could be added at the same time.

Ensure entropy sources are effective. Embedded or headless devices may not have access to sources of randomness assumed by the operating system, such as user-input devices or disk timing. Device makers should ensure that effective entropy sources are present, and that these are being harvested in advance of cryptographic operations.

Use hardware random number generators when possible. Security-critical devices should use a hardware random number generator for cryptographic randomness whenever possible.

For certificate authorities:

Check for repeated, weak, and factorable keys Certificate authorities have a uniquely broad view of keys contained in TLS certificates. We recommend that they repeat our work against their certificate databases and take steps to protect their customers by alerting them to potentially weak keys.

For end users:

Regenerate default or automatically generated keys. Cryptographic keys and certificates that were shipped with the device or automatically generated at first boot should be manually regenerated. Ideally, certificates and keys should be generated on another device (such as a desktop system) with access to adequate entropy.

Check for known weak keys. We have created a key-check service that individuals can use to check their TLS certificates and SSH host keys against our database of keys we have identified as vulnerable.

For security and crypto researchers:

Secure randomness remains unsolved in practice. The fact that all major operating systems now provide cryptographic RNGs might lead security experts to believe that any entropy problems that still occur are the fault of developers taking foolish shortcuts. Our findings suggest otherwise: entropy-related vulnerabilities can result from complex interaction between hardware, operating systems, applications, and cryptographic primitives. We have yet to develop the engineering practices and principles necessary to make predictably secure use of unpredictable randomness across the diverse variety of systems where it is required.

Primitives should fail gracefully under weak entropy. Cryptographic primitives are usually designed to be secure under ideal conditions, but practice will subject them to conditions that are less than ideal. We find that RSA and DSA, with surprising frequency, are used in practice under weak entropy scenarios where, due to the design of these cryptosystems, the private keys are totally compromised. More attention is needed to ensure that future primitives degrade gracefully under likely failure modes such as this.

8 Related Work

HTTPS surveys The HTTPS public-key infrastructure has been a focus of attention in recent years, and researchers have performed several large-scale scans to measure TLS usage and CA behavior. In contrast, our study addresses problems that are mostly separate from the CA ecosystem.

In 2010, the Electronic Frontier Foundation (EFF) and iSEC Partners debuted the SSL Observatory project [18] and released the largest public repository of TLS certificates. The authors used their data to analyze the CA infrastructure and noted several vulnerabilities. We owe the inspiration for our work to their fascinating dataset, in which we first identified several of the entropy problems we describe; however, we ultimately performed our own scans to have more up-to-date and complete data.

In 2011, Holz et al. [26] scanned the Alexa top 1 million domains and observed TLS sessions passing through the Munich Scientific Research Network (MWN). Their study recorded 960,000 certificates and was the largest academic study of TLS data at the time. They report many statistics gathered from their survey, mainly focusing on the state of the CA infrastructure. We note that they examined repeated keys and dismissed them as “curious, but not very frequent.” Yilek et al. [37] performed daily scans of 50,000 TLS servers over several months to track replacement time for certificates affected by the Debian weak key bug. Our count of Debian certificates provides another data point on this subject.

Problems with random number generation Several significant vulnerabilities relating to weak random number generation have been found in widely used software. In 1996, the Netscape browser’s SSL implementation was found to use fewer than a million possible seeds for its PRNG [19]. In May 2008, Bello discovered that the version of OpenSSL included in the Debian Linux distribution contained a bug that caused keys to be generated with only 15 bits of entropy [5]. The problem caused only 294,912 distinct keys to be generated per key size during a two year period before the error was found [37].

Gutmann [22] draws lessons about secure software design from the example of developer responses to an OpenSSL update intended to ensure that the entropy pool was properly seeded before use. He observes that many developers responded by working around the safety checks in ways that supplied no randomness whatsoever. The root cause, according to Gutmann, was that the OpenSSL design left the difficult job of supplying sufficient entropy to library users. He concludes that PRNGs should handle entropy-gathering themselves.

Guterman, Pinkas, and Reinmann analyzed the Linux random number generator in 2006 [23]. In contrast to

our analysis, which focuses on empirical measurement of an instrumented Linux kernel, theirs was based mainly on a review of the LRNG design. They point out several weaknesses from a cryptographic perspective, some of which have since been remedied. In a brief experimental section, they observe that the only entropy source used by the OpenWRT Linux distribution was network interrupts..

Weak entropy and cryptography In 2004, Bauer and Laurie [2] computed the pairwise GCDs of 18,000 RSA keys from the PGP web of trust and discovered a pair with a common factor of 9, demonstrating that the keys had been generated with broken (or omitted) primality testing.

The DSA signature weakness we investigate is well known and appears to be folklore. In 2010, the hacking group fail0verflow computed the ECDSA private key used for code signing on the Sony PS3 after observing that the signatures used repeated ephemeral keys [12]. Several more sophisticated attacks against DSA exist: Bellare, Goldwasser, and Miccancio [4] show that the private key is revealed if the ephemeral key is generated using a linear congruential generator, and Howgrave-Graham and Smart [27] give a method to compute the private key from a fraction of the bits of the ephemeral key.

Ristenpart and Yilek [34] developed “virtual machine reset” attacks in 2010 that induce repeated DSA ephemeral keys after a VM reset, and they implement “hedged” cryptography to protect against this type of randomness failure. Hedged public key encryption was introduced by Bellare et al. in 2009 and is designed to fail as gracefully as possible in the face of bad randomness [3].

As we were preparing this paper for submission, an independent group of researchers uploaded a preprint [31] reporting that they had computed the pairwise GCD of RSA moduli from the EFF SSL Observatory dataset and a database of PGP keys. Their work is concurrent and independent to our own; we were unaware of these authors’ efforts before their work was made public. They declined to report the GCD computation method they used. We responded by publishing a blog post [25] describing our GCD computation approach and summarizing some of the key findings we detail in this paper.

The authors of the concurrent work report similar results to our own on the fraction of keys that were able to be factored, and thus the two results provide validation for each other. In their paper, however, the authors draw very different conclusions than we do. They do not analyze the source of these entropy failures, and they conclude that RSA is “significantly riskier” than DSA. In contrast, we performed original scans that targeted SSH as well as TLS, and we looked for DSA repeated signature weaknesses as well as cofactorable RSA keys. We find that SSH DSA private keys are compromised at a higher rate than RSA keys, and we conclude that the fundamental problem is an implementational issue rather than a cryptographic one.

Furthermore, the authors of the concurrent work state that they “cannot explain the relative frequencies and appearance” of the weak keys they observed and report no attempt to determine their source. In this work, we performed extensive investigation to trace the vulnerable keys back to specific devices and software implementations, and we have notified the responsible developers and manufacturers. We find that the weak keys can be explained by specific design and implementation failures at various levels of the software stack, and we make detailed recommendations to developers and users that we hope will lessen the occurrence of these problems in the future.

9 Conclusion

In this work, we investigated the security of random number generation on a broad scale by performing and analyzing the most comprehensive Internet-wide scans of TLS certificates and SSH host keys to date. Using the global view provided by our data, we discovered that insecure RNGs are in widespread use, leading to a significant number of vulnerable RSA and DSA keys.

Our experiences suggest that the type of scanning and analysis we performed can be a useful tool for finding subtle flaws in cryptographic implementations, and we hope it will be applied more broadly in future work. Previous examples of random number generation flaws were found by painstakingly reverse engineering individual devices or implementations, or through luck when a collision was observed by a single user. Our scan data allowed us to essentially mine for vulnerabilities and detect problems in dozens of different devices and implementations in a single shot. Many of the collisions we found were too rare to ever have been observed by a single user but quickly became apparent with a near-global view of the universe of public keys. The results are a reminder to all that vulnerabilities can sometimes be hiding in plain sight.

Acknowledgments

The authors thank Dan Bernstein and Tanja Lange for discussion of batch factorization and OpenSSL, and Hovav Shacham for advice on many aspects of this work. We also thank Jake Appelbaum, Michael Bailey, Kevin Borders, Keith Brautigam, Ransom Briggs, Jesse Burns, Aleksander Durumeric, Prabal Dutta, Peter Eckersley, Andy Isaacson, James Kasten, Ben Laurie, Stephen Schultze, Ron Rivest, and David Robinson.

This material is based upon work supported by the National Science Foundation under Award No. DMS-1103803, the MURI program under AFOSR Grant No. FA9550-08-1-0352, and a National Science Foundation Graduate Research Fellowship.

References

- [1] random(4) Linux manual page. <http://www.kernel.org/doc/man-pages/online/pages/man4/random.4.html>.
- [2] BAUER, M., AND LAURIE, B. Factoring silly keys from the key servers. In *The Shoestring Foundation Weblog* (July 2004). <http://shoestringfoundation.org/cgi-bin/blosxom.cgi/2004/07/01#non-pgp-key>.
- [3] BELLARE, M., BRAKERSKI, Z., NAOR, M., RISTENPART, T., SEGEV, G., SHACHAM, H., AND YILEK, S. Hedged public-key encryption: How to protect against bad randomness. In *Proc. Asiacrypt 2009* (Dec. 2009), M. Matsui, Ed., pp. 232–249.
- [4] BELLARE, M., GOLDWASSER, S., AND MICCIANCIO, D. “Pseudo-random” generators within cryptographic applications: the DSS case. In *Advances in Cryptology—CRYPTO ’97* (Aug. 1997), B. S. Kaliski Jr., Ed., pp. 277–291.
- [5] BELLO, L. DSA-1571-1 OpenSSL—Predictable random number generator, 2008. Debian Security Advisory. <http://www.debian.org/security/2008/dsa-1571>.
- [6] BERNSTEIN, D. J. How to find the smooth parts of integers. <http://cr.yp.to/papers.html#smoothparts>.
- [7] BERNSTEIN, D. J. Fast multiplication and its applications. *Algorithmic Number Theory* (May 2008), 325–384.
- [8] BLUM, M., AND MICALI, S. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Comput.* 13, 4 (1984), 850–864.
- [9] BONEH, D. Twenty years of attacks on the RSA cryptosystem. *Notices of the AMS* 46, 2 (1999), 203–213.
- [10] BRIER, E., CLAVIER, C., CORON, J., AND NACCACHE, D. Cryptanalysis of RSA signatures with fixed-pattern padding. In *Advances in Cryptology—Crypto 2001*, pp. 433–439.
- [11] BROWN, D. R. L. Standards for efficient cryptography 1: Elliptic curve cryptography, 2009. <http://www.secg.org/download/aid-780/sec1-v2.pdf>.
- [12] BUSHING, MARCAN, SEGHER, AND SVEN. Console hacking 2010: PS3 epic fail. Talk at 27th Chaos Communication Congress (2010). http://events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf.
- [13] CHOR, B., AND GOLDREICH, O. Unbiased bits from sources of weak randomness and probabilistic communication complexity. In *Proc. 26th IEEE Symposium on Foundations of Computer Science* (1985), pp. 429–442.
- [14] COX, M., ENGELSCHALL, R., HENSON, S., LAURIE, B., ET AL. The OpenSSL project. <http://www.openssl.org>.
- [15] DAVIS, D., IHAKA, R., AND FENSTERMACHER, P. Cryptographic randomness from air turbulence in disk drives. In *Advances in Cryptology—CRYPTO ’94* (1994), pp. 114–120.
- [16] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol, Version 1.2. RFC 5246.
- [17] DORRENDORF, L., GUTTERMAN, Z., AND PINKAS, B. Cryptanalysis of the Windows random number generator. In *Proc. 14th ACM Conference on Computer and Communications Security* (2007), CCS ’07, pp. 476–485.
- [18] ECKERSLEY, P., AND BURNS, J. An observatory for the SSLiverse. Talk at Defcon 18 (2010). <https://www.eff.org/files/DefconSSLiverse.pdf>.
- [19] GOLDBERG, I., AND WAGNER, D. Randomness and the Netscape browser. *Dr. Dobbs’s Journal* 21, 1 (1996), 66–70.
- [20] GRANLUND, T., ET AL. The GNU multiple precision arithmetic library. <http://gmplib.org/>.
- [21] GUTMANN, P. Software generation of random numbers for cryptographic purposes. In *Proc. 7th USENIX Security Symposium* (1998), pp. 243–257.
- [22] GUTMANN, P. Lessons learned in implementing and deploying crypto software. In *Proc. 11th USENIX Security Symposium* (2002), pp. 315–325.
- [23] GUTTERMAN, Z., PINKAS, B., AND REINMAN, T. Analysis of the Linux random number generator. In *Proc. 2006 IEEE Symposium on Security and Privacy* (May 2006), pp. 371–385.
- [24] HEFFNER, C., ET AL. LittleBlackBox: Database of private SSL/SSH keys for embedded devices. <http://code.google.com/p/littleblackbox/>.
- [25] HENINGER, N., ET AL. There’s no need to panic over factorable keys—just mind your Ps and Qs. *Freedom to Tinker weblog* (2012). <https://freedom-to-tinker.com/blog/nadiah/new-research-theres-no-need-panic-over-factorable-keys-just-mind-your-ps-and-qs>.
- [26] HOLZ, R., BRAUN, L., KAMMENHUBER, N., AND CARLE, G. The SSL landscape—A thorough analysis of the X.509 PKI using active and passive measurements. In *Proc. 2011 ACM SIGCOMM Internet Measurement Conference* (2011), pp. 427–444.
- [27] HOWGRAVE-GRAHAM, N., AND SMART, N. Lattice attacks on digital signature schemes. *Designs, Codes and Cryptography* 23, 3 (2001), 283–290.
- [28] KLEINJUNG, T., AOKI, K., FRANKE, J., LENSTRA, A., THOMÉ, E., BOS, J., GAUDRY, P., KRUPPA, A., MONTGOMERY, P., OSVIK, D., TE RIELE, H., TIMOFEEV, A., AND ZIMMERMANN, P. Factorization of a 768-bit RSA modulus. In *Advances in Cryptology—CRYPTO 2010* (2010), T. Rabin, Ed., pp. 333–350.
- [29] LAWSON, N. DSA requirements for random k value, 2010. <http://rdist.root.org/2010/11/19/dsa-requirements-for-random-k-value/>.
- [30] LENSTRA, A., LENSTRA, H., MANASSE, M., AND POLLARD, J. The number field sieve. In *The development of the number field sieve*, A. Lenstra and H. Lenstra, Eds., vol. 1554 of *Lecture Notes in Mathematics*. 1993, pp. 11–42.
- [31] LENSTRA, A. K., HUGHES, J. P., AUGIER, M., BOS, J. W., KLEINJUNG, T., AND WACHTER, C. Ron was wrong, Whit is right. Cryptology ePrint Archive, Report 2012/064, 2012. <http://eprint.iacr.org/2012/064.pdf>.
- [32] LOCKE, G., AND GALLAGHER, P. FIPS PUB 186-3: Digital Signature Standard (DSS). Federal Information Processing Standards Publication (2009).
- [33] LYON, G. F. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure, USA, 2009.
- [34] RISTENPART, T., AND YILEK, S. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *Proc. ISOC Network and Distributed Security Symposium* (2010).
- [35] RIVEST, R. L., SHAMIR, A., AND ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* 21 (Feb. 1978), 120–126.
- [36] WOOLLEY, R., MURRAY, M., DOUNIN, M., AND ERMILOV, R. FreeBSD security advisory FreeBSD-SA-08:11.arc4random, 2008. <http://lists.freebsd.org/pipermail/freebsd-security-notifications/2008-November/000117.html>.
- [37] YILEK, S., RESCORLA, E., SHACHAM, H., ENRIGHT, B., AND SAVAGE, S. When private keys are public: Results from the 2008 Debian OpenSSL vulnerability. In *Proc. 2009 ACM SIGCOMM Internet Measurement Conference*, pp. 15–27.
- [38] YLONEN, T. SSH—secure login connections over the internet. In *Proc. 6th USENIX Security Symposium* (1996), pp. 37–42.
- [39] YLÖNEN, T., AND LONVICK, C. The secure shell (SSH) protocol architecture. <http://merlot.tools.ietf.org/html/rfc4251>.

TARDIS: Time and Remanence Decay in SRAM to Implement Secure Protocols on Embedded Devices without Clocks

Amir Rahmati
UMass Amherst

Mastooreh Salajegheh
UMass Amherst

Dan Holcomb
UC Berkeley

Jacob Sorber
Dartmouth College

Wayne P. Burleson
UMass Amherst

Kevin Fu
UMass Amherst

Abstract

Lack of a locally trustworthy clock makes security protocols challenging to implement on batteryless embedded devices such as contact smartcards, contactless smartcards, and RFID tags. A device that knows how much time has elapsed between queries from an untrusted reader could better protect against attacks that depend on the existence of a rate-unlimited encryption oracle.

The TARDIS (Time and Remanence Decay in SRAM) helps locally maintain a sense of time elapsed without power and without special-purpose hardware. The TARDIS software computes the expiration state of a timer by analyzing the decay of existing on-chip SRAM. The TARDIS enables coarse-grained, hourglass-like timers such that cryptographic software can more deliberately decide how to throttle its response rate. Our experiments demonstrate that the TARDIS can measure time ranging from seconds to several hours depending on hardware parameters. Key challenges to implementing a practical TARDIS include compensating for temperature and handling variation across hardware.

Our contributions are (1) the algorithmic building blocks for computing elapsed time from SRAM decay; (2) characterizing TARDIS behavior under different temperatures, capacitors, SRAM sizes, and chips; and (3) three proof-of-concept implementations that use the TARDIS to enable privacy-preserving RFID tags, to deter double swiping of contactless credit cards, and to increase the difficulty of brute-force attacks against e-passports.

1 Introduction

“Timestamps require a secure and accurate system clock—not a trivial problem in itself.”
—Bruce Schneier, *Applied Cryptography* [43]

Even a perfect cryptographic protocol can fail without a trustworthy source of time. The notion of a trustworthy clock is so fundamental that security protocols rarely state

Platform	Attack	#Queries
MIFARE Classic	Brute-force [15]	$\geq 1,500$
MIFARE DESFire	Side-channel [35]	250,000
UHF RFID tags	Side-channel [34]	200
TI DST	Reverse eng. [7, 8]	$\sim 75,000$
GSM SIM card	Brute-force [16]	150,000

Table 1: Practical attacks on intermittently powered devices. These attacks require repeated interactions between the reader and the device. Throttling the reader’s attempts to query the device could mitigate the attacks.

this assumption. While a continuously powered computer can maintain a reasonably accurate clock without trusting a third party, a batteryless device has no such luxury. Contact smartcards, contactless smartcards, and RFIDs can maintain a locally secured clock during the short duration of a power-up (e.g., 300 ms), but not after the untrusted external reader removes power.

It’s Groundhog Day! Again. Unawareness of time has left contactless payment cards vulnerable to a number of successful attacks (Table 1). For instance, Kasper et al. [35] recently demonstrated how to extract the 112-bit key from a MIFARE DESFire contactless smartcard (used by the Clipper all-in-one transit payment card¹). The side channel attack required approximately 10 queries/s for 7 hours. Some RFID credit cards are vulnerable to replay attacks because they lack a notion of time [21]. Oren and Shamir [34] show that power analysis attacks on UHF RFID tags can recover the password protecting a “kill” command with only 200 queries. At USENIX Security 2005, Bono et al. [8] implemented a brute-force attack against the Texas Instruments Digital Signature Transponder (DST) used in engine immobilizers and the ExxonMobile SpeedPassTM. The first stage of the attack required approximately 75,000 online “oracle” queries to

¹No relation to the Clipper Chip [27].

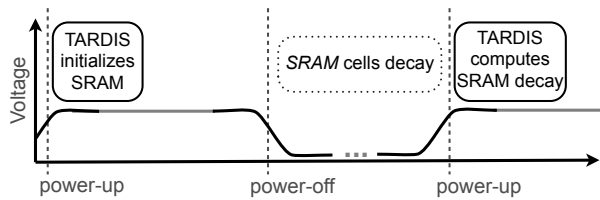


Figure 1: TARDIS estimates time by counting the number of SRAM cells that have a value of zero in power-up (*computes SRAM decay*). Initially, a portion of SRAM cells are set to one (*initializes SRAM*) and their values decay during power-off. The dots in the power-off indicate the arbitrary and unpredictable duration of power-off.

recover the proprietary cipher parameters [7].

A batteryless device could mitigate the risks of brute-force attacks, side-channel attacks, and reverse engineering by throttling its query response rate. However, the tag has no access to a trustworthy clock to implement throttling. A smartcard does not know whether the last interrogation was 5 seconds ago or 5 days ago.

Enter the TARDIS. To enable security protocols on intermittently powered devices without clocks, we propose Time and Remanence Decay in SRAM (TARDIS) to keep track of time without a power source and without additional circuitry. The TARDIS relies on the behavior of decaying SRAM circuits to estimate the duration of a power failure (Figure 1). Upon power-up, the TARDIS initializes a region in SRAM of an intermittently powered device. Later, during power-off, the SRAM starts to decay. Upon the next power-up, the TARDIS measures the fraction of SRAM cells that retain their state. In many ways, TARDIS operation resembles the functioning of an hourglass: the un-decayed, decaying, and fully decayed stages of SRAM are analogous to full, emptying, and empty hourglass states.

Contributions. Our primary contributions are:

- Algorithmic building blocks to demonstrate the feasibility of using SRAM for a trustworthy source of time without power.
- Empirical evaluation that characterizes the behavior of SRAM-based timekeeping under the effects of temperature, capacitance, and SRAM size.
- Enabling three security applications using SRAM-based TARDIS: sleepy RFID tags, squealing credit cards, and forgiving e-passports.

State of the Art. Today, batteryless devices often implement monotonically increasing counters as a proxy for timekeeping. RFID credit cards occasionally include transaction counters to defend against replay attacks. Yet

the counters introduce vulnerabilities for denial of service and are difficult to reset based on time elapsed; one credit card ceases to function after the counter rolls over [21]. While one can maintain a real-time clock (RTC) with a battery on low-power mobile devices [40], batteryless platforms do not support RTCs across power failures [31, 41, 9] because of the quiescent current draw.

While a timer of just a few seconds would suffice to increase the difficulty of brute-force attacks (Table 1), our experimental results indicate that an SRAM timer can reliably estimate the time of power failures from a few seconds up to several hours. For example, using a $100\ \mu F$ capacitor at room temperature, the TARDIS expiration time can exceed 2 hours of time. We evaluate the energy and time overhead of the TARDIS, its security against thermal and power-up attacks, and its precision across different platforms.

The primary novelty of the TARDIS is that a moderately simple software update can enable a sought-after security primitive on existing hardware without power. While data remanence is historically considered an undesirable security property [19], the TARDIS uses remanence to improve security. At the heart of the TARDIS are SRAM cells, which are among the most common building blocks of digital systems. The ubiquity of SRAM is due in part to ease of integration: in contrast with flash memory and DRAM, SRAM requires only a simple CMOS process and nominal supply voltage.

2 Intermittently Powered Devices: Background, Observations, and Challenges

New mobile applications with strict size and cost constraints, as well as recent advances in low-power microcontrollers, have given rise to a new class of intermittently powered device that is batteryless and operates purely on harvested energy. These devices—including contact and contactless smart cards and computational RFID tags (CRFIDs) [38, 41, 56, 55]—typically have limited computational power, rely on wireless transmissions from a reader both for energy and for timing information, and lose power frequently due to minimal energy storage. For example, when a contactless transit card is brought sufficiently close to a reader in a subway, the card gets enough energy to perform the requested tasks. As soon as the card is out of the reader range, it loses power and is unable to operate until presented to another reader. Since a tag loses power in the absence of a reader, it doesn't have any estimation of time between two interactions with a reader.

A typical secure communication between a reader and a tag is shown in Figure 2. The tag will only respond to the reader's request if it has authenticated itself by correctly answering the challenge sent by the tag. Two problems

	SRAM	DRAM
Purpose	Fast local memory	Large main memory
Location	Usually on-chip w/ CPU	Usually off-chip
Applications	CPU caches, microcontrollers	Desktop computers, notebooks, servers
Storage technology	Cross-coupled transistors	Capacitors
Normal operation	Constantly powered	Intermittently refreshed
Decay state	50% zero/one bits	All zero bits

Table 2: Because CPUs of embedded devices generally do not have on-chip DRAM, the TARDIS operates on SRAM. SRAM and DRAM differ fundamentally in their manufacture, operation, intended use, and state of decay.

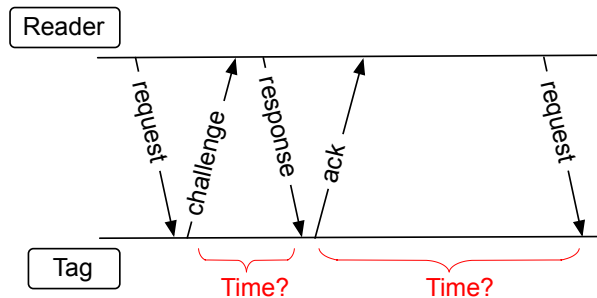


Figure 2: The tag cannot determine the time between a challenge and a response or the time between two sessions. The reader could respond to the tag as tardily as it likes or query the tag as quickly as it wants.

arise in this scheme:

- The tag is unaware of the amount of time spent by the reader to answer the challenge, so an adversary has an unlimited amount of time to crack a challenge.
- The tag is unaware of the time between two different queries, so an adversary can send a large number of queries to the tag in a short time space. This can make various brute-force attacks possible on these devices.

Traditionally, computing devices have either had a direct connection to a reliable power supply or large batteries that mask disconnections and maintain a constant supply of power to the circuit. In either case, a reliable sense of time can be provided using an internal clock. Time measurement errors, due to clock drift or power failures, can be corrected by synchronizing with a trusted peer or other networked time source. Current embedded systems address the timekeeping issue in one of the following ways:

1. A system can power a real-time clock (RTC); however, this is not practical on intermittently powered devices due to their tight energy budget. Even if the

system uses a low-power RTC (e.g., NXP PCF2123 RTC chip [32]), the RTC component has to be constantly powered (for example, using a battery). This choice also increases the cost of manufacturing and it does not benefit devices that are already deployed.

2. A system can keep time by accessing an external device (e.g., an RFID tag reader) or by secure time synchronization [14, 46]. This option introduces security concerns and may either require significant infrastructure or severely limit range and mobility.

2.1 Threat Model and Assumptions

“...if the attack surface includes an awful lot of clocks that you do not control, then it’s worth some effort to try and make your system not depend on them anymore.”—Ross Anderson [30]

The primary goal of the adversary in our model is to distort the TARDIS timekeeping. Our threat model considers semi-invasive attacks common to smart cards [15, 35]. We will not discuss attacks such as buffer overflows which are against the systems that would integrate the TARDIS; we focus on the attacks aimed at the TARDIS itself. Our adversarial model considers two classes of attacks: (1) thermal attacks that use heating and cooling [19] to distort the speed of memory decay; and (2) power-up attacks that keep the tag partially powered to prevent memory decay.

3 The TARDIS Algorithms

The TARDIS exploits SRAM decay during a power-off to estimate time. An example of the effect of time on SRAM decay in the absence of power is visualized in Figure 3. In this experiment, a 100×135 pixel bitmap image of a different TARDIS [1] was stored into the SRAM of a TI MSP430 microcontroller. The contents of the memory were read 150, 190, and 210 seconds after the power was disconnected. The degree of image distortion is a function of the duration of power failure.²

²The 14.6 KB image was too large to fit in memory, and therefore was divided into four pieces with the experiment repeated for each to

Figure 1 shows the general mechanism of the TARDIS. When a tag is powered up, the TARDIS initializes a region in SRAM cells to 1. Once the power is cut off, the SRAM cells decay and their value might reset from 1 to 0. The next time the tag is powered up, the TARDIS tracks the time elapsed after the power loss based on the percentage of cells remaining 1. Algorithm 1 gives more details about the implementation of the TARDIS.

MEASURE_TEMPERATURE: To detect and compensate for temperature changes that could affect the decay rate (Section 6), the TARDIS uses the on-board temperature sensor found on most microcontrollers. The procedure **MEASURE_TEMPERATURE** stores inside-the-chip temperature in the flash memory upon power-up. The procedure **DECAY** calls the **TEMPERATURE_ANALYZE** function to decide if the temperature changes are normal.

TIME: The TARDIS **TIME** procedure returns *time* and *decay*. The precision of the *time* returned can be derived from the *decay*. If the memory decay has not started (*decay* = 0), the procedure returns $\{time, 0\}$ meaning that the time duration is less than *time*. If the SRAM decay has started but has not finished yet ($0 \leq decay \leq 50\%$), the return value *time* is an estimate of the elapsed time based on the *decay*. If the SRAM decay has finished (*decay* $\simeq 50\%$), the return result is $\{time, 50\}$ meaning that the time elapsed is greater than *time*.

ESTIMATION: The procedure **ESTIMATE** uses a lookup table filled with entries of decay, temperature, and time stored in non-volatile memory. This table is computed based on a set of experiments on SRAM in different temperatures. Once the time is looked up based on the measured decay and the current temperature, the result is returned as *time* by the **ESTIMATE** procedure. The pre-compiled lookup table does not necessarily need to be calibrated for each chip as we have observed that chip-to-chip variation affects decay only negligibly (Section 6).

3.1 TARDIS Performance

The two most resource-consuming procedures of the TARDIS are **INIT** (initializing parts of the SRAM as well as measuring and storing the temperature) and **DECAY** (counting the zero bits and measuring the temperature). Table 3 shows that energy consumed in total by these two procedures is about $48.75 \mu J$ and it runs in $15.20 ms$.

Our experiments of time and energy measurements are performed on Moo RFID[56] sensor tags that use an MSP430F2618 microcontroller with 8 KB of memory, and a $10 \mu F$ capacitor. A tag is programmed to perform one of the procedures, and the start and end of the task is marked by toggling a GPIO pin. The tag's capacitor is

get the complete image. The microcontroller was tested in a circuit shown in Figure 6 with a $10 \mu F$ capacitor at $26^\circ C$. No block transfer computation was necessary.

Algorithm 1 TARDIS Implementation

```

INIT(addr, size)
1  for  $i \leftarrow 1$  to size
2      do  $memory(addr + i - 1) \leftarrow 0xFF$ 
3   $temperature \leftarrow MEASURE\_TEMPERATURE()$ 

DECAY(addr, size)
1   $decay \leftarrow COUNT0S(addr, size)$ 
2   $\succ$  Proc. COUNT0S counts the number of 0s in a byte.
3  if TEMPERATURE_ANALYZE(temperature)
4   $\succ$  This procedure decides if the temperature changes
    are expected considering the history of temperature
    values stored in flash memory.
5  then return decay
6  else return error

EXPIRED(addr, size)
1   $\succ$  Checks whether SRAM decay has finished.
2   $decay \leftarrow DECAY(addr, size)$ 
3  if ( $decay \geq \%50 \times 8 \times size$ )
4  then return true
5  else return false

TIME(addr, size, temperature)
1   $\succ$  Estimate the passage of time by comparing the
    percentage of decayed bits to a precompiled table.
2   $decay \leftarrow DECAY(addr, size) / (8 \times size)$ 
3   $time \leftarrow ESTIMATE(decay, temperature)$ 
4  return  $\{time, decay\}$ 

```

charged up to $4.5 V$ using a DC power supply and then disconnected from the power supply so that the capacitor is the only power source for the tag. In the experiments, the DC power supply is used instead of an RF energy supply because it is difficult to disconnect the power harvesting at a precise capacitor voltage. We measured the voltage drop of the capacitor and the GPIO pin toggling using an oscilloscope. The energy consumption of the task is the difference of energy ($\frac{1}{2} \times CV^2$) at the start and end of the task. The reported measurement is the average of ten trials.

4 Securing Protocols with the TARDIS

There are many cases where the security of real-world applications has been broken because the adversary could query the device as many times as required for attack. Table 1 gives a summary of today's practical attacks on intermittently powered devices. By integrating the TARDIS, these applications could throttle their response rates and

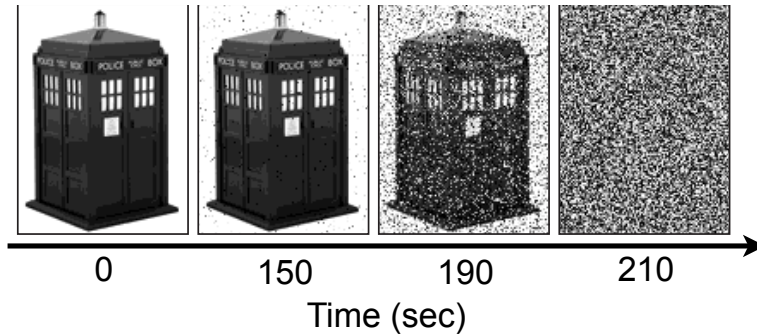


Figure 3: Programs without access to a trustworthy clock can determine time elapsed during a power failure by observing the contents of uninitialized SRAM. These bitmap images of the TARDIS [1] represent four separate trials of storing the bitmap in SRAM, creating an open circuit across the voltage supply for the specified time at 26°C , then immediately returning a normal voltage supply and reading uninitialized SRAM upon reboot. The architecture of a contactless card is modeled using a $10\ \mu\text{F}$ capacitor and a diode in series with the MSP430 microcontroller’s voltage supply pin. The degree of decay is a function of the duration of power failure, enabling hourglass-like timekeeping precision without power. No TARDIS was harmed or dematerialized in this experiment.

Procedure	Energy Cost	Exec. Time
INIT	$11.53\ \mu\text{J} \pm 2.47$	$2.80\ \text{ms} \pm 0.0\bar{0}$
DECAY	$37.22\ \mu\text{J} \pm 9.31$	$12.40\ \text{ms} \pm 1.10$

Table 3: Overhead of TARDIS INIT and DECAY procedures measured for TARDIS *size* of 256 bytes.

improve their security.

We discuss six security protocols that could strengthen their defense against brute-force attacks by using the TARDIS. To demonstrate the ease of integrating the TARDIS, we have implemented and tested three of these security protocols on the Moo, a batteryless microcontroller-based RFID tag with sensors but without a clock [56]. Our prototypes demonstrate the feasibility of the TARDIS and its capabilities in practice.

Sleepy RFID Tags: To preserve the users privacy and prevent traceability, one could use a “kill” command to permanently deactivate RFID tags on purchased items [25]. However, killing a tag disables many features that a customer could benefit from after purchase. For example, smart home appliances (e.g., refrigerators or washing machines) may no longer interact with related items even though they have RFID tags in them. One could temporarily deactivate RFID tags by putting them to “sleep.” However, lack of a simple and practical method to wake up the tags has made this solution inconvenient [25]. By providing a secure notion of time, the TARDIS makes it possible to implement *sleepy tags* that can sleep temporarily without requiring additional key PINs or cryptographic

solutions. We consider a time resolution on the order of hours more appropriate for this application.

To extend the sleep time of sleepy tags, one could use a counter along with the TARDIS as follows: upon power-up, the tag checks the TARDIS timer, and it does not respond to the reader if the timer has not expired. If the TARDIS timer has expired, the tag decreases the counter by one and initializes the TARDIS again. This loop will continue while the counter is not zero. For example, using a counter initially set to 1000 and a TARDIS resolution time of 10 seconds, the tag could maintain more than 2 hours of delay. Since the tag exhausts its counter every time it wakes up, the reader interacting with the tag has to query the tag intermittently.

The TARDIS could prevent yet another attack on Electronic Product Code (EPC) tags that use “kill” commands. To prevent accidental deactivation of tags, a reader must issue the right PIN to kill a tag [12]. An adversary could brute-force the PIN (32 bits for EPC Class1 Gen2 tags). The TARDIS enables the RFID tag to slow down the unauthorized killing of a tag by increasing the delay between queries and responses.

Squealing Credit Cards: Today, a consumer cannot determine if her card has been used more than once in a short period of time unless she receives a receipt. This is because a card cannot determine the time elapsed between two reads as the card is powered on only when it communicates with the reader. The TARDIS enables a “time lock” on the card such that additional reads would be noticed. Thus a consumer could have some assurance that after exposing a card to make a purchase, an accidental second read or an adversary trying to trick the card into

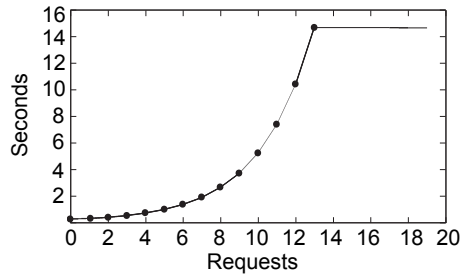


Figure 4: Measured response time of a 2010-issued French passport [5]. The passport imposes up to 14 seconds of delay on its responses after unsuccessful execution. The delay will remain until a correct reading happens even if the passport were removed from the reader’s field for a long time.

responding would be revealed. *Squealing credit cards* would work similarly to today’s credit cards, but they are empowered by the TARDIS to estimate the time between queries and warn the user audibly (a cloister bell) if a second read is issued to the card too quickly. A time lock of about one minute can be considered enough for these applications.

Forgiving E-passports: RFID tags are used in e-passports to store holder’s data such as name, date of birth, biometric ID, and a unique chip ID number. E-passports are protected with techniques such as the Basic Access Control (BAC) protocol, shielding, and passive authentication. However, in practice, e-passports are not fully protected. An adversary can brute-force the BAC key in real time by querying the passport 400 times per minute for a few weeks [6]. Another attack can accurately trace a specific passport by sending hundreds of queries per minute [11].

To mitigate the effect of brute-force attacks, French e-passports have implemented a delay mechanism—we imagine using a counter—to throttle the read rate [5]. This delay increases to 14 seconds after 14 unsuccessful attempts (Figure 4) and would occur even if the passport was removed from the RF field for several days. Once the tag is presented with an authorized reader, the delay will be enforced and then reset to zero. The TARDIS provides a time-aware alternative that delays unauthorized access but ignores the previous false authentication attempts if the passport has been removed from the reader’s range for an appropriate duration. A time duration matching the maximum implemented delay (14 seconds for French passports) would be enough to implement this function.

Passback - Double-tap Prevention: In mass transportation and other similar card entry systems, the goal of the

operator is to prevent multiple people from accessing the system simultaneously using the same card. To achieve this goal, systems are typically connected to a central database that prevents a card from being used twice in a short time frame.³ Using the TARDIS, a card could implement delay before permitting re-entry rather than requiring the system to check a central database.

Resurrecting Duckling: Secure communication in ad-hoc wireless networks faces many obstacles because of the low computing power and scarce energy resources of these devices. Stajano et al. [45] proposed a policy in which these devices would transiently accept a new owner. The devices will later return to an unprogrammed status when the owner no longer needs them, they receive a kill command, or another predefined reset condition is met. Later, others can reclaim and reuse these devices.

For wirelessly powered devices, the TARDIS can provide a sense of time, allowing them to be “reborn” with a new owner only if there is an extended power outage. A legitimate user can continue to power the device wirelessly, but if she wishes to transfer ownership to another entity, she must power it down for a long enough time (defined by the user). Otherwise, the RFID tag refuses to interact with anyone not possessing the present cryptographic key. An example of this application is secure pairing for computational contact lenses [22]. The controller could be cryptographically bound until power disappears for more than a few minutes. Another use of this application is to make stealing SIM cards difficult [16]. The card could refuse to boot if it has been unpowered for a fair amount of time.

Time-out in Authentication Protocols: Because RFID tags rely on a reader as their source of energy, they cannot measure the delay between a request to the reader and its corresponding response. The tag ignorance gives the reader virtually unlimited time to process the request and response in an authentication algorithm. Having unlimited response time enables the adversary to employ various attacks on the request message with the goal of breaking it. Using the TARDIS will limit the adversary time frame for a successful attack. An example of these protocols can be seen in the e-passport BAC protocol where the reader and passport create a session key for communication. Using The TARDIS would enable passports to enforce expiration of these keys.

4.1 Implementation and Evaluation

For the implementation of *sleepy tags*, *squealing credit cards*, and *forgiving e-passports*, we have chosen the Moo, a batteryless microcontroller-based RFID tag. We have

³Houston METRO system: <http://www.ridemetro.org/fareinfo/default.aspx>

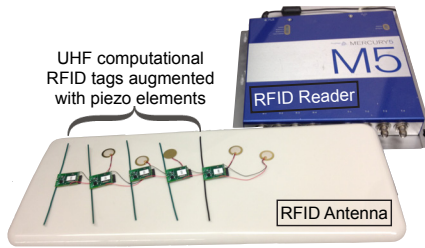


Figure 5: Our applications are implemented and tested on the Moo RFID sensors and are remotely powered by a RFID reader (ThingMagic M5 [51]).

Algorithm 2 An example of TARDIS usage in a protocol.

```

TARDIS_EXAMPLE(addr, size)
1  if EXPIRED(addr, size)
2    then RESPOND_TO_READER()
3     INIT(addr, size)
4  else BUZZ_PIEZO_ELEMENT()

```

augmented this tag with a piezo-element [20] so that it can audibly alert the user to events.

Implementation: We have implemented a TARDIS library that provides the procedures INIT and EXPIRE listed in Algorithm 1. For the three implemented protocols, a 1-bit precision of time—whether or not the timer had expired—was enough. The programs used for all three protocols are similar and are shown in Algorithm 2. The tag was programmed to call the EXPIRE procedure upon power-up; if the timer had expired, it would respond to the reader and call INIT; otherwise, the tag would buzz its piezo-element. In the case of the *squealing credit cards* protocol the tag was programmed to respond to the reader after buzzing, but for the two other applications, the tag stopped communicating with the reader.

We used a ThingMagic reader [51] and its corresponding antenna to query the tag. When the tag was queried for the first time upon removal from the RF field, it buzzed. The tag stayed quiet whenever it was queried constantly or too quickly.

Experimental Setup: To measure the TARDIS resolution time on this platform, we powered up the tag to 3.0 V using an external power supply and then disconnected it. We observed the voltage drop over time on an oscilloscope and measured the elapsed time between loss of power and when SRAM decay has finished.⁴ We conducted our experiments on five tags, which use a 10 μF capacitor as its

⁴Our experiments (Section 6) have shown that SRAM decay finishes when the tag voltage reaches 50 mV.

primary power source. The TARDIS resolution time on average was 12.03 seconds with a standard deviation of 0.11 seconds. A similar tag, which uses 100 mF, yields a TARDIS resolution time of 145.85 seconds. These time measurements are specific to the platform we have chosen for our experiment. The resolution could potentially be extended to hours using additional capacitors (Table 5).

5 Security Analysis

Depending on the application, the adversary may wish either to slow down or to speed up the expiration of the TARDIS. We discuss four different attacks that try to distort the TARDIS interpretation of time.

Cooling Attacks. An adversary might try to reduce the system’s temperature, aiming to slow down the memory decay rate. Other works [19] have used this technique to prevent data decay in DRAM for the purpose of data extraction. Cooling attacks might target the TARDIS timer in cases where the adversary needs to slow the passage of time. As explained in Algorithm 1, the TARDIS measures and records a device’s temperature over time and therefore it can prevent cooling attacks by observing unexpected temperature changes.

Heating Attacks. In contrast to cooling attacks, an attacker might need to speed up the TARDIS timer. For example, someone might try to decrease the delay between queries in order to speed up brute-force attacks. Similarly to the defense against cooling attacks, the TARDIS will report an error indicating unexpected temperature changes.

Pulse Attacks. A more sophisticated attack is a combination of the cooling and heating attacks such that the temperature would remain the same in the beginning and the end of the attack. It should be noted that this is not a trivial attack because the adversary needs to restore the original internal temperature to prevent the thermal sensor from noticing any difference. A defense against pulse attacks is to implement a thermal fuse [10] on the chip that will activate when the chip is exposed to a high temperature. The activation of this fuse will then either notify the TARDIS of temperature tampering on the next boot-up or possibly prevent the system from booting up at all.

Voltage Control Attack. Another possible attack scenario would be to power up the system wirelessly to a minimum voltage that is not sufficient for booting up but sufficient for stopping the memory decay. This would prevent the device from noticing the unauthorized reader and it would stop the memory from decaying further (see Figure 8). The voltage control attack can freeze the TARDIS timer at a specific time as long as it sustains the power sup-

ply. We imagine that this attack is difficult to implement because of the inherent design of the readers. Many factors (e.g., distance) affect the voltage received by the tags and tags are very sensitive to environmental effects. The readers are also generally designed to flood the targeted environment with energy to provide the tags in range with more than the maximum required power [54]. Excessive power that may have been generated by these devices is then filtered out in tags using voltage regulators. To implement this attack, we imagine the adversary would need to control the input voltage to the tag with a very high precision. If the tag voltage for any reason drops, the SRAM will decay irreversibly. At the same time, the adversary would need to prevent the tags from fully powering up and noticing the unauthorized reader.

6 Factors Affecting SRAM Decay

In our evaluation of the TARDIS, we examine the decay behavior of SRAM and three factors that have major effects on this behavior. All experiments use the same circuit (Figure 6), and follow the same general procedure.

Experimental Setup: A microcontroller runs a program that sets all available memory bits to 1. The power is then effectively disconnected for a fixed amount of time (*off-time*). When power is reapplied to the chip, the program records the percentage of remaining 1-bits to measure memory decay, and then it resets all bits to 1 in preparation for the next time power is disconnected. A Data Acquisition (DAQ) unit from Agilent (U2541A series) precisely controls the timing of power-ups and power-downs between 3 and 0 Volts, and also measures the voltage across the microcontroller throughout the experiment. An inline diode between the power supply and microcontroller models the diode at the output of the power harvesting circuit in RFIDs; it also prevents the DAQ from grounding VCC during the off-time when the DAQ is still physically connected but is not supplying power. In all experiments, microcontrollers from the TI MSP430 family are used to ensure maximum consistency. The microcontroller used in all experiments is MSP430F2131 with 256 B of SRAM unless stated otherwise.

In all of the experiments, temperature is controlled by conducting all tests inside of a Sun Electronics EC12 Environmental Chamber [47] capable of creating a thermally stable environment from -184°C to $+315^{\circ}\text{C}$ with 0.5°C precision. We use an OSXL450 infrared non-contact thermometer [33] with $\pm 2^{\circ}\text{C}$ accuracy to verify that our microcontroller has reached thermal equilibrium within the chamber before testing. For all the experiments, we have collected at least 10 trials.

Defining Stages of Decay: Three distinct stages of decay are observed in all experiments. Figure 7 illus-

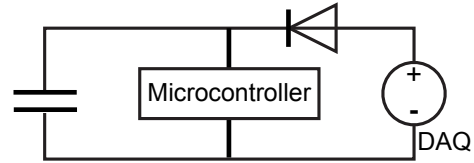


Figure 6: General circuit used during the experiments. The microcontroller is held in an environmental chamber to ensure consistent temperature during the tests. The Data Acquisition (DAQ) unit both provides power to the microcontroller and records the voltage decay.

Term	Definition
SRAM Decay	Change of value in SRAM cells because of power outage
Decay Stage 1	Time before the first SRAM cell decays
Decay Stage 2	Time between the decay of first SRAM cell and last one
Decay Stage 3	Time after the last SRAM cell decays
Ground State	The state that will be observed in an SRAM cell upon power-up, after a very long time without power
DRV	Data Retention Voltage, minimum voltage at which each cell can store a datum
DRV Probabil- ity(v)	Probability that a randomly chosen cell will have a DRV equal to v and a written state that is opposite its ground state.

Table 4: Definition of the terms used to explain the behavior of SRAM decay and the theory behind it.

trates the three stages of SRAM decay measured on a TI MSP430F2131 with 256 B of SRAM and a $10\ \mu\text{F}$ capacitor, at 26°C . We vary the *off-time* from 0 to 400 seconds in 20-second increments. In the first stage, no memory cells have decayed; during the second stage, a fraction of the cells, but not all, have decayed; by the third stage the cells have decayed completely (see Table 4 for a summary of term definitions). Observations made during Stages 1 or 3 provide a single bit of coarse information, indicating only that Stage 2 has not yet begun or else that Stage 2 has already been completed. Observations made during Stage 2 can provide a more accurate notion of time based on the percentage of decayed bits.

Decay vs. Voltage: The decay rate of SRAM is expected to depend only on its voltage level (Section 7). Temperature, SRAM size, and circuit capacitance all affect the rate of voltage depletion and thus only have secondary effects on memory decay. Our experimental results (Figure 8) for five sets of tests (each at least 10 trials) support this hypothesis. The same setup as explained before was

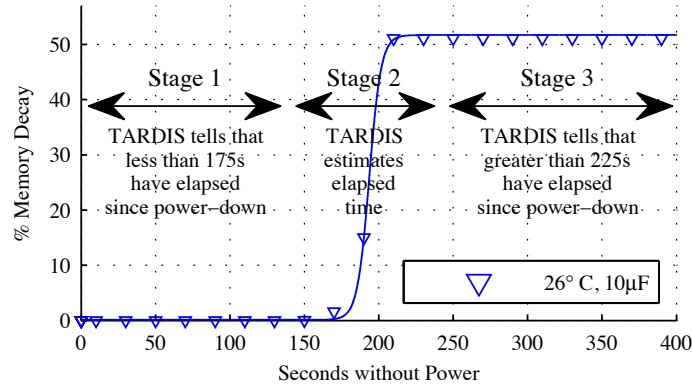


Figure 7: The TARDIS presents a three-stage response pattern according to its amount of decay. Before 175 seconds, the percentage of bits that retain their 1-value across a power-off is 100%. For times exceeding 225 seconds, the TARDIS memory has fully decayed. The decay of memory cells between these two thresholds can provide us with a more accurate measurement of time during that period. This graph presents our results measured on a TI MSP430F2131 with 256 B of SRAM and a 10 μF capacitor at 26°C.

used and five different temperatures (one with a 10 mF capacitor and four of them without) were tested.

Impact of Temperature: The work of Skorobogatov [44] shows that low temperature can increase the remanence time of SRAM, and the work of Halderman et al. [19] similarly shows that low temperature can extend the remanence time of DRAM. For the TARDIS using SRAM decay to provide a notion of time, the interesting question is the opposite case of whether high temperature can decrease remanence. We use the same experimental setup as before (without using capacitors) to investigate how decay time varies across five different elevated temperatures (in the range of 28°C – 50°C). The off-time of the microcontroller varied from 0 to a maximum of 5 seconds. Figure 9 shows that the decay time is non-zero across all temperatures. This indicates that the TARDIS could work at various temperatures as long as changes in the temperature are compensated for. For the TARDIS, this compensation is done by using temperature sensors which are available in many of the today’s microcontrollers.⁵

Impact of Additional Capacitance: Capacitors can greatly extend the resolution time of the TARDIS. In our experiment, we have tested five different capacitors ranging from 10 μF to 10 mF at 26.5°C. For this experiment, the capacitors were fully charged in the circuit and their voltage decay traces were recorded. These traces were later used in conjunction with our previous remanence-vs.-decay results (Section 6) to calculate the time frame

⁵According to the TI website, 37% of their microcontrollers are equipped with temperature sensors.

Cap. Size	Stage 1 (s)	Stage 2 (s)
0 μF	1.22e0	8.80e-1
10 μF	1.75e2	5.00e1
100 μF	1.13e3	8.47e2
1000 μF	1.17e4	9.50e3
10000 μF	1.43e5	>5.34e4*

* Test was interrupted.

Table 5: Estimated time in Stage 1 and Stage 2 of the TARDIS increases as capacitor size increases. The experiments are done on a MSP430F2131 microcontroller at 26.5°C and an SRAM size of 256 B. Stage 1 is the time after the power failure but before the SRAM decay. Stage 2 represents the duration of SRAM decay.

achievable with each capacitor. Table 5 summarizes the results for the duration of TARDIS Stage 1 and 2 based on capacitor size. The voltage decay traces, our conversion function (DRV Prob.), and the resulting SRAM-decay-over-time graph can be seen in Figure 10.

Results ranging from seconds to days open the path for a wide variety of applications for the TARDIS, as it can now be tweaked to work in a specific time frame. Current RFID-scale devices generally use capacitors ranging from tens of picofarads to tens of microfarads (e.g., [2] [3]). Although a 10 mF capacitor size might be large compared to the size of today’s transiently powered devices, the progress in capacitors’ size and capacity may very well make their use possible in the near future.

Impact of SRAM Size: Our hypothesis is that SRAM size has an inverse relation with decay time. This is ex-

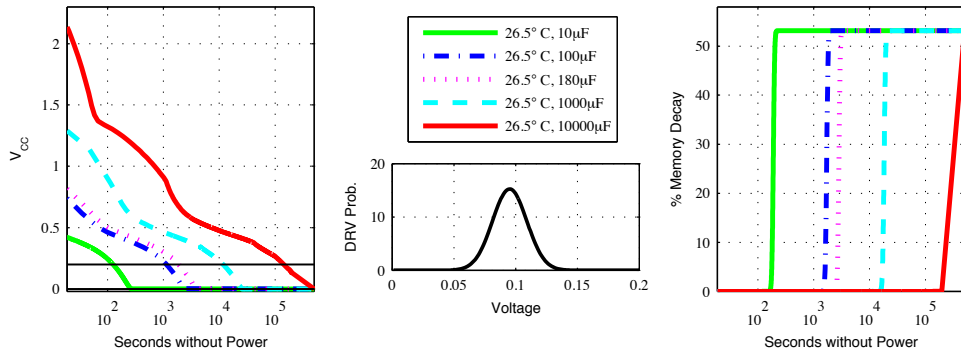


Figure 10: For five different capacitor values, measured supply voltage traces are combined with a pre-characterized DRV distribution to predict decay as a function of time. The decaying supply voltages after power is turned off are shown at left. The known DRV probabilities (Equation 4) for $26.5^{\circ}C$ are shown at center. Equation 5 maps every supply voltage measurement to a predicted decay, thus creating the memory-decay-vs.-time plots shown at right. The two horizontal lines in the left image at approximately 150 and 50 mV are the voltages where the first and last bits of SRAM will respectively decay.

pected because a larger SRAM will have a larger leakage current and thus will drain the capacitor more quickly. We tested three different models of MSP430 microcontroller with SRAM sizes of 256 B, 2 KB, and 8 KB at $28^{\circ}C$ with no capacitor. The DAQ sweeps off-time from 0 to a maximum of 5 seconds. The experiment results are consistent with our hypothesis and are shown in Figure 11. It should be noted that SRAM size is not the only difference between these three models, as they also have slightly different power consumptions.

Impact of Chip Variation: The chip-to-chip variation of the same microcontroller model is not expected to have a major effect on the TARDIS. We tested three instances of the MSP430F2131 with 256 B of memory and no capacitor at $27^{\circ}C$. The off-time changes from 0 to a maximum of 2.5 seconds with increments of 0.2 seconds. The result shown in Figure 12 matches our expectation and shows that changes in decay time due to chip-to-chip variation are insignificant (notice that no capacitor is used and the temperature for one of the chips is one degree higher). This result indicates that TARDIS would work consistently across different chips of the same platform and can be implemented on a system without concern for chip-to-chip variation.

TARDIS Simulation: We verified the TARDIS mechanism using SPICE simulation of a small SRAM array of 50 cells; the transistor models are 65 nm PTM, the power pin is connected to V_{CC} through a D1N4148 diode, and the decoupling capacitor is 70 nF. Each transistor is assigned a random threshold voltage deviation chosen uniformly from range ± 100 mV. Each line in Figure 13 plots the voltage difference across the two state nodes A and B for a single SRAM cell. Because all state nodes remain be-

tween 0V and V_{CC} during the discharge, the differential voltage is roughly enveloped by $\pm V_{CC}$ as shaded in grey. A positive differential voltage indicates a stored state of 1 (the written state), and a negative differential is a state of 0. Some of the nodes are observed to flip state, starting when V_{CC} reaches 200 mV at 0.55 seconds after power is disconnected. As V_{CC} discharges further, more cells decay by crossing from state 1 to 0. When V_{CC} is powered again at 1.05 seconds, each cell locks into its current state by fully charging either A or B and discharging the other; this is observed in Figure 13 as an increase in the magnitude of the differential voltage of each cell.

7 Inside an SRAM Cell

Each SRAM cell holds state using two cross-coupled inverters as shown in Figure 14; the access transistors that control reading and writing to the cell are omitted from the figure. The cross-coupled inverters are powered via connections to the chip's power supply node. The two states of the SRAM cell, representing a logical 1 and logical 0, are symmetrical. In each state, under normal conditions, the voltage of either A or B is approximately V_{cc} while the voltage of the other is approximately 0V.

Data Retention Voltage: The minimum voltage at which each cell can store either a 0 or 1 is referred to as the cell's data retention voltage (DRV) [36]. Since DRV depends on random process variation, any set of SRAM cells will have a distribution of DRVs. Although the actual DRV distribution depends on process and design parameters, typical values fall within the range of 50 mV to 250 mV; a published design in $0.13 \mu m$ has a distribution of DRVs ranging from 80 mV to 250 mV, and our own analysis in

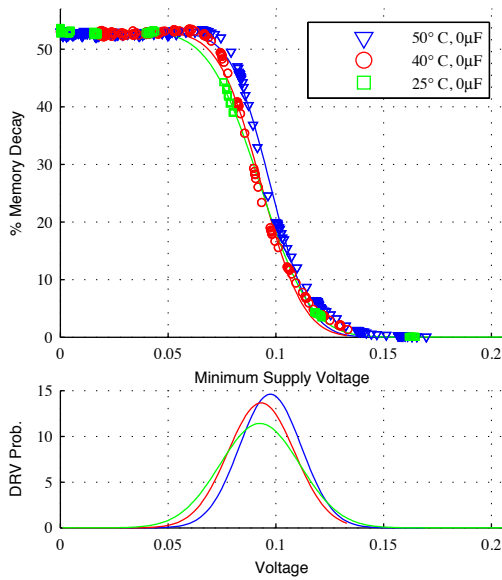


Figure 8: Regardless of temperature, the amount of decay depends almost entirely on the minimum supply voltage reached during a power-down. The bottom graph shows the 3-parameter DRV probabilities (Equation 4) that best predict the observed relationships between decay and minimum supply voltage for each of the three temperatures. The fit lines in the upper graph show the relationships between decay and minimum supply voltage that are predicted by these DRV models (Section 10).

this work estimates a majority of DRVs to be in the range of 50 mV to 160 mV (Figure 8).

7.1 Memory Decay Mechanisms

Memory decay occurs in SRAM when a cell loses its state during a power cycle and subsequently initializes to the opposite state upon restoration of power. Given that each cell typically favors one power-up state over the other [23, 17], memory decay can be observed only when the last-written state opposes the favored power-up state. We denote the favored power-up state as the *ground state*, since this is the value an SRAM cell will take at power-up after a very long time without power. We say that a cell written with the value opposite its ground state is *eligible* for memory decay. Each eligible cell will decay once the supply voltage falls below the cell’s DRV. Cells that are randomly assigned very low DRVs thus do not decay until the supply voltage is very low. With sufficient capacitance, it can take days for all eligible cells to decay.

Supply voltage decays according to Equation 1, where V_{CC} , I_{CC} , and C_{CC} represent the supply voltage, current,

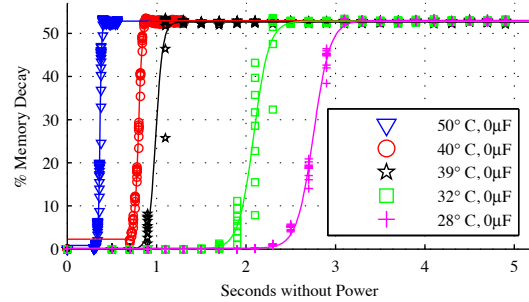


Figure 9: The duration of SRAM decay is non-zero across all temperatures even when no capacitor is used. For any given temperature, the duration of SRAM decay is consistent across trials. Increasing the temperature from 28°C to 50°C reduces the duration of both Stage 1 and Stage 2 decay by approximately 80%.

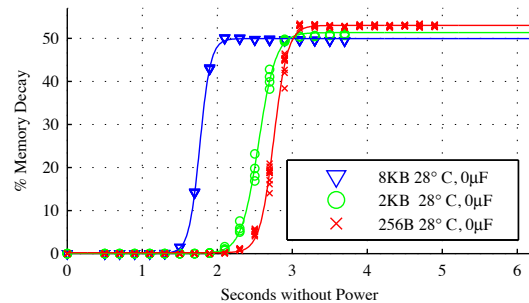


Figure 11: Different microcontrollers within the TI MSP430 family with different SRAM sizes exhibit different decay times, but follow the same general trend. The MSP430F2618, MSP430F169, and MSP430F2131 respectively have 8 KB, 2 KB, and 256 B of SRAM.

and capacitance of the power supply node. The voltage decay is slowed by a large capacitance and low current, and the following paragraphs explain why both are present in our TARDIS application.

$$\frac{dv_{CC}}{dt} = \frac{I_{CC}}{C_{CC}} \tag{1}$$

Large Capacitance: The large amount of charge stored on the power supply node is due to the decoupling capacitance that designers add between V_{CC} and gnd . During normal operation, this capacitance serves to stabilize the supply voltage to the functional blocks of the chip, including SRAM. In some experiments, the time ranges measurable by the TARDIS are further extended by supplementing the standard decoupling capacitors with additional explicit capacitance.

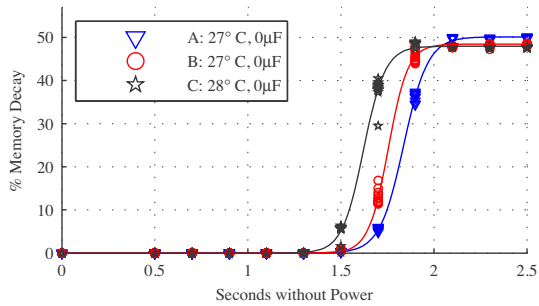


Figure 12: Decay versus time in 3 different instances of the MSP430F2131 microcontroller at similar temperatures. The durations of Stage 1 and Stage 2 decay match closely across instances.

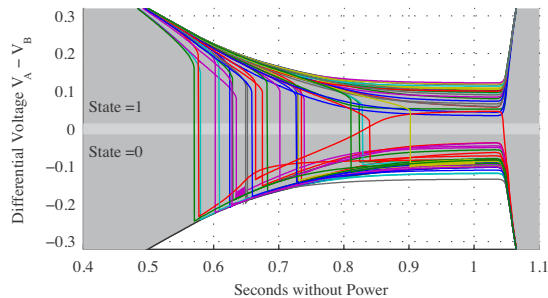


Figure 13: The differential voltage of SRAM cells during decay. The envelope of $\pm V_{CC}$ is shaded in grey. All cells are in the 1 state when power is first turned off. As V_{CC} decays, some cells flip from 1 to 0. The cells stabilize when power is restored. The number of zeros after the restoration of power is used to estimate the duration of the power outage.

Low Leakage Current: The total current I_{CC} comprises the operating current of the microcontroller and the SRAM's data-retention current; both currents are functions of the supply voltage. The current during the voltage decay is shown in Figure 15, and explained here:

Immediately after power is disconnected, supply voltages are above 1.4 V and the microcontroller is operational. The observed current is between 250 μA and 350 μA , consistent with the 250 μA current specified for the lowest-power operating point (1.8 V with 1 MHz clock) of the MSP430F2131 [50]. The SRAM current is negligible by comparison. The high current consumption causes the voltage to decay quickly while the microcontroller remains active.

As the voltage drops below 1.4 V, the microcontroller deactivates and kills all clocks to enter an ultra-low power RAM-retention mode in an attempt to avoid losing data.

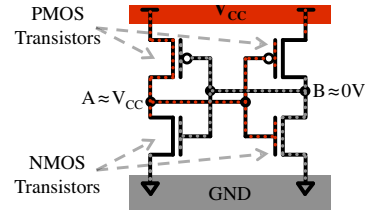


Figure 14: The state-holding portion of an SRAM cell consists of two cross-coupled inverters tied to the chip's power and ground nodes.

The nominal current consumed in this mode is only the data-retention current, specified to be 0.1 μA for the 256 B of SRAM in the MSP430F2131 [50]. In our observations, I_{CC} is between 0.5 μA and 10 μA during the time that V_{CC} is between 0.5 V and 1.4 V. This current is 1.5 – 3 orders of magnitude smaller than the current when the microcontroller is active. With so little current being consumed, the supply voltage decays very slowly. The current further decreases as the supply voltage drops into subthreshold, and cells begin to experience memory decay.⁶

Impact of Temperature: Increasing the temperature leads to more rapid memory decay for two reasons. First, increasing the temperature increases the leakage currents that persist through data-retention mode. Increased leakage currents lead to a faster supply voltage decay, causing the supply voltage to drop below DRVs sooner. Second, temperature expedites memory decay by increasing the DRV of SRAM cells [36], causing them to decay at slightly higher supply voltages. Prior work shows a modest 13mV increase in DRV when temperature increases from 27°C to 100°C [36].

7.2 Choosing a State to Write

It is possible to increase the maximum observable memory decay by making every cell eligible for decay. This would be accomplished by characterizing the ground state of each SRAM cell over many remanence-free trials [17, 23], and then writing each cell with its non-ground state in order to make its memory decay observable. In contrast to writing a uniform 1 to all cells, this approach can extract more timing information from the same collection of SRAM cells. However, this alternative requires storing the ground states in non-volatile memory (or equivalently storing written states in non-volatile memory) in order to

⁶Note that setting V_{CC} to 0 V during the power-down, instead of leaving it floating, reduces voltage and memory decay times by at least an order of magnitude [44] by providing a low impedance leakage path to rapidly drain the capacitance; we have observed this same result in our experiments as well.

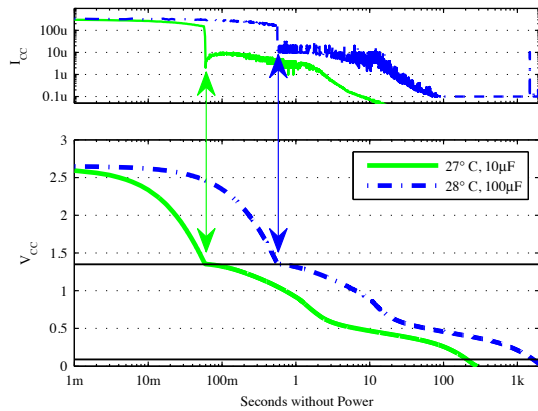


Figure 15: Supply voltage and current during two power-down events with different capacitors. The voltage V_{CC} is measured directly, and the current I_{CC} is calculated per Equation 1 using the measured $\frac{dV_{CC}}{dt}$ and known capacitor values. The voltage initially decays rapidly due to the high current draw of the microcontroller. When V_{CC} reaches 1.40V the microcontroller turns off and I_{CC} drops by several orders of magnitude, leading to a long and slow voltage decay. At the time when V_{CC} crosses the horizontal line at 0.09V, approximately half of all eligible cells will have decayed.

evaluate whether or not a cell has decayed. Our approach of writing a uniform 1 to all cells makes it possible to evaluate memory decay without this overhead simply by evaluating the Hamming Weight of the SRAM state.

8 Alternative Approaches

The more general question of how to keep time without a power source is fundamental and has numerous applications in security and real-time computing. Techniques for keeping time without power or with very reduced power typically rely on physical processes with very long time constants. In CMOS, the most obvious process with a long time constant is the leakage of charge off of a large capacitor through a reverse-biased diode or MOSFET in the cut-off region.

An unexplored alternative to the TARDIS is charging a capacitor whenever the device is active, and checking the capacitor’s voltage at a subsequent power-up to determine whether the device has been active recently. The power-up measurement can be performed using an ADC if available, or else by checking whether or not the remaining voltage is sufficient to register as a logical 1. This approach differs from the TARDIS in incurring monetary and power costs due to the use of a dedicated capacitor and dedi-

cated input-output pins for charging the capacitor and sensing its voltage. Furthermore, the capacitor voltage is still dynamic after power-up, leaving the measurement sensitive to timing variations caused by interrupts. By comparison, the TARDIS uses no dedicated capacitor or input-output pins; its measurement materializes in SRAM at power-up and remains static thereafter until being read and subsequently overwritten.

The EPC Gen2 protocol [12] requires UHF RFID tags to maintain four floating-gate based “inventorial flags” used to support short power gaps without losing the selected/inventoried status. An interesting alternative approach could co-opt these flags to provide a notion of time; however, the flags only persist between 500ms and 5s across power failures. In comparison, the SRAM-based approach in the TARDIS has a resolution time from seconds to hours and has a temperature compensation mechanism. Another advantage of the TARDIS is that it works on any SRAM-based device regardless of the existence of special circuits to support inventorial flags.

9 Related Work

RFID Security and Privacy: The inability of intermittently powered devices to control their response rates has made them susceptible to various attacks. An RFID tag could be easily “killed” by exhausting all possible 32-bit “kill” keys. Such unsafe “kill” commands could be replaced with a “sleep” command [25]; however, lack of a timer to wake up the tag in time has made the use of the “sleep” command inconvenient. The key to e-passports can be discovered in real time by brute-force attacks [6]. The attack could be slowed down if the e-passport had a trustworthy notion of time. The minimalist model [24] offered for RFID tags assumes a scheme that enforces a low query-response rate. This model could be implemented using the TARDIS.

Secure Timers: To acquire a trustworthy notion of time, multiple sources of time can be used to increase the security level of a timer [40]; but this requires the device to interact actively with more than one source of time, which is not practical for RFID tags that use passive radio communication. The same issues prevent us from using the Lamport clock and other similar mechanisms that provide order in distributed systems [26]. This inability to acquire secure time precludes the use of many cryptographic protocols, including timed-release cryptography [29] [39]

Ultra-low Power Clocks: With the rise of pervasive computing come a need for low-power clocks and counters. Two example applications for low-power clocks are timestamping secure transactions and controlling when a device

should wake from a sleep state. The lack of a rechargeable power source in some pervasive platforms requires ultra-low power consumption. Low voltage and subthreshold designs have been used to minimize power consumption of digital circuits since the 1970s [48]. Circuits in wrist-watches combine analog components and small digital designs to operate at hundreds of nW [53]. A counter designed for smart cards uses adiabatic logic to operate at 14KHz while consuming 11nW of power [49]. A gate-leakage-based oscillator implements a temperature-invariant clock that operates at sub-Hz frequencies while consuming 1pW at 300mV [28]. A TI-recommended technique [37] for the MSP430 is to charge a dedicated external capacitor from the microcontroller while in a low-power sleep mode with clocks deactivated; the microcontroller is triggered to wake up when the capacitor voltage surpasses a threshold. But all of these solutions, while very low-power, still require a constant supply voltage and hence a power source in the form of a battery or a persistently charged storage capacitor. However, embedded systems without reliable power and exotic low-power timers may still benefit from the ability to estimate time elapsed since power-down.

Attacks Based on Memory Remanence: Processes with long time constants can also raise security concerns by allowing data to be read from supposedly erased memory cells. Drowsy caches [13] provide a good background on the electrical aspects of data retention. Gutmann stated that older SRAM cells can retain stored state for days without power [18]. Gutmann also suggest exposing the device to higher temperatures to decrease the retention time. Anderson and Kuhn first proposed attacks based on low-temperature SRAM data remanence [4]. Experimental data demonstrating low-temperature data remanence on a variety of SRAMs is provided by Skorobogatov [44], who also shows that remanence is increased when the supply during power-down is left floating instead of grounded. More recent freezing attacks have been demonstrated on a 90nm technology SRAM [52], as well as on DRAM [19]. Data remanence also imposes a fundamental limit on the throughput of true random numbers that can be generated using power-up SRAM state as an entropy source [42]. The TARDIS, in finding a constructive use for remanence and decay, can thus be seen as a counterpoint to the attacks discussed in this section. The TARDIS is the first *constructive* method that takes advantage of SRAM remanence to increase the security and privacy of intermittently powered devices.

10 Conclusions

A trustworthy source of time on batteryless devices could equip cryptographic protocols for more deliberate defense against semi-invasive attacks such as differential power

analysis and brute-force attacks. The TARDIS uses remanence decay in SRAM to compute the time elapsed during a power outage—ranging from seconds to hours depending on hardware parameters. The mechanism provides a coarse-grained notion of time for intermittently powered computers that otherwise have no effective way of measuring time. Applications using the TARDIS primarily rely on timers with hourglass-like precision to throttle queries. The TARDIS consists purely of software, making the mechanism easy to deploy on devices with SRAM. A novel aspect of the TARDIS is its use of memory decay or data remanence for improved security rather than attacking security. Without the TARDIS, batteryless devices are unlikely to give you the time of day.

Acknowledgments

The authors would like to thank our shepherd Jonathan McCune; Gesine Hinterwalter, Karsten Nohl, David Oswald, and Joshua Smith for their feedback on applications; Gildas Avoine for information on passport communication and feedback on applications; Matt Reynolds for information on the EPC gen2 protocol; Quinn Stewart for proofreading; and members of the UMass SPQR lab for reviewing early versions of this paper.

This research is supported by NSF grants CNS-0831244, CNS-0845874, CNS-0923313, CNS-0964641, SRC task 1836.074, Gigascale Systems Research Center, and a Sloan Research Fellowship. Any opinions, findings, conclusions, and recommendations expressed in these materials are those of the authors and do not necessarily reflect the views of the sponsors. Portions of this work are patent pending.

References

- [1] The TARDIS, British Broadcasting Channel. <http://www.bbc.co.uk/doctorwho/characters/tardis.shtml>, November 1963.
- [2] Hpc0402b/c - high performance, high precision wire-bondable 0402 capacitor for smartcard, high-frequency and substrate-embedded applications. <http://www.vishay.com/docs/10120/hpc0402b.pdf>, Dec. 2008.
- [3] An introduction to the architecture of Moo 1.0. https://spqr.cs.umass.edu/moo/Documents/Moo_01242011.pdf, May 2011.
- [4] ANDERSON, R., AND KUHN, M. Tamper resistance: a cautionary note. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce* (1996).
- [5] AVOINE, G. Personal communication on French passports. 2012.
- [6] AVOINE, G., KALACH, K., AND QUISQUATER, J.-J. ePassport: Securing international contacts with contactless chips. In *Financial Cryptography and Data Security* (2008), G. Tsudik, Ed., Springer-Verlag, pp. 141–155.
- [7] BONO, S., February 2012. Personal communication.

- [8] BONO, S. C., GREEN, M., STUBBLEFIELD, A., JUELS, A., RUBIN, A. D., AND SZYDLO, M. Security analysis of a cryptographically-enabled RFID device. In *Proceedings of the 14th USENIX Security Symposium* (2005).
- [9] BUETTNER, M., GREENSTEIN, B., WETHERALL, D., AND SMITH, J. R. Revisiting smart dust with RFID sensor networks, 2008.
- [10] CANTHERM. Thermal cut-offs. http://www.cantherm.com/products/thermal_fuses/sdf.html, 2011. Last Viewed May 14, 2012.
- [11] CHOTHIA, T., AND SMIRNOV, V. A traceability attack against e-Passports. In *14th International Conference on Financial Cryptography and Data Security* (2010), Springer.
- [12] EPCGLOBAL. *EPC Radio-Frequency Identity Protocols Class-1 Generation-2 UHF RFID Protocol for Communication at 860 MHz–960 MHz, Version 1.2.0*.
- [13] FLAUTNER, K., KIM, N. S., MARTIN, S., BLAAUW, D., AND MUDGE, T. Drowsy caches: simple techniques for reducing leakage power. In *Proc. 29th IEEE/ACM International Symposium on Computer Architecture* (2002), pp. 148–157.
- [14] GANERIWAL, S., ČAPKUN, S., HAN, C.-C., AND SRIVASTAVA, M. B. Secure time synchronization service for sensor networks. In *Proceedings of the 4th ACM Workshop on Wireless Security* (2005), WiSe '05, pp. 97–106.
- [15] GARCIA, F. D., ROSSUM, P. V., VERDULT, R., AND SCHREUR, R. Wirelessly pickpocketing a MIFARE Classic card. In *IEEE Symposium on Security and Privacy* (May 2009), pp. 3–15.
- [16] GOLDBERG, I., AND BRICENCO, M. GSM cloning. <http://www.isaac.cs.berkeley.edu/isaac/gsm-faq.html>, 1999. Last Viewed February 19, 2012.
- [17] GUAJARDO, J., KUMAR, S., SCHRIJEN, G., AND TUYLS, P. FPGA intrinsic PUFs and their use for IP protection. In *Cryptographic Hardware and Embedded Systems (CHES)* (2007), pp. 86–80.
- [18] GUTMANN, P. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the 6th USENIX Security Symposium* (Jan 1996).
- [19] HALDERMAN, J., SCHOEN, S., HENINGER, N., CLARKSON, W., PAUL, W., CALANDRINO, J., FELDMAN, A., APPELBAUM, J., AND FELTEN, E. Lest we remember: Cold boot attacks on encryption keys. In *Proceedings of the 17th USENIX Security Symposium* (2008).
- [20] HALPERIN, D., HEYDT-BENJAMIN, T. S., RANSFORD, B., CLARK, S. S., DEFEND, B., MORGAN, W., FU, K., KOHNO, T., AND MAISEL, W. H. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *Proceedings of the 29th Annual IEEE Symposium on Security and Privacy* (May 2008), pp. 129–142.
- [21] HEYDT-BENJAMIN, T. S., BAILEY, D. V., FU, K., JUELS, A., AND OHARE, T. Vulnerabilities in first-generation RFID-enabled credit cards. In *Proceedings of Eleventh International Conference on Financial Cryptography and Data Security, Lecture Notes in Computer Science, Vol. 4886* (February 2007), pp. 2–14.
- [22] HO, H., SAEEDI, E., KIM, S., SHEN, T., AND PARVIZ, B. Contact lens with integrated inorganic semiconductor devices. In *Micro Electro Mechanical Systems, 2008. MEMS 2008. IEEE 21st International Conference on* (Jan. 2008), pp. 403–406.
- [23] HOLCOMB, D. E., BURLISON, W. P., AND FU, K. Power-up SRAM state as an identifying fingerprint and source of true random numbers. *IEEE Transactions on Computers* (2009).
- [24] JUELS, A. Minimalist cryptography for low-cost RFID tags (extended abstract). In *Security in Communication Networks*, C. Blundo and S. Cimato, Eds., vol. 3352 of *Lecture Notes in Computer Science*. Springer, 2005, pp. 149–164.
- [25] JUELS, A. RFID security and privacy: A research survey. *IEEE Journal on Selected Areas in Communications* 24, 2 (February 2006), 381–394.
- [26] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.
- [27] LEWIS, P. H. Of privacy and security: The clipper chip debate. *The New York Times*, April 24, 1994.
- [28] LIN, Y., SYLVESTER, D. M., AND BLAAUW, D. T. A sub-pW timer using gate leakage for ultra low-power sub-Hz monitoring systems. *Custom Integrated Circuits Conference* (2007).
- [29] MAO, W. Timed-release cryptography. In *Selected Areas in Cryptography VIII (SAC'01)* (2001), Prentice Hall, pp. 342–357.
- [30] MCGRAW, G. Silver bullet podcast: Interview with Ross Anderson. <http://www.cigital.com/silver-bullet/show-070/>. Show #70, January 31, 2012.
- [31] NXP Semiconductors MIFARE classic. http://www.nxp.com/products/identification_and_security/smart_cards/mifare_smart_cards/mifare_classic/. Last Viewed February 18, 2012.
- [32] NXP Semiconductors SPI real time clock/calendar. http://www.nxp.com/documents/data_sheet/PCF2123.pdf. Last Viewed February 18, 2012.
- [33] OMEGA ENGINEERING, I. *OSXL450 Infrared Non-Contact Thermometer Manual*.
- [34] OREN, Y., AND SHAMIR, A. Remote password extraction from RFID tags. *Computers, IEEE Transactions on* 56, 9 (Sept. 2007), 1292–1296.
- [35] OSWALD, D., AND PAAR, C. Breaking MIFARE DESFire MF3ICD40: Power analysis and templates in the real world. In *Cryptographic Hardware and Embedded Systems (CHES)* (2011), pp. 207–222.
- [36] QIN, H., CAO, Y., MARKOVIC, D., VLADIMIRESCU, A., AND RABAEY, J. SRAM leakage suppression by minimizing standby supply voltage. In *Proceedings of 5th International Symposium on Quality Electronic Design* (2004), pp. 55–60.
- [37] RAJU, M. UltraLow Power RC Timer Implementation using MSP430. In *Texas Instruments Application Report SLAA119* (2000).
- [38] RANSFORD, B., CLARK, S., SALAJEGHEH, M., AND FU, K. Getting things done on computational RFIDs with energy-aware checkpointing and voltage-aware scheduling. In *USENIX Workshop on Power Aware Computing and Systems (HotPower '08)* (Dec. 2008).
- [39] RIVEST, R. L., SHAMIR, A., AND WAGNER, D. A. Time-lock puzzles and timed-release crypto. Tech. rep., Cambridge, MA, USA, 1996.
- [40] ROUSSEAU, L. Secure time in a portable device. In *Gemplus Developer Conference* (2001).
- [41] SAMPLE, A. P., YEAGER, D. J., POWLEDGE, P. S., MAMISHEV, A. V., AND SMITH, J. R. Design of an RFID-based battery-free programmable sensing platform. *IEEE Transactions on Instrumentation and Measurement* 57, 11 (Nov. 2008), 2608–2615.
- [42] SAXENA, N., AND VORIS, J. We can remember it for you wholesale: Implications of data remanence on the use of RAM for true random number generation on RFID tags. In *Proceedings of the Conference on RFID Security* (2009).

- [43] SCHNEIER, B. *Applied cryptography (2nd ed.): Protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., 1995.
- [44] SKOROBOGATOV, S. Low temperature data remanence in static RAM. Tech. Rep. UCAM-CL-TR-536, University of Cambridge Computer Laboratory, 2002.
- [45] STAJANO, F., AND ANDERSON, R. The resurrecting duckling: Security issues for ad-hoc wireless networks. In *Security Protocols*, B. Christianson, B. Crispo, J. Malcolm, and M. Roe, Eds., vol. 1796 of *Lecture Notes in Computer Science*. Springer, 2000, pp. 172–182.
- [46] SUN, K., NING, P., AND WANG, C. TinySeRSync: secure and resilient time synchronization in wireless sensor networks. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (2006)*, CCS '06, pp. 264–277.
- [47] SUN ELECTRONIC SYSTEMS, I. *Model EC1X Environmental Chamber User and Repair Manual*, 2011.
- [48] SWANSON, R., AND MEINDL, J. D. Ion-implanted complementary MOS transistors in low-voltage circuits. *International Solid-State Circuits Conference (May 1972)*.
- [49] TESSIER, R., JASINSKI, D., MAHESHWARI, A., NATARAJAN, A., XU, W., AND BURLESON, W. An energy-aware active smart card. *IEEE Transaction on Very Large Scale Integration (VLSI) Systems (2005)*.
- [50] TEXAS INSTRUMENTS INC. MSP430F21x1 Mixed Signal Microcontroller. In *Texas Instruments Application Report SLAS439F (Sep. 2004, revised Aug. 2011)*.
- [51] THINGMAGIC INC. *Mercury 4/ MERCURY 5 User Guide*, February 2007.
- [52] TUAN, T., STRADER, T., AND TRIMBERGER, S. Analysis of data remanence in a 90nm FPGA. *Custom Integrated Circuits Conference (2007)*.
- [53] VITTOZ, E. Low-power design: Ways to approach the limits. *International Solid-State Circuits Conference (May 1994)*.
- [54] XU, X., GU, L., WANG, J., AND XING, G. Negotiate power and performance in the reality of RFID systems. In *PerCom (2010)*, IEEE Computer Society, pp. 88–97.
- [55] YEAGER, D., ZHANG, F., ZARRASVAND, A., GEORGE, N., DANIEL, T., AND OTIS, B. A 9 μ a, addressable Gen2 sensor tag for biosignal acquisition. *IEEE Journal of Solid-State Circuits 45*, 10 (Oct. 2010), 2198–2209.
- [56] ZHANG, H., GUMMESON, J., RANSFORD, B., AND FU, K. Moo: A batteryless computational RFID and sensing platform. Tech. Rep. UM-CS-2011-020, Department of Computer Science, University of Massachusetts Amherst, Amherst, MA, June 2011.

Appendix

Model of Decay Probabilities

Knowing the DRV distribution of a collection of SRAM cells makes it possible to predict the amount of memory decay that will result from reaching any known minimum supply voltage during a power cycle. We propose a simple and intuitive 3-parameter (α, μ, σ) model to characterize the DRV distribution. We chose the parameters such that the model predictions agree with empirical data relating memory decay to minimum supply voltage.

Cells eligible for memory decay after being written with a value of 1 are those with a ground state of 0. We

use $g = 0$ to denote cells with a 0 ground state, and use α to denote the fraction of cells with this ground state; α is therefore the largest fraction of cells that can decay after writing a 1 to all cells.

$$\Pr(g = 0) = \alpha \quad (2)$$

Among cells that are eligible for memory decay, we assume that DRVs are normally distributed with mean μ and standard deviation σ (Equation 3).

$$DRV | (g = 0) \sim \mathcal{N}(\mu, \sigma^2) \quad (3)$$

The probability of a randomly selected cell being eligible for memory decay and having $DRV = v$ is given by Equation 4. This is an α -scaled instance of the PDF of a normally distributed random variable, and we refer to this as the “DRV probability” of voltage v .

$$\Pr((g = 0) \wedge (DRV = v)) = \frac{\alpha}{\sigma\sqrt{2\pi}} e^{-(v-\mu)^2/(2\sigma^2)} \quad (4)$$

If the minimum voltage of a power cycle is known, then the 3-parameter model can predict the memory decay. The cells that will decay are eligible cells with a DRV that is above the minimum supply voltage reached during the power cycle. A closed-form equation for predicting the memory decay from the minimum voltage and model parameters is then given by Equation 5; this equation is 1 minus the CDF of a normally distributed random variable, scaled by α .

$$D_{PRED}(v_{min}, \alpha, \mu, \sigma) = \alpha \left(1 - \frac{1 + \operatorname{erf}\left(\frac{v_{min}-\mu}{\sigma\sqrt{2}}\right)}{2} \right) \quad (5)$$

A 3-parameter model is evaluated according to how well its predicted memory decay matches empirical data. The evaluation is performed using a set of n observations $\langle v_0, D(v_0) \rangle, \langle v_1, D(v_1) \rangle, \dots, \langle v_{n-1}, D(v_{n-1}) \rangle$; each observation is a measurement of the minimum supply voltage reached during a power cycle, and the memory decay observed across that power cycle. The prediction error of any model is defined according to Equation 6. We initially use the set of measurements to find the model parameters that minimize the prediction error (see Figure 8).

$$ERR(\alpha, \mu, \sigma) = \sum_{i=0}^{n-1} (D_{PRED}(v_i, \alpha, \mu, \sigma) - D(v_i))^2 \quad (6)$$

After measurements are used to fit the model parameters to empirical data, the model is subsequently used to predict memory-decay-vs.-time curves from voltage-vs.-time measurements (see Figure 10).

Gone in 360 Seconds: Hijacking with Hitag2

Roel Verdult Flavio D. Garcia

*Institute for Computing and Information Sciences
Radboud University Nijmegen, The Netherlands.
{rverdult,flaviog}@cs.ru.nl*

Josep Balasch

*KU Leuven ESAT/COSIC and IBBT
Kasteelpark Arenberg 10, 3001 Heverlee, Belgium
josep.balasch@esat.kuleuven.be*

Abstract

An electronic vehicle immobilizer is an anti-theft device which prevents the engine of the vehicle from starting unless the corresponding transponder is present. Such a transponder is a passive RFID tag which is embedded in the car key and wirelessly authenticates to the vehicle. It prevents a perpetrator from hot-wiring the vehicle or starting the car by forcing the mechanical lock. Having such an immobilizer is required by law in several countries. Hitag2, introduced in 1996, is currently the most widely used transponder in the car immobilizer industry. It is used by at least 34 car makes and fitted in more than 200 different car models. Hitag2 uses a proprietary stream cipher with 48-bit keys for authentication and confidentiality. This article reveals several weaknesses in the design of the cipher and presents three practical attacks that recover the secret key using only wireless communication. The most serious attack recovers the secret key from a car in less than six minutes using ordinary hardware. This attack allows an adversary to bypass the cryptographic authentication, leaving only the mechanical key as safeguard. This is even more sensitive on vehicles where the physical key has been replaced by a keyless entry system based on Hitag2. During our experiments we managed to recover the secret key and start the engine of many vehicles from various makes using our transponder emulating device. These experiments also revealed several implementation weaknesses in the immobilizer units.

1 Introduction

In the past, most cars relied only on mechanical keys to prevent a hijacker from stealing the vehicle. Since the '90s most car manufacturers incorporated an electronic car immobilizer as an extra security mechanism in their vehicles. From 1995 it is mandatory that all cars sold in the EU are fitted with such an immobilizer device, ac-

ording to European directive 95/56/EC. Similar regulations apply to other countries like Australia, New Zealand (AS/NZS 4601:1999) and Canada (CAN/ULC S338-98). An electronic car immobilizer consists of two main components: a small transponder chip which is embedded in (the plastic part of) the car key, see Figure 1; and a reader which is located somewhere in the dashboard of the vehicle and has an antenna coil around the ignition, see Figure 2.



Figure 1: Car keys with a Hitag2 transponder/chip

The transponder is a passive RFID tag that operates at a low frequency wave of 125 kHz. It is powered up when it comes in proximity range of the electronic field of the reader. When the transponder is absent, the immobilizer unit prevents the vehicle from starting the engine.

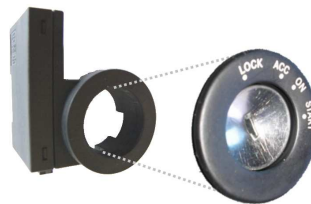


Figure 2: Immobilizer unit around the ignition barrel

A distinction needs to be made with remotely operated central locking system, which opens the doors, is battery powered, operates at a ultra-high frequency (UHF) of 433 MHz, and only activates when the user pushes a

button on the remote key. More recent car keys are often deployed with a hybrid chip that supports the battery powered ultra-high frequency as well as the passive low frequency communication interface.

With the Hitag2 family of transponders, its manufacturer NXP Semiconductors (formerly Philips Semiconductors) leads the immobilizer market [34]. Figure 4 shows a list containing some of the vehicles that are deployed with a Hitag2 transponder. Even though NXP boasts “Unbreakable security levels using mutual authentication, challenge-response and encrypted data communication”¹, it uses a shared key of only 48 bits.

Since 1988, the automotive industry has moved towards the so-called keyless ignition or keyless entry in their high-end vehicles [26]. In such a vehicle the mechanical key is no longer present and it has been replaced by a start button like the one shown in Figure 3. The only anti-theft mechanism left in these vehicles is the immobilizer. Startlingly, many keyless ignition or entry vehicles sold nowadays are still based on the Hitag2 cipher. In some keyless entry cars Hitag2 is also used as a backup mechanism for opening the doors, e.g., when the battery of the remote is depleted.



Figure 3: Keyless hybrid transponder and engine start/stop button

Related work

A similar immobilizer transponder is produced by Texas Instruments under the name Digital Signature Transponder (DST). It is protected by a different proprietary cryptographic algorithm that uses a secret key of only 40 bits. The workings of these algorithms are reversed engineered by Bono et al. in [10]. Francillon et al. demonstrated in [18] that it is possible to relay in real-time the (encrypted) communication of several keyless entry systems. The article shows that in some cases such a communication can be intercepted over a distance of at least 100 meters.

¹http://www.nxp.com/products/automotive/car_access_immobilizers/immobilizer/

Make	Models
Acura	CSX, MDX, RDX, TL, TSX
Alfa Romeo	156, 159, 166, Brera, Giulietta, Mito, Spider
Audi	A8
Bentley	Continental
BMW	Serie 1, 5, 6, 7 , all bikes
Buick	Enclave, Lucerne
Cadillac	BLS, DTS, Escalade, SRX, STS, XLR
Chevrolet	Avanache, Caprice, Captiva, Cobalt, Equinox, Express, HHR Impala, Malibu, Montecarlo, Silverado, Suburban, Tahoe Trailblazer, Uplander
Chrysler	300C, Aspen, Grand Voyager, Pacifica, Pt Cruiser, Sebring Town Country, Voyager
Citroen	Berlingo , C-Crosser, C2, C3, C4 , C4 Picasso, C5 , C6, C8 Nemo, Saxo, Xsara, Xsara Picasso
Dacia	Duster, Logan , Sandero
Daewoo	Captiva, Windstorm
Dodge	Avenger, Caliber, Caravan, Charger, Dakota, Durango Grand Caravan, Journey, Magnum, Nitro, Ram
Fiat	500, Bravo, Croma, Daily, Doblo, Fiorino, Grande Punto Panda, Phedra, Ulysse, Scudo
GMC	Acadia, Denali, Envoy, Savana, Siera, Terrain, Volt, Yukon
Honda	Accord, Civic , CR-V, Element, Fit, Insight, Stream, Jazz, Odyssey, Pilot, Ridgeline, most bikes
Hummer	H2, H3
Hyundai	130, Accent, Atos Prime, Coupe, Elantra, Excel, Getz Grandeur, I30 , Matrix, Santafe, Sonata, Terracan, Tiburon Tucson, Tuscani
Isuzu	D-Max
Iveco	35C11, Eurostar, New Daily, S-2000
Jeep	Commander, Compass, Grand Cherokee, Liberty, Patriot Wrangler
Kia	Carens, Carnival, Ceed, Cerato, Magentis, Mentor, Optima Picanto, Rio, Sephia, Sorento, Spectra, Sportage
Lancia	Delta, Musa, Phedra
Mini	Cooper
Mitsubishi	380, Colt, Eclipse, Endeavor, Galant, Grandis, L200 Lancer, Magna, Outlander, Outlander, Pajero, Raider
Nissan	Almera, Juke, Micra , Pathfinder, Primera, Qashqai, Interstar Note, Xterra
Opel	Agila, Antara, Astra, Corsa, Movano, Signum, Vectra Vivaro, Zafira
Peugeot	106, 206 , 207, 307 , 406, 407, 607, 807, 1007, 3008, 5008 Beeper, Partner, Boxer , RCZ
Pontiac	G5, G6, Pursuit, Solstice, Torrent
Porsche	Cayenne
Renault	Clio , Duster, Kangoo, Laguna II , Logan, Master Megane , Modus, Sandero, Trafic , Twingo
Saturn	Aura, Outlook, Sky, Vue
Suzuki	Alto, Grand Vitara, Splash, Swift, Vitara, XL-7
Volkswagen	Touareg, Phaeton

Figure 4: Vehicles using Hitag2 [29] – boldface indicates vehicles we tested

The history of the NXP Hitag2 family of transponders overlaps with that of other security products designed and deployed in the late nineties, such as Kee-loq [8, 13, 27, 28], MIFARE Classic [12, 19, 22, 35], CryptoMemory [4, 5, 23] or iClass [20, 21]. Originally,

information on Hitag2 transponders was limited to data sheets with high level descriptions of the chip's functionality [36], while details on the proprietary cryptographic algorithms were kept secret by the manufacturer. This phase, in which security was strongly based on obscurity, lasted until in 2007 when the Hitag2 inner workings were reverse engineered [47]. Similarly to its predecessor Crypto1 (used in MIFARE Classic), the Hitag2 cipher consists of a 48 bit Linear Feedback Shift Register (LFSR) and a non-linear filter function used to output keystream. The publication of the Hitag2 cipher attracted the interest of the scientific community. Courtois et al. [14] were the first to study the strength of the Hitag2 stream cipher to algebraic attacks by transforming the cipher state into a system of equations and using SAT solvers to perform key recovery attacks. Their most practical attack requires two days computation and a total of four eavesdropped authentication attempts to extract the secret key. A more efficient attack, requiring 16 chosen initialization vectors (IV) and six hours of computations, was also proposed. However, and as noted by the authors themselves, chosen-IV attacks are prevented by the Hitag2 authentication protocol (see Sect. 3.5), thus making this attack unfeasible in practice.

In [42], Soos et al. introduced a series of optimizations on SAT solvers that made it possible to reduce the attack time of Courtois et al. to less than 7 hours. More recently, Štembera and Novotný [45] implemented a brute-force attack that could be carried out in less than two hours by using the COPACOBANA² high-performance cluster of FPGAs. Note however, that such attack would require about 4 years if carried out on a standard PC. Finally, Sun et. al [44] tested the security of the Hitag2 cipher against cube attacks. Although according to their results the key can be recovered in less than a minute, this attack requires chosen initialization vectors and thus should be regarded as strictly theoretical.

Our contribution

In this paper, we show a number of vulnerabilities in the Hitag2 transponders that enable an adversary to retrieve the secret key. We propose three attacks that extract the secret key under different scenarios. We have implemented and successfully executed these attacks in practice on more than 20 vehicles of various make and model. On all these vehicles we were able to use an emulating device to bypass the immobilizer and start the vehicle.

Concretely, we found the following vulnerabilities in Hitag2.

- The transponder lacks a pseudo-random number generator, which makes the authentication proced-

²<http://www.copacobana.org>

ure vulnerable to replay attacks. Moreover, the transponder provides known data when a read command is issued on the block where the transponder's identity is stored, allowing to recover keystream. Redundancy in the commands allow an adversary to expand this keystream to arbitrary lengths. This means that the transponder provides an arbitrary length keystream oracle.

- With probability 1/4 the output bit of the cipher is determined by only 34 bits of the internal state. As a consequence, (on average) one out of four authentication attempts leaks one bit of information about the secret key.
- The 48 bit internal state of the cipher is only randomized by a nonce of 32 bits. This means that 16 bits of information over the secret key are persistent throughout different sessions.

We exploit these vulnerabilities in the following three practical attacks.

- The first attack exploits the malleability of the cipher and the fact that the transponder does not have a pseudo-random number generator. It uses a keystream shifting attack following the lines of [16]. This allows an adversary to first get an authentication attempt from the reader which can later be replayed to the transponder. Exploiting the malleability of the cipher, this can be used to read known plaintext (the identity of the transponder) and recover keystream. In a new session the adversary can use this keystream to read any other memory block (with exception of the secret key when configured correctly) within milliseconds. When the key is not read protected, this attack can also be used to read the secret key. This was in fact the case for most vehicles we tested from a French car make.
- The second attack is slower but more general in the sense that the same attack strategy can be applied to other LFSR based ciphers. The attack uses a time/memory tradeoff as proposed in [3, 6, 7, 11, 25, 38]. Exploiting the linear properties of the LFSR, we are able to efficiently generate the lookup table, reducing the complexity from 2^{48} to 2^{37} encryptions. This attack recovers the secret key regardless of the read protection configuration of the transponder. It requires 30 seconds of communication with the transponder and another 30 seconds to perform 2000 table lookups.
- The third attack is also the most powerful, as it only requires a few authentication attempts from the car immobilizer to recover the secret key (assuming that

the adversary knows a valid transponder *id*). This cryptanalytic attack exploits dependencies among different sessions and a low degree determination of the filter function used in the cipher. In order to execute this attack, an adversary first gathers 136 partial authentication attempts from the car. This can be done within one minute. Then, the adversary needs to perform 2^{35} operations to recover the secret key. This takes less than five minutes on an ordinary laptop.

Furthermore, besides looking into the security aspects of Hitag2 we also study how it is deployed and integrated in car immobilizer systems by different manufacturers. Our study reveals that in many vehicles the transponder is misconfigured by having readable or default keys, and predictable passwords, whereas the immobilizer unit employs weak pseudo-random number generators. All cars we tested use identifier white-listing as an additional security mechanism. This means that in order to use our third attack to hijack a car, an adversary first needs to eavesdrop, guess or wirelessly pickpocket a legitimate transponder *id*, see Section 7.5.

Following the principle of responsible disclosure, we have contacted the manufacturer NXP and informed them of our findings six months ahead of publication. We have also provided our assistance in compiling a document to inform their customers about these vulnerabilities. The communication with NXP has been friendly and constructive. NXP encourages the automotive industry for years to migrate to more secure products that incorporate strong and community-reviewed ciphers like AES [15]. It is surprising that the automotive industry is reluctant to migrate to secure products given the cost difference of a better chip (≤ 1 USD) in relation to the prices of high-end car models ($\geq 50,000$ USD).

2 Hardware setup

Before diving into details about Hitag2, this section introduces the experimental platform we have developed in order to carry out attacks in real-life deployments of car immobilizer systems. In particular, we have built a portable and highly flexible setup allowing us to i) eavesdrop communications between Hitag2 readers and transponders, ii) emulate a Hitag2 reader, and iii) emulate a Hitag2 transponder. Figure 5 depicts our setup in the setting of eavesdropping communications between a reader and a transponder.

The central element of our experimental platform is the Proxmark III board³, originally developed by Jonathan Westhues⁴, and designed to work with RFID

³<http://www.proxmark.org>
⁴<http://cq.cx/proxmark3.pl>



Figure 5: Experimental setup for eavesdropping

transponders ranging from low frequency (125 kHz) to high frequency (13.56 MHz). The Proxmark III board cost around 200 USD and comes equipped with a FPGA and an ARM microcontroller. Low-level RF operations such as modulation/demodulation are carried out by the FPGA, whereas high-level operations such as encoding/decoding of frames are performed in the microcontroller.

Hitag2 tags are low frequency transponders used in proximity area RFID applications [36]. Communication from reader to transponder is encoded using Binary Pulse Length Modulation (BPLM), whereas from transponder to reader it can be encoded using either Manchester or Biphase coding. In order to eavesdrop, generate, and read communications from reader to transponder, we added support for encoding/decoding BPLM signals, see Figure 6.

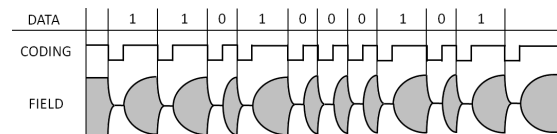


Figure 6: Reader modulation of a *read* command

For the transponder side, we have also added the functionalities to support the Manchester coding scheme as shown in Figure 7.

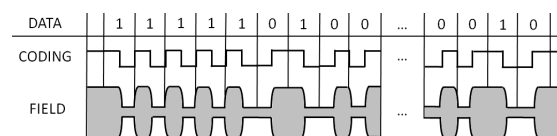


Figure 7: Communication from transponder to reader

3 Hitag2

This section describes Hitag2 in detail. Most of this information is in the public domain. We first describe the Hitag2 functionality, memory structure, and communication protocols, this comes mostly from the product data sheet [36]. Then we describe the cipher and the authentication protocol which was previously reverse engineered in [47]. In Section 3.7 we show that it is possible to run the cipher backwards which we use in our attacks.

We first need to introduce some notation. Let $\mathbb{F}_2 = \{0, 1\}$ the field of two elements (or the set of Booleans). The symbol \oplus denotes exclusive-or (XOR) and 0^n denotes a bitstring of n zero-bits. Given two bitstrings x and y , xy denotes their concatenation. \bar{x} denotes the bitwise complement of x . We write y_i to denote the i -th bit of y . For example, given the bitstring $y = 0x03$, $y_0 = y_1 = 0$ and $y_6 = y_7 = 1$. We denote encryptions by $\{-\}$.

3.1 Functionality

Access to the Hitag2 memory contents is determined by pre-configured security policies. Hitag2 transponders offer up to three different modes of operation:

1. In *public mode* the contents of the user data pages are simply broadcast by the transponder once it is powered up.
2. In *password mode* reader and transponder authenticate each other by interchanging their passwords. Communication is carried out in the clear, therefore this authentication procedure is vulnerable to replay attacks.
3. In *crypto mode* the reader and the transponder perform a mutual authentication by means of a 48-bit shared key. Communication between reader and transponder is encrypted using a proprietary stream cipher. This mode is used in car immobilizer systems and will be the focus of this paper.

3.2 Memory

Hitag2 transponders have a total of 256 bits of non-volatile memory (EEPROM) organized in 8 blocks of 4 bytes each. Figure 8 illustrates the memory contents of a transponder configured in crypto mode. Block 0 stores the read-only transponder identifier; the secret key is stored in blocks 1 and 2; the password and configuration bits in block 3; blocks 4 till 7 store user defined memory. Access to any of the memory blocks in crypto mode is only granted to a reader after a successful mutual authentication.

Block	Contents
0	transponder identifier id
1	secret key low $k_0 \dots k_{31}$
2	secret key high $k_{32} \dots k_{47}$ — reserved
3	configuration — password
4–7	user defined memory

Figure 8: Hitag2 memory map in crypto mode [36]

3.3 Communication

The communication protocol between the reader and transponder is based on the master-slave principle. The reader sends a command to the transponder, which then responds after a predefined period of time. There are five different commands: *authenticate*, *read*, *read*, *write* and *halt*. As shown in Figure 9, the *authenticate* command has a fixed length of 5 bits, whereas the others have a length of at least 10 bits. Optionally, these 10 bits can be extended with a redundancy message of size multiple of 5 bits. A redundancy message is composed by the bit-complement of the last five bits of the command. According to the datasheet [36] this feature is introduced to “achieve a higher confidence level”.

In crypto mode the transponder starts in a halted state and is activated by the *authenticate* command. After a successful authentication, the transponder enters the active state in which it only accepts active commands which are encrypted. Every encrypted bit that is transferred consists of a plaintext bit XOR-ed with one bit of the keystream. The active commands have a 3-bit argument n which represents the offset (block number) in memory. From this point we address Hitag2 active commands by referring to *commands* and explicitly mention authentication otherwise.

Command	Bits	State
<i>authenticate</i>	11000	halted
<i>read</i>	$11n_0n_1n_200\overline{n_0n_1n_2} \dots$	active
<i>read</i>	$01n_0n_1n_210\overline{n_0n_1n_2} \dots$	active
<i>write</i>	$10n_0n_1n_201\overline{n_0n_1n_2} \dots$	active
<i>halt</i>	$00n_0n_1n_211\overline{n_0n_1n_2} \dots$	active

Figure 9: Hitag2 commands using block number n

Next we define the function *cmd* which constructs a bit string that represents a command c on block n with r redundancy messages.

Definition 3.1. Let c be the first 2-bit command as defined in Figure 9, n be a 3-bit memory block number

and r be the number of redundancy messages. Then, the function $cmd: \mathbb{F}_2^2 \times \mathbb{F}_2^3 \times \mathbb{N} \rightarrow \mathbb{F}_2^{(10+5r)}$ is defined by

$$cmd(c, n, 0) = cn\bar{c}n$$

$$cmd(c, n, r+1) = \begin{cases} cmd(c, n, r)cn, & r \text{ is odd;} \\ cmd(c, n, r)\bar{c}n, & \text{otherwise.} \end{cases}$$

For example, the command to read block 0 with two redundancy messages results in the following bit string.

$$cmd(11, 0, 2) = 11000\ 00111\ 11000\ 00111$$

The encrypted messages between reader and transponder are transmitted without any parity bits. The transponder response always starts with a prefix of five ones, see Figure 10. In the remainder of this paper we will omit this prefix. A typical forward and backwards communication takes about 12 ms.

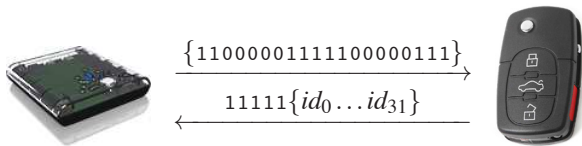


Figure 10: Message flow for reading memory block 0

3.4 Cipher

In crypto mode, the communication between transponder and reader (after a successful authentication) is encrypted with the Hitag2 stream cipher. This cipher has been reverse engineered in [47]. The cipher consists of a 48-bit linear feedback shift register (LFSR) and a non-linear filter function f . Each clock tick, twenty bits of the LFSR are put through the filter function, generating one bit of keystream. Then the LFSR shifts one bit to the left, using the generating polynomial to generate a new bit on the right. See Figure 11 for a schematic representation.

Definition 3.2. The feedback function $L: \mathbb{F}_2^{48} \rightarrow \mathbb{F}_2$ is defined by $L(x_0 \dots x_{47}) := x_0 \oplus x_2 \oplus x_3 \oplus x_6 \oplus x_7 \oplus x_8 \oplus x_{16} \oplus x_{22} \oplus x_{23} \oplus x_{26} \oplus x_{30} \oplus x_{41} \oplus x_{42} \oplus x_{43} \oplus x_{46} \oplus x_{47}$.

The filter function f consists of three different circuits f_a, f_b and f_c which output one bit each. The circuits f_a and f_b are employed more than once, using a total of twenty input bits from the LFSR. Their resulting bits are used as input for f_c . The circuits are represented by three boolean tables that contain the resulting bit for each input.

Definition 3.3 (Filter function). The filter function $f: \mathbb{F}_2^{48} \rightarrow \mathbb{F}_2$ is defined by

$$f(x_0 \dots x_{47}) = f_c(f_a(x_2, x_3, x_5, x_6), f_b(x_8, x_{12}, x_{14}, x_{15}), f_b(x_{17}, x_{21}, x_{23}, x_{26}), f_b(x_{28}, x_{29}, x_{31}, x_{33}), f_a(x_{34}, x_{43}, x_{44}, x_{46})),$$

where $f_a, f_b: \mathbb{F}_2^4 \rightarrow \mathbb{F}_2$ and $f_c: \mathbb{F}_2^5 \rightarrow \mathbb{F}_2$ are

$$f_a(i) = (0xA63C)_i$$

$$f_b(i) = (0xA770)_i$$

$$f_c(i) = (0xD949CBB0)_i.$$

For future reference, note that each of the building blocks of f (and hence f itself) has the property that it outputs zero for half of the possible inputs (respectively one).

Remark 3.4 (Cipher schematic). Figure 11 is different from the schematic that was introduced by [47] and later used by [14, 19, 44, 45]. The input bits of the filter function in Figure 11 are shifted by one with respect to those of [47]. The filter function in the old schematic represents a keystream bit at the previous state $f(x_{i-1} \dots x_{i+46})$, while the one in Figure 11 represents a keystream bit of the current state $f(x_i \dots x_{i+47})$. Furthermore, we have adapted the boolean tables to be consistent with our notation.

3.5 Authentication protocol

The authentication protocol used in Hitag2 in crypto mode, reverse engineered and published online in 2007 [47], is depicted in Figure 12. The reader starts the communication by sending an authenticate command, to which the transponder answers by sending its identifier id . From this point on, communication is encrypted, i.e., XOR-ed with the keystream. The reader responds with its encrypted challenge n_R and the answer $a_R = 0xFFFFFFFF$ also encrypted to prove knowledge of the key; the transponder finishes with its encrypted answer a_T (corresponding to block 3 in Fig. 8) to the challenge of the reader.

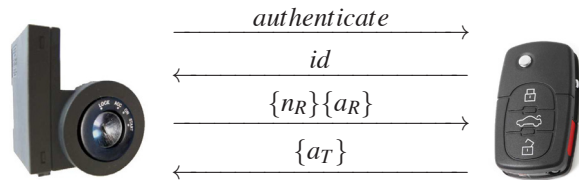


Figure 12: Hitag2 authentication protocol

During the authentication protocol, the internal state of the stream cipher is initialized. The initial state consists of the 32-bits identifier concatenated with the first 16 bits of the key. Then reader nonce n_R XORed with the last 32 bits of the key is shifted in. During initialization, the LFSR feedback is disabled. Since communication is encrypted from n_R onwards, the encryption of the later bits of n_R are influenced by its earlier bits. Authentication is achieved by reaching the same internal state of the cipher after shifting in n_R .

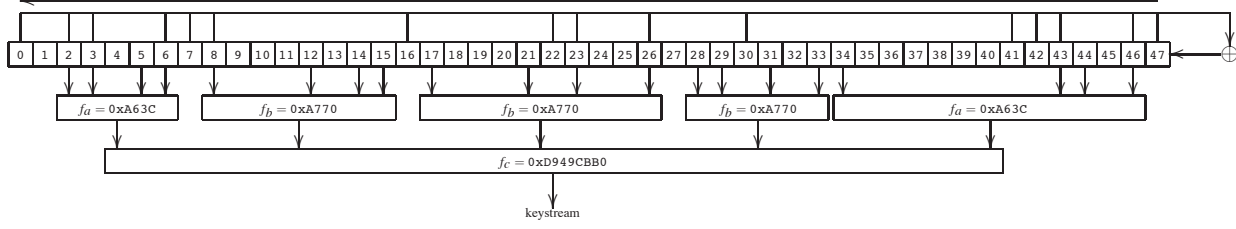


Figure 11: Structure of the Hitag2 stream cipher, based on [47]

3.6 Cipher Initialization

The following precisely defines the initialization of the cipher and the generation of the LFSR-stream $a_0a_1\dots$ and the keystream $b_0b_1\dots$.

Definition 3.5. Given a key $k = k_0\dots k_{47} \in \mathbb{F}_2^{48}$, an identifier $id = id_0\dots id_{31} \in \mathbb{F}_2^{32}$, a reader nonce $n_R = n_{R_0}\dots n_{R_{31}} \in \mathbb{F}_2^{32}$, a reader answer $a_R = a_{R_0}\dots a_{R_{31}} \in \mathbb{F}_2^{32}$, and a transponder answer $a_T = a_{T_0}\dots a_{T_{31}} \in \mathbb{F}_2^{32}$, the internal state of the cipher at time i is $\alpha_i := a_i\dots a_{47+i} \in \mathbb{F}_2^{48}$. Here the $a_i \in \mathbb{F}_2$ are given by

$$\begin{aligned} a_i &:= id_i & \forall i \in [0, 31] \\ a_{32+i} &:= k_i & \forall i \in [0, 15] \\ a_{48+i} &:= k_{16+i} \oplus n_{R_i} & \forall i \in [0, 31] \\ a_{80+i} &:= L(a_{32+i}\dots a_{79+i}) & \forall i \in \mathbb{N}. \end{aligned}$$

Furthermore, we define the keystream bit $b_i \in \mathbb{F}_2$ at time i by

$$b_i := f(a_i\dots a_{47+i}) \quad \forall i \in \mathbb{N}.$$

Define $\{n_R\}_i, \{a_R\}_i, \{a_T\}_i \in \mathbb{F}_2$ by

$$\begin{aligned} \{n_R\}_i &:= n_{R_i} \oplus b_i & \forall i \in [0, 31] \\ \{a_R\}_i &:= a_{R_i} \oplus b_{32+i} & \forall i \in [0, 31] \\ \{a_T\}_i &:= a_{T_i} \oplus b_{64+i} & \forall i \in [0, 31]. \end{aligned}$$

Note that the $a_i, \alpha_i, b_i, \{n_R\}_i, \{a_R\}_i,$ and $\{a_T\}_i$ are formally functions of $k, id,$ and n_R . Instead of making this explicit by writing, e.g., $a_i(k, id, n_R)$, we just write a_i where $k, id,$ and n_R are clear from the context.

3.7 Rollback

To recover the key it is sufficient to learn the internal state of the cipher α_i at any point i in time. Since an attacker knows id and $\{n_R\}$, the LFSR can then be rolled back to time zero.

Definition 3.6. The rollback function $R: \mathbb{F}_2^{48} \rightarrow \mathbb{F}_2^{48}$ is defined by $R(x_1\dots x_{48}) := x_2 \oplus x_3 \oplus x_6 \oplus x_7 \oplus x_8 \oplus x_{16} \oplus x_{22} \oplus x_{23} \oplus x_{26} \oplus x_{30} \oplus x_{41} \oplus x_{42} \oplus x_{43} \oplus x_{46} \oplus x_{47} \oplus x_{48}$.

If one first shifts the LFSR left using L to generate a new bit on the right, then R recovers the bit that dropped out on the left, i.e.,

$$R(x_1\dots x_{47} L(x_0\dots x_{47})) = x_0. \quad (1)$$

Theorem 3.7. In the situation from Definition 3.5, we have

$$\begin{aligned} a_{32+i} &= R(a_{33+i}\dots a_{80+i}) & \forall i \in \mathbb{N} \\ a_i &= id_i & \forall i \in [0, 31]. \end{aligned}$$

Proof. Straightforward, using Definition 3.5 and Equation (1). \square

If an attacker manages to recover the internal state of the LFSR $\alpha_i = a_i a_{i+1} \dots a_{i+47}$ at some time i , then she can repeatedly apply Theorem 3.7 to recover $a_0 a_1 \dots a_{79}$ and, consequently, the keystream $b_0 b_1 b_2 \dots$. By having eavesdropped $\{n_R\}$ from the authentication protocol, the adversary can further calculate

$$n_{R_i} = \{n_R\}_i \oplus b_i \quad \forall i \in [0, 31].$$

Finally, the adversary can compute the secret key as follows

$$\begin{aligned} k_i &= a_{32+i} & \forall i \in [0, 15] \\ k_{16+i} &= a_{48+i} \oplus n_{R_i} & \forall i \in [0, 31]. \end{aligned}$$

4 Hitag2 weaknesses

This section describes three weaknesses in the design of Hitag2. The first one is a protocol flaw while the last two concern the cipher's design. These weaknesses will later be exploited in Section 5.

4.1 Arbitrary length keystream oracle

This weakness describes that without knowledge of the secret key, but by having only one authentication attempt, it is possible to gather an arbitrary length of keystream bits from the transponder. Section 3.3 describes the reader commands that can modify or halt a Hitag2 transponder. As mentioned in Definition 3.1 it is possible to extend the length of such a command with a multiple of five bits. A 10-bit command can have an optional number of redundancy messages r so that the total bit count of the message is $10 + 5r$ bits. Due to power and memory constraints, Hitag2 seems to be designed

to communicate without a send/receive buffer. Therefore, all cipher operations are performed directly at arrival or transmission of bits. Experiments show that a Hitag2 transponder successfully accepts encrypted commands from the reader which are sent with 1000 redundancy messages. The size of such a command consists of $10 + 5 \times 1000 = 5010$ bits.

Since there is no challenge from the transponder it is possible to replay any valid $\{n_R\}\{a_R\}$ pair to the transponder to achieve a successful authentication. After receiving a_T , the internal state of the transponder is initialized and waits for an encrypted command from the reader as defined in Figure 9. Without knowledge of the keystream bits $b_{96}b_{97}\dots$ and onwards, all possible combinations need to be evaluated. A command consist of at least 10 bits, therefore there are 2^{10} possibilities. Each command requires a 3-bit parameter containing the block number. Both *read* and *read* receive a 32-bit response, while the write and halt have a different response length. Hence, when searching for 10-bit encrypted commands that get a 32-bit response there are exactly 16 out of the 2^{10} values that match. On average the first *read* command is found after 32 attempts, the complement of this *read* and its parameters are a linear difference and therefore take only 15 attempts more.

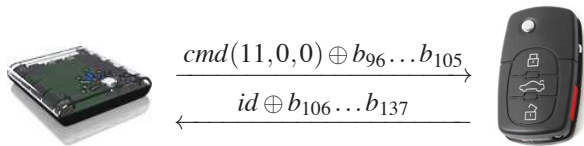


Figure 13: Read *id* without redundancy messages

One of the 16 guesses represents the encrypted bits of the read command on the first memory block. This block contains the *id* which is known plaintext since it is transmitted in the clear during the authentication. Therefore, there is a guess such that the communicated bits are equal to the messages in Figure 13.

With the correct guess, 40 keystream bits can be recovered. This keystream is then used to encrypt a slightly modified read command on block 0 with six redundancy messages, as explained in Section 3.3. The transponder responds with the next 32-bit of keystream which are used to encrypt the identifier as shown in Figure 14. Hence the next 30 keystream bits were retrieved using previously recovered keystream and by extending the *read* command.

This operation can be repeated many times. For example, using the recovered keystream bits $b_{96}\dots b_{167}$ it is possible to construct a 70-bit *read* command with 12 redundancy messages etc. In practice it takes less than 30 seconds to recover 2048 bits of contiguous keystream.

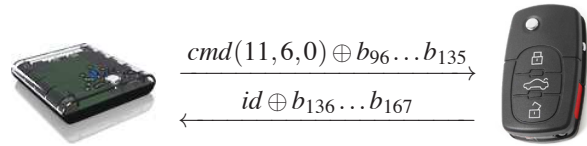


Figure 14: Read *id* using 6 redundancy messages

4.2 Dependencies between sessions

Section 3.6 shows that at cipher state α_{79} the cipher is fully initialized and from there on the cipher only produces keystream. This shows that the 48-bit internal state of the cipher is randomized by a reader nonce n_R of only 32 bits. Consequently, at state α_{79} , only LFSR bits 16 to 47 are affected by the reader nonce. Therefore LFSR bits 0 to 15 remain constant throughout different session which gives a strong dependency between them. These 16 session persistent bits correspond to bits $k_0\dots k_{15}$ of the secret key.

4.3 Low degree determination of the filter function

The filter function $f: \mathbb{F}_2^{48} \rightarrow \mathbb{F}_2$ consists of three building blocks f_a, f_b and f_c arranged in a two layer structure, see Figure 11. Due to this particular structure, input bits $a_{34}\dots a_{47}$ only affect the rightmost input bit of f_c . Furthermore, simple inspection of f_c shows that in 8 out of 32 configurations of the input bits, the rightmost input bit has no influence on the output of f_c . In those cases the output of f_c is determined by its 4-leftmost input bits. Furthermore, this means that with probability 1/4 the filter function f is determined by the 34-leftmost bits of the internal state. The following theorem states this precisely.

Theorem 4.1. *Let X be a uniformly distributed variable over \mathbb{F}_2^{34} . Then*

$$\mathbb{P}[\forall Y, Y' \in \mathbb{F}_2^{14} : f(XY) = f(XY')] = 1/4.$$

Proof. By inspection. □

Definition 4.2. *The function that checks for this property $P: \mathbb{F}_2^{48} \rightarrow \mathbb{F}_2$ is defined by*

$$P(x_0\dots x_{47}) = (0x84D7)_i$$

where

$$i = f_a(x_2x_3x_5x_6)f_b(x_8x_{12}x_{14}x_{15}) \\ f_b(x_{17}x_{21}x_{23}x_{26})f_b(x_{28}x_{29}x_{31}x_{33}).$$

Because $P(x_0\dots x_{47})$ only depends on $x_0\dots x_{33}$ we shall overload notation and see $P(\cdot)$ as a function $\mathbb{F}_2^{34} \rightarrow \mathbb{F}_2$, writing $P(x_0\dots x_{47})$ as $P(x_0\dots x_{33}0^{14})$.

5 Attacks

This section describes three attacks against Hitag2. The first attack is straightforward and grants an adversary read and write access to the memory of the transponder. The cryptanalysis described in the second attack recovers the secret key after briefly communicating with the car and the transponder. This attack uses a general technique that can be applied to other LFSR-like stream ciphers. The third attack describes a custom cryptanalysis of the Hitag2 cipher. It only requires a few authentication attempts from the car and allows an adversary to recover the secret key with a computational complexity of 2^{35} operations. The last two attacks allow a trade-off between time/memory/data and time/traces respectively. For the sake of simplicity we describe these attacks with concrete values that are either optimal or what we consider ‘sensible’ in view of currently available hardware.

5.1 Malleability attack

This attack exploits the arbitrary length keystream oracle weakness described in Section 4.1, and the fact that during the authentication algorithm the transponder does not provide any challenge to the reader. This notorious weaknesses allow an adversary to first acquire keystream and then use it to read or write any block on the card with constant communication and computational complexity. After the recovery of the keystream bits $b_{96} \dots b_{137}$ as shown in Figure 13 an adversary can dump the complete memory of the transponder which includes its password. Recovery of the keystream and creating a memory dump from the transponder takes in total less than one second and requires only to be in proximity distance of the victim. This shows a similar scenario to [22] where Garcia et al. show how to wirelessly pickpocket a MIFARE Classic card from the victim.

The memory blocks where the cryptographic key is stored have an extra optional protection mechanism. There is a one time programmable configuration bit which determines whether these blocks are readable or not. If the reader tries to read a protected block, then the transponder does not respond. In that case the adversary can still use the attacks presented in Section 5.2 and Section 5.3. If the transponder is not correctly configured, it enables an adversary to read all necessary data to start the car.

5.2 Time/memory tradeoff attack

This attack is very general and it can be applied to any LFSR-based stream cipher as long as enough contiguous keystream is available. This is in fact the case with Hitag2 due to the weakness described in Section 4.1. It

extends the methods of similar time/memory tradeoffs articles published over the last decades [3, 6, 7, 11, 25, 38]. This attack requires communication with the reader and the transponder. The next proposition introduces a small trick that makes it possible to quickly perform n cipher steps at once. Intuitively, this proposition states that the linear difference between a state s and its n -th successor is a combination of the linear differences generated by each bit. This will be later used in the attack.

Proposition 5.1. *Let s be an LFSR state and $n \in \mathbb{N}$. Furthermore, let $d_i = \text{suc}^n(2^i)$ i.e., the LFSR state that results from running the cipher n steps from the state 2^i . Then*

$$\text{suc}^n(s) = \bigoplus_{i=0}^{47} (d_i \cdot s_i).$$

To perform the attack the adversary A proceeds as follows:

1. Only once, A builds a table containing 2^{37} entries. Each entry in the table is of the form $\langle ks, s \rangle$ where $s \in \mathbb{F}_2^{48}$ is an LFSR state and $ks \in \mathbb{F}_2^{48}$ are 48 bits of keystream produced by the cipher when running from s . Starting from some state where $s \neq 0$, the adversary generates 48 bits of keystream and stores it. Then it uses Theorem 5.1 to quickly jump $n = 2^{11}$ cipher states to the next entry in the table. This reduces the computational complexity of building the table from 2^{48} to $48 \times 2^{37} = 2^{42.5}$ cipher ticks. Moreover, in order to improve lookup time the table is sorted on ks and divided into 2^{24} sub-tables encoded in the directory structure like `/ks_byte1/ks_byte2/ks_byte3.bin` where each `ks_byte3.bin` file has only 8 KB. The total size of this table amounts 1.2 TB.
2. A emulates a transponder and runs an authentication attempt with the target car. Following the authentication protocol, the car answers with a message $\{n_R\}\{a_R\}$.
3. Next, the attacker wirelessly replays this message to the legitimate transponder and uses the weakness described in Section 4.1 to obtain 256 bytes of keystream $ks_0 \dots ks_{2048}$. Note that this might be done while the key is in the victim’s bag or pocket.
4. The adversary sets $i = 0$.
5. Then it looks up (in logarithmic time) the keystream $ks_i \dots ks_{i+47}$ in the table from step 1.
6. If the keystream is not in the table then it increments i and goes back to step 5. If there is a match, then the corresponding state is a candidate internal state. A uses the rest of the keystream to confirm if this is the internal state of the cipher.

- Finally, the adversary uses Theorem 3.7 to rollback the cipher state and recover the secret key.

Complexity and time. In step 1 the adversary needs to pre-compute a 1.2 TB table which requires $2^{42.5}$ cipher ticks, which is equal to 2^{37} encryptions. During generation, each entry is stored directly in the corresponding .bin file as mentioned before. Each of these 8 KB files also needs to be sorted but it only takes a few minutes to sort them all. Computing and sorting the whole table takes less than one day on a standard laptop. Steps 2-3 take about 30 seconds to gather the 256 bytes of key-stream from the transponder. Steps 4-6 require (in worst case) 2000 table lookups which take less than 30 seconds on a standard laptop. This adds to a total of one minute to execute the attack from begin to end.

5.3 Cryptanalytic attack

A combination of the weaknesses described in Section 4.2 and 4.3 enable an attacker to recover the secret key after gathering a few authentication attempts from a car. In case that identifier white-listing is used as a secondary security measure, which is in fact the case for all the cars we tested, the adversary first needs to obtain a valid transponder *id*, see Section 7.5.

The intuition behind the attack is simple. Suppose that an adversary has a guess for the first 34 bits of the key. One out of four traces is expected to have the property from Theorem 4.1 which enables the adversary to perform a test on the first bit of $\{a_R\}$. The dependencies between sessions described in Section 4.2 allow the attacker to perform this test many times decreasing drastically the amount of candidate (partial) keys. If an attacker gathers 136 traces this allows her (on average) to perform $136/4 = 34$ bit tests, i.e. just as much as key bits were guessed. For the small amount of candidate keys that pass these tests (typically 2 or 3), the adversary performs an exhaustive search for the remaining 14 bits of the key. A precise description of this attack follows.

- The attacker uses a transponder emulator (like the Proxmark III) to initiate 136 authentication attempts with the car using a fixed transponder *id*. In this way the attacker gathers 136 traces of the form $\{n_R\}\{a_R\}$. Next the attacker starts searching for the secret key. For this we split the key k in three parts $k = \vec{k}\hat{k}\vec{k}$ where $\vec{k} = k_0 \dots k_{15}$, $\hat{k} = k_{16} \dots k_{33}$, and $\vec{k} = k_{34} \dots k_{47}$.

- for each $\vec{k} = k_0 \dots k_{15} \in \mathbb{F}_2^{16}$ the attacker builds a table $T_{\vec{k}}$ containing entries

$$\langle y \oplus b_0 \dots b_{17}, \overline{b_{32}}, \vec{k}y \rangle$$

for all $y \in \mathbb{F}_2^{18}$ such that $P(\vec{k}y0^{14}) = 1$. Note that the expected size of this table is $2^{18} \times 1/4 = 2^{16}$ which easily fits in memory.

- For each $\hat{k} = k_{16} \dots k_{33} \in \mathbb{F}_2^{18}$ and for each trace $\{n_R\}\{a_R\}$, the attacker sets $z := \hat{k} \oplus \{n_R\}_0 \dots \{n_R\}_{17}$. If there is an entry in $T_{\vec{k}}$ for which $y \oplus b_0 \dots b_{17}$ equals z but $\overline{b_{32}} \neq \{a_R\}_0$ then the attacker learns that \hat{k} is a bad guess, so he tries the next one. Otherwise, if $\overline{b_{32}} = \{a_R\}_0$ then \hat{k} is still a viable guess and therefore the adversary tries the next trace.
- Each $\vec{k}\hat{k}$ that passed the test for all traces is a partial candidate key. For each such candidate (typically 2 or 3), the adversary performs an exhaustive search for the remaining key bits $\vec{k} = k_{34} \dots k_{47}$. For each full candidate key, the adversary decrypts two traces and checks whether both $\{a_R\}$ decrypt to all ones as specified in the authentication protocol. If a candidate passes this test then it is the secret key. If none of them passes then the adversary goes back to Step 2 and tries the next \vec{k} .

Complexity and time. In step 1, the adversary needs to gather 136 partial authentication traces. This can be done within 1 minute using the Proxmark III. In steps 2 and 3, the adversary needs to build 2^{16} tables. For each of these tables the adversary needs to compute 2^{18} encryptions plus 2^{18} table lookups. Step 4 has negligible complexity thus we ignore it. This adds to a total complexity of $2^{16} \times (2^{18} + 2^{18}) = 2^{35}$ encryptions/lookups. Note that it is straightforward to split up the search space of \vec{k} in as many processes as you wish. On an standard quad-core laptop this computation takes less than five minutes. Therefore, the whole attack can be performed in less than 360 seconds which explains the title of the paper.

This attack is faster than other practical attacks proposed in [14, 45]. The following table shows a comparison between this attack and other attacks from the literature.

Attack	Description	Practical	Computation	Traces	Time
[45]	brute-force	yes	2 102 400 min	2	4 years
[14]	sat-solver	yes	2 880 min	4	2 days
[42]	sat-solver	no ¹	386 min	N/A	N/A
[44]	cube	no ²	1 min	500	N/A
Our	cryptanalytic	yes	5 min	136	6 min

¹Soos et al. require 50 bits of contiguous keystream.

²Sun et al. require control over the encrypted reader nonce $\{n_R\}$

Figure 15: Comparison of attack times and requirements

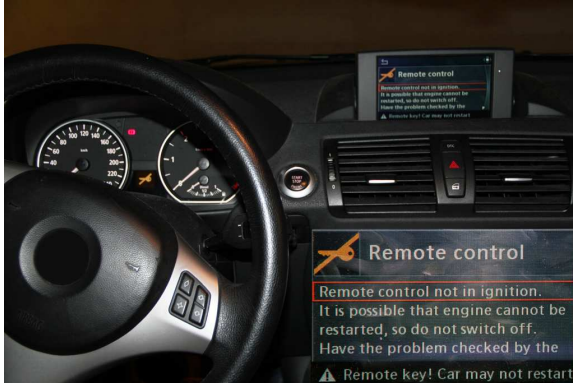


Figure 16: Left: Authentication failure message
Right: Successful authentication using a Proxmark III

6 Starting a car

In order to elaborate on the practicality of our attacks, this section describes our experience with one concrete vehicle. For this we have chosen a German car, mainly due to the fact that it has keyless ignition. Instead of the typical mechanical key, this car has a hybrid remote control which contains a Hitag2 transponder. In the dashboard of the car there is a slot to insert the remote and a button to start the engine. When a piece of plastic of suitable size is inserted in this slot the car repeatedly attempts to authenticate the transponder (and fails). This car uses an identifier white-list as described in Section 7.5. The same section explains how to wirelessly pickpocket a valid identifier from the victim's remote. As soon as the car receives a valid identifier, the dashboard lights up and the LCD screen pops-up displaying the message shown in Figure 16-Left. Note also the sign on the dashboard. At this point we used the Proxmark to quickly gather enough traces and execute the attack from Section 5.3 to recover the secret key. This car is one of the few that we tested that does not have a predictable password so we wirelessly read it from the victim's remote. Then we use the Proxmark to emulate the transponder. Figure 16-Right shows that the car accepts the Proxmark as if it was the legitimate transponder. The same picture shows (by looking at the tachometer) that at this stage it is possible to start the engine.

7 Implementation weaknesses

To verify the practicality of our attacks, we have tested all three of them on at least 20 different car models from various makes. During our experiments we found that, besides the weaknesses in cipher and protocol, the transponder is often misconfigured and poorly integrated in the cars. Most of the cars we tested use a default

or predictable transponder password. Some generate nonces with a very low entropy. Most car keys have vehicle-dependant information stored in the user defined memory of the transponder, but none of the tested cars actually check this data. Some cars use Hitag2 for keyless ignition systems, which are more vulnerable because they lack a physical key. This section summarizes some of the weaknesses we found during our practical experiments. Especially, Section 7.4 shows the implications of the attack described in Section 5.3 when the transponder uses a predictable password. Section 7.5 describes how to circumvent identifier white-listing. This is an additional security mechanism which is often used in vehicle immobilizers.

7.1 Weak random number generators

From the cars we tested, most pseudo-random number generators (PRNG) use the time as a seed. The time intervals do not have enough precision. Multiple authentication attempts within a time frame of one second get the same random number. Even worse, we came across two cars which have a PRNG with dangerously low entropy. The first one, a French car (A), produces nonces with only 8 bits of entropy, by setting 24 of the 32 bits always to zero as shown in Figure 17.

Origin	Message	Description
CAR	18	authenticate
TAG	39 0F 20 10	<i>id</i>
CAR	0A 00 00 00 23 71 90 14	{ n_R }{ <i>a_R</i> }
TAG	27 23 F8 AF	{ <i>a_T</i> }
CAR	18	authenticate
TAG	39 0F 20 10	<i>id</i>
CAR	56 00 00 00 85 CA 95 BA	{ n_R }{ <i>a_R</i> }
TAG	38 07 50 C5	{ <i>a_T</i> }

Figure 17: Random numbers generated by car A

Another French car (B), produced random looking nonces, but in fact, the last nibble of each byte was determined by the last nibble of the first byte. A subset of these nonces are shown in Figure 18.

{n _R }	{a _R }
20 D1 0B 08	56 36 F3 66
70 61 1B 58	1B 18 F3 38
B0 A1 5B 98	1E 94 62 3A
DO 41 FB B8	01 3B 54 10
25 1A 3C AD	15 88 5E 19
05 7A 9C 8D	F7 4D F7 70
C5 3A 5C 4D	30 B1 4A D4
E5 DA FC 6D	D8 BD 79 C3

Figure 18: Random numbers generated by car B

7.2 Low entropy keys

Some cars have repetitive patterns in their keys which makes them vulnerable to dictionary attacks. Recent models of a Korean car (C) use the key with the lowest entropy we came across. It tries to access the transponder in password mode as well as in crypto mode. For this it uses the default password MIKR and a key of the form 0xFFFF*****FF as shown in Figure 19.

Origin	Message	Description
CAR	18	authenticate
TAG	E4 13 05 1A	<i>id</i>
CAR	4D 49 4B 52	password = MIKR
CAR	18	authenticate
TAG	E4 13 05 1A	<i>id</i>
CAR	DA 63 3D 24 A7 19 07 12	{n _R }{a _R }
TAG	EC 2A 4B 58	{a _T }

Figure 19: Car C authenticates using the default password and secret key 0xFFFF814632FF

7.3 Readable keys

Section 5.1 shows how to recover the memory dump of a Hitag2 transponder. Almost all makes protect the secret key against read operations by setting the bits of the configuration in such a way that block one and two are not readable. Although there are some exceptions. For example, experiments show that most cars from a French manufacturer have *not* set this protection bit. This enables an attacker to recover the secret key in an instant. Even more worrying, many of these cars have the optional feature to use a remote key-less entry system which have a much wider range and are therefore more vulnerable to wireless attacks. The combination

of a transponder that is wirelessly accessible over a distance of several meters and a non protected readable key is most worrying.

7.4 Predictable transponder passwords

The transponder password is encrypted and sent in the transponder answer *a_T* of the authentication protocol. This is an additional security mechanism of the Hitag2 protocol apart from the cryptographic algorithm. Besides the fact that the transponder proves knowledge of the secret key, it sends its password encrypted. In general it is good to have some fall back scenario and countermeasure if the used cryptosystem gets broken. Section 5.3 demonstrates how to recover the secret key from a vehicle. But to start the engine, it is necessary to know the transponder password as well. Experiments show that at least half of the cars we tested on use default or predictable passwords.

7.5 Identifier pickpocketing

The first generation of vehicle immobilizers were not able to compute any cryptographic operations. These transponders were simply transmitting a constant (unique) identifier over the RF channel. Legitimate transponder identifiers were white-listed by the vehicle and only those transponders in the white-list would enable the engine to start. Most immobilizer units in cars still use such white-listing mechanism, which is actually encouraged by NXP. These cars would only attempt to authenticate transponders in their white-list. This is an extra obstacle for an attacker, namely recovering a genuine identifier from the victim before being able to execute any attack. There are (at least) two ways for an adversary to wirelessly pickpocket a Hitag2 identifier:

- One option is to use the low-frequency (LF) interface to wirelessly pickpocket the identifier from the victim's key. This can be done within proximity distance and takes only a few milliseconds. According to the Hitag2 datasheet [36], the communication range of a transponder is up to one meter. Although, Hitag2 transponders embedded into car keys are optimized for size and do not achieve such a communication distance. However, an adversary can use tuned equipment with big antennas that ignore radiation regulations (e.g., [17]) in order to reach a larger reading distance. Many examples in the literature show the simplicity and low-cost of such a setup [24, 30, 31, 43].
- Another option is to use the wide range ultra-high frequency (UHF) interface. For this an adversary needs to eavesdrop the transmission of a hybrid

Hitag2 transponder [39] when the victim presses a button on the remote (e.g. to close the doors). Most keyless entry transponders broadcast their identifier in the clear on request (see for example [39]).

With respect to the LF interface, the UHF interface has a much wider transmission range. As shown in [18] it is not hard to eavesdrop such a transmission from a distance of 100 meters. From a security perspective, the first generation Hitag2 transponders have a physical advantage over the hybrid transponders since they only support the LF interface.

8 Mitigation

This section briefly discusses a simple but effective authentication protocol for car immobilizers and it also describes a number of mitigating measures for the attacks proposed in Section 5. For more details we refer the reader to [1, 9].

First of all we emphasize that it is important for the automotive industry to migrate from weak proprietary ciphers to a peer-reviewed one such as AES [15], used in cipher block chaining mode (CBC). A straightforward mutual authentication protocol is sketched in Figure 20. The random nonces n_R , n_T , secret key k and transponder password PWD_T should be at least 128 bits long. Comparable schemes are proposed in the literature [32, 33, 46, 48, 49].

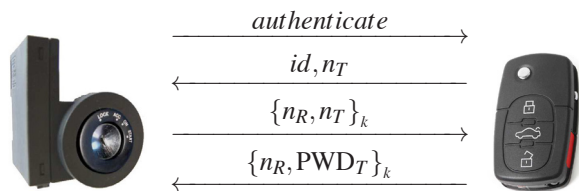


Figure 20: Immobilizer authentication protocol using AES

There are already in the market immobilizer transponders which implement AES like the ATA5795[2] from Atmel and the Hitag AES / Pro[37] from NXP. It should be noted that, although they use a peer-reviewed encryption algorithm, their authentication protocol is still proprietary and therefore lacks public and academic scrutiny.

In order to reduce the applicability of our cryptographic attack, the automotive industry could consider the following measures. This attack is the most sensitive as it does not require access to the car key. These countermeasures should be interpreted as palliating (but not a solution) before migrating to a more secure and openly designed product.

- **Extend the transponder password**

The transponder password is an important part of the authentication protocol but grievously it has only an entropy of 24 bits. Such a password is easy to find via exhaustive search. Furthermore, as we mentioned in Section 7.4, manufacturers often deployed their cars with predictable transponder passwords. As shown in Figure 8, there are four pages available of user defined memory in a Hitag2 transponder. These could be used to extend the transponder password with 128 bits of random data to increase its entropy. This implies that an adversary needs to get access to the transponder's memory before being able to steal a car.

- **Delay authentication after failure**

The cryptographic car-only attack explained in Section 5.3 requires several authentication attempts to reduce the computational complexity. Extending the time an adversary needs to gather these traces increases the risk of being caught. To achieve this, the immobilizer introduces a pause before re-authenticating that grows incrementally or exponentially with the number of sequential incorrect authentications. An interesting technique to implement such a countermeasure is proposed in [40]. The robustness, availability and usability of the product is affected by this delay, but it increases the attack time considerably and therefore reduces the risk of car theft.

Besides these measures, it is important to improve the pseudo-random number generator in the vehicles which is used to generate reader nonces. Needless to say, the same applies to cryptographic keys and transponder passwords. NIST has proposed a statistical test suite which can be used to verify the quality of a pseudo-random number generator [41].

9 Conclusions

We have found many serious vulnerabilities in the Hitag2 and its usage in the automotive industry. In particular, Hitag2 allows replaying reader data to the transponder; provides an unlimited keystream oracle and uses only one low-entropy nonce to randomize a session. These weaknesses allow an adversary to recover the secret key within seconds when wireless access to the car and key is available. When only communication with the car is possible, the adversary needs less than six minutes to recover the secret key. The cars we tested use identifier white-listing. To circumvent this, the adversary first needs to obtain a valid transponder id by other means e.g., eavesdrop it when the victim locks the doors. This

UHF transmission can be intercepted from a distance of 100 meters [18]. We have executed all our attacks (from Section 5) in practice within the claimed attack times. We have experimented with more than 20 vehicles of various makes and models and found also several implementation weaknesses.

In line with the principle of responsible disclosure, we have notified the manufacturer NXP six months before disclosure. We have constructively collaborated with NXP, discussing mitigating measures and giving them feedback to help improve the security of their products.

10 Acknowledgments

The authors would like to thank Bart Jacobs for his firm support in the background. We are also thankful to E. Barendsen, L. van den Broek, J. de Bue, Y. van Dalen, E. Gouwens, R. Habraken, I. Haerkens, S. Hoppenbrouwers, K. Koster, S. Meeuwsen, J. Reule, J. Reule, I. Roggema, L. Spix, C. Terheggen, M. Vaal, S. Vernooij, U. Zeitler, B. Zwanenburg, and those who prefer to remain anonymous for (bravely) volunteering their cars for our experiments.

References

- [1] Ross J. Anderson. *Security Engineering: A guide to building dependable distributed systems*. Wiley, 2010.
- [2] Atmel. Embedded avr microcontroller including rf transmitter and immobilizer lf functionality for remote keyless entry - ATA5795, 2010.
- [3] Steve Babbage. A space/time tradeoff in exhaustive search attacks on stream ciphers. In *European Convention on Security and Detection*, volume 408 of *Conference Publications*, pages 161–166. IEEE Computer Society, 1995.
- [4] Josep Balasch, Benedikt Gierlichs, Roel Verdult, Lejla Batina, and Ingrid Verbauwhede. Power analysis of Atmel CryptoMemory - recovering keys from secure EEPROMs. In *12th Cryptographers' Track at the RSA Conference (CT-RSA 2012)*, volume 7178 of *Lecture Notes in Computer Science*, pages 19–34. Springer-Verlag, 2012.
- [5] Alex Biryukov, Ilya Kizhvatov, and Bin Zhang. Cryptanalysis of the Atmel cipher in SecureMemory, CryptoMemory and CryptoRF. In *9th Applied Cryptography and Network Security (ACNS 2011)*, pages 91–109. Springer-Verlag, 2011.
- [6] Alex Biryukov, Sourav Mukhopadhyay, and Palash Sarkar. Improved time-memory trade-offs with multiple data. In *13th International Workshop on Selected Areas in Cryptography (SAC 2006)*, volume 3897 of *Lecture Notes in Computer Science*, pages 110–127. Springer-Verlag, 2006.
- [7] Alex Biryukov and Adi Shamir. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In *6th International Conference on the Theory and Application of Cryptology and Information Security, Advances in Cryptology (ASIACRYPT 2000)*, volume 1976 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 2000.
- [8] Andrey Bogdanov. Linear slide attacks on the KeeLoq block cipher. In *Information Security and Cryptology (INSCRYPT 2007)*, volume 4990 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2007.
- [9] Andrey Bogdanov and Christof Paar. On the security and efficiency of real-world lightweight authentication protocols. In *1st Workshop on Secure Component and System Identification (SECSI 2008)*. ECRYPT, 2008.
- [10] Stephen C. Bono, Matthew Green, Adam Stubblefield, Ari Juels, Aviel D. Rubin, and Michael Szydlo. Security analysis of a cryptographically-enabled RFID device. In *14th USENIX Security Symposium (USENIX Security 2005)*, pages 1–16. USENIX Association, 2005.
- [11] Johan Borst, Bart Preneel, Joos Vandewalle, and Joos V. On the time-memory tradeoff between exhaustive key search and table precomputation. In *19th Symposium in Information Theory in the Benelux*, pages 111–118, 1998.
- [12] Nicolas T. Courtois. The dark side of security by obscurity - and cloning MIFARE Classic rail and building passes, anywhere, anytime. In *4th International Conference on Security and Cryptography (SECRYPT 2009)*, pages 331–338. INSTICC Press, 2009.
- [13] Nicolas T. Courtois, Gregory V. Bard, and David Wagner. Algebraic and slide attacks on KeeLoq. In *15th International Workshop on Fast Software Encryption (FSE 2000)*, volume 5086 of *Lecture Notes in Computer Science*, pages 97–115. Springer-Verlag, 2008.

- [14] Nicolas T. Courtois, Sean O’Neil, and Jean-Jacques Quisquater. Practical algebraic attacks on the Hitag2 stream cipher. In *12th Information Security Conference (ISC 2009)*, volume 5735 of *Lecture Notes in Computer Science*, pages 167–176. Springer-Verlag, 2009.
- [15] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer-Verlag, 2002.
- [16] Gerhard de Koning Gans, Jaap-Henk Hoepman, and Flavio D. Garcia. A practical attack on the MI-FARE Classic. In *8th Smart Card Research and Advanced Applications Conference (CARDIS 2008)*, volume 5189 of *Lecture Notes in Computer Science*, pages 267–282. Springer-Verlag, 2008.
- [17] Federal Communications Commission FCC. Guidelines for evaluating the environmental effects of radio frequency radiation. Technical report, Federal Communications Commission FCC, April 2009.
- [18] Aurélien Francillon, Boris Danev, and Srdjan Čapkun. Relay attacks on passive keyless entry and start systems in modern cars. In *18th Network and Distributed System Security Symposium (NDSS 2011)*. The Internet Society, 2011.
- [19] Flavio D. Garcia, Gerhard de Koning Gans, Ruben Muijers, Peter van Rossum, Roel Verdult, Ronny Wichers Schreur, and Bart Jacobs. Dismantling MI-FARE Classic. In *13th European Symposium on Research in Computer Security (ESORICS 2008)*, volume 5283 of *Lecture Notes in Computer Science*, pages 97–114. Springer-Verlag, 2008.
- [20] Flavio D. Garcia, Gerhard de Koning Gans, and Roel Verdult. Exposing iClass key diversification. In *5th USENIX Workshop on Offensive Technologies (USENIX WOOT 2011)*, pages 128–136, San Francisco, CA, USA, 2011. USENIX Association.
- [21] Flavio D. Garcia, Gerhard de Koning Gans, Roel Verdult, and Milosch Meriac. Dismantling iClass and iClass Elite. In *17th European Symposium on Research in Computer Security (ESORICS 2012)*, *Lecture Notes in Computer Science*. Springer-Verlag, 2012.
- [22] Flavio D. Garcia, Peter van Rossum, Roel Verdult, and Ronny Wichers Schreur. Wirelessly pickpocketing a mifare classic card. In *30th IEEE Symposium on Security and Privacy (S&P 2009)*, pages 3–15. IEEE Computer Society, 2009.
- [23] Flavio D. Garcia, Peter van Rossum, Roel Verdult, and Ronny Wichers Schreur. Dismantling SecureMemory, CryptoMemory and CryptoRF. In *17th ACM Conference on Computer and Communications Security (CCS 2010)*, pages 250–259. ACM/SIGSAC, 2010.
- [24] Gerhard P. Hancke. Practical attacks on proximity identification systems (short paper). In *27th IEEE Symposium on Security and Privacy (S&P 2006)*, pages 328–333. IEEE Computer Society, 2006.
- [25] Martin E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980.
- [26] Motoki Hirano, Mikio Takeuchi, Takahisa Tomoda, and Kin-Ichiro Nakano. Keyless entry system with radio card transponder. *IEEE Transactions on Industrial Electronics*, 35:208–216, 1988.
- [27] Sebastiaan Indestege, Nathan Keller, Orr Dunkelman, Eli Biham, and Bart Preneel. A practical attack on KeeLoq. In *27th International Conference on the Theory and Application of Cryptographic Techniques, Advances in Cryptology (EUROCRYPT 2008)*, volume 4965 of *Lecture Notes in Computer Science*, pages 1–8. Springer-Verlag, 2008.
- [28] Markus Kasper, Timo Kasper, Amir Moradi, and Christof Paar. Breaking KeeLoq in a flash: on extracting keys at lightning speed. In *2nd International Conference on Cryptology in Africa, Progress in Cryptology (AFRICACRYPT 2009)*, volume 5580 of *Lecture Notes in Computer Science*, pages 403–420. Springer-Verlag, 2009.
- [29] Keyline. Transponder guide. http://www.keyline.it/files/884/transponder_guide_16729.pdf, 2012.
- [30] Ziv Kfir and Avishai Wool. Picking virtual pockets using relay attacks on contactless smartcard. In *1st International Conference on Security and Privacy for Emerging Areas in Communications Networks (SecureComm 2005)*, pages 47–58. IEEE Computer Society, 2005.
- [31] Ilan Kirschenbaum and Avishai Wool. How to build a low-cost, extended-range RFID skimmer. In *15th USENIX Security Symposium (USENIX Security 2006)*, pages 43–57. USENIX Association, 2006.

- [32] Kerstin Lemke, Ahmad-Reza Sadeghi, and Christian Stble. An open approach for designing secure electronic immobilizers. In *Information Security Practice and Experience (ISPEC 2005)*, volume 3439 of *Lecture Notes in Computer Science*, pages 230–242. Springer-Verlag, 2005.
- [33] Kerstin Lemke, Ahmad-Reza Sadeghi, and Christian Stuble. Anti-theft protection: Electronic immobilizers. *Embedded Security in Cars*, pages 51–67, 2006.
- [34] Karsten Nohl. Immobilizer security. In *8th International Conference on Embedded Security in Cars (ESCAR 2010)*, 2010.
- [35] Karsten Nohl, David Evans, Starbug, and Henryk Plötz. Reverse engineering a cryptographic RFID tag. In *17th USENIX Security Symposium (USENIX Security 2008)*, pages 185–193. USENIX Association, 2008.
- [36] Transponder IC, Hitag2. Product Data Sheet, Nov 2010. NXP Semiconductors.
- [37] Hitag pro. Product Data Sheet, 2011. NXP Semiconductors.
- [38] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In *23rd International Cryptology Conference, Advances in Cryptology (CRYPTO 2003)*, volume 2729 of *Lecture Notes in Computer Science*, pages 617–630. Springer-Verlag, 2003.
- [39] Security transponder plus remote keyless entry – Hitag2 plus, PCF7946AT. Product Profile, Jun 1999. Philips Semiconductors.
- [40] Amir Rahmati, Mastooreh Salajegheh, Dan Holcomb, Jacob Sorber, Wayne P. Burleson, and Kevin Fu. TARDIS: Time and remanence decay in SRAM to implement secure protocols on embedded devices without clocks. In *21st USENIX Security Symposium (USENIX Security 2012)*. USENIX Association, 2012.
- [41] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, and San Vo. A statistical test suite for the validation of random number generators and pseudo random number generators for cryptographic applications. *NIST Special Publication*, pages 800–822, 2001.
- [42] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *12th International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer-Verlag, 2009.
- [43] Frank Stajano and Ross J. Anderson. The resurrecting duckling: Security issues for ad-hoc wireless networks. In *7th International Workshop on Security Protocols (WSP 2000)*, volume 1796 of *Lecture Notes in Computer Science*, pages 172–182. Springer-Verlag, 2000.
- [44] Siwei Sun, Lei Hu, Yonghong Xie, and Xiangyong Zeng. Cube cryptanalysis of Hitag2 stream cipher. In *10th International Conference on Cryptology and Network Security (CANS 2011)*, volume 7092 of *Lecture Notes in Computer Science*, pages 15–25. Springer-Verlag, 2011.
- [45] Petr Štembera and Martin Novotný. Breaking Hitag2 with reconfigurable hardware. In *14th Euro-micro Conference on Digital System Design (DSD 2011)*, pages 558–563. IEEE Computer Society, 2011.
- [46] Pang-Chieh Wang, Ting-Wei Hou, Jung-Hsuan Wu, and Bo-Chiuan Chen. A security module for car appliances. *International Journal of World Academy Of Science, Engineering and Technology*, 26:155–160, 2007.
- [47] I.C. Wiener. Philips/NXP Hitag2 PCF7936/46/47/52 stream cipher reference implementation. <http://cryptolib.com/ciphers/hitag2/>, 2007.
- [48] Marko Wolf, Andre Weimerskirch, and Thomas Wollinger. State of the art: Embedding security in vehicles. *EURASIP Journal on Embedded Systems*, 2007:074706, 2007.
- [49] Jung-Hsuan Wu, Chien-Chuan Kung, Jhan-Hao Rao, Pang-Chieh Wang, Cheng-Liang Lin, and Ting-Wei Hou. Design of an in-vehicle anti-theft component. In *8th International Conference on Intelligent Systems Design and Applications (ISDA 2008)*, volume 1, pages 566–569. IEEE Computer Society, 2008.

Taking proof-based verified computation a few steps closer to practicality

Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish

The University of Texas at Austin

Abstract. We describe GINGER, a built system for unconditional, general-purpose, and nearly practical verification of outsourced computation. GINGER is based on PEPPER, which uses the PCP theorem and cryptographic techniques to implement an *efficient argument* system (a kind of interactive protocol). GINGER slashes the query size and costs via theoretical refinements that are of independent interest; broadens the computational model to include (primitive) floating-point fractions, inequality comparisons, logical operations, and conditional control flow; and includes a parallel GPU-based implementation that dramatically reduces latency.

1 Introduction

We are motivated by *outsourced computing*: cloud computing (in which clients outsource computations to remote computers), peer-to-peer computing (in which peers outsource storage and computation to each other), volunteer computing (in which projects outsource computations to volunteers' desktops), etc.

Our goal is to build a system that lets a client outsource computation verifiably. The client should be able to send a description of a computation and the input to a server, and receive back the result together with some auxiliary information that lets the client *verify* that the result is correct. For this to be sensible, the verification must be faster than executing the computation locally.

Ideally, we would like such a system to be *unconditional*, *general-purpose*, and *practical*. That is, we don't want to make assumptions about the server (trusted hardware, independent failures of replicas, etc.), we want a setup that works for a broad range of computations, and we want the system to be usable by real people for real computations in the near future.

In principle, the first two properties above have been achievable for almost thirty years, using powerful tools from complexity theory and cryptography. Interactive proofs (IPs) and probabilistically checkable proofs (PCPs) show how one entity (usually called the *verifier*) can be convinced by another (usually called the *prover*) of a given mathematical assertion—without the verifier having to fully inspect a proof [5, 6, 19, 32]. In our context, the mathematical assertion is that a given computation was carried out correctly; though the proof is as long as the computation, the theory implies—surprisingly—that the verifier need only inspect the proof in a small number of randomly-chosen locations or query the prover a relatively small number of times.

The rub has been the third property: practicality. These protocols have required expensive encoding of compu-

tations, monstrously large proofs, high error bounds, prohibitive overhead for the prover, and intricate constructions that make the asymptotically efficient schemes challenging to implement correctly.

However, a line of recent work indicates that approaches based on IPs and PCPs are closer to practicality than previously thought [21, 44, 45, 49]. More generally, there has been a groundswell of work that aims for potentially practical verifiable outsourced computation, using theoretical tools [11, 12, 20, 24, 25].

Nonetheless, these works have notable limitations. Only a handful [21, 44, 45, 49] have produced working implementations, all of which impose high costs on the verifier and prover. Moreover, their model of computation is *arithmetic circuits* over finite fields, which represent non-integers awkwardly, control flow inefficiently, and comparisons and logical operations only by degenerating to verbose *Boolean* circuits. Arithmetic circuits are well-suited to integer computations and numerical straight line computations (e.g., multiplying matrices and computing second moments), but the intersection of these two domains leaves few realistic applications.

This paper describes a built system, called GINGER, that addresses these problems, thereby taking general-purpose proof-based verified computation several steps closer to practicality. GINGER is an *efficient argument* system [37, 38]: an interactive proof system that assumes the prover to be computationally bounded. Its starting point is the PEPPER protocol [45] (which is summarized in Section 2). GINGER's contributions are as follows.

(1) GINGER *demonstrates the strength of linear commitment* (§3). This paper proves that PEPPER's commitment primitive [45], which generalizes the commitment primitive of Ishai et al. [35], is surprisingly powerful: it not only commits an untrusted entity to a function and extracts evaluations of that function (as previously shown) but also ensures that the function is linear. (The primitive embeds a *strong linearity test*.) This result sharply reduces the required number of queries (from 500 to 3) and a key error bound, and hence overhead.

(2) GINGER *supports a general-purpose programming model* (§4). Although the model does not handle looping concisely, it includes primitive floating-point quantities, inequality comparisons, logical expressions, and conditional control flow. Moreover, we have a compiler (derived from Fairplay [39]) that transforms computations expressed in a general-purpose language to an executable verifier and prover. The core technical challenge is representing computations as additions and multiplications over a finite field (as required by the verification proto-

col); for instance, “not equal” and “if/else” do not obviously map to this formalism, inequalities are problematic because finite fields are not ordered, and fractions compound the difficulties. GINGER overcomes these challenges with techniques that, while not deep, require care and detail.¹ These techniques should apply to other protocols that use arithmetic constraints or circuits.

(3) GINGER exploits parallelism to slash latency (§5). The prover can be distributed across machines, and some of its functions are implemented in graphics hardware (GPUs). Moreover, GINGER’s verifier can use a GPU for its cryptographic operations. Allowing the verifier to have a GPU models the present (many computers have GPUs) and a plausible future in which specialized hardware for cryptographic operations is common.²

We have implemented and evaluated GINGER (§6). Compared to PEPPER [45], its base, GINGER lowers network costs by 1–2 orders of magnitude (to hundreds of KB or less in our experiments). The verifier’s costs drop by multiples and possibly orders of magnitude, depending on the cost of encryption; if we model encryption as free, the verifier can gain from outsourcing when batch-verifying as few as 20 computations (down from 3900 in PEPPER). The prover’s CPU costs drop by 10–15%, which is not much, but our parallel implementation reduces latency with near-linear speedup. Computing with rational numbers in GINGER is roughly three times more expensive than computing with integers, and arithmetic constraints permit far smaller representations than a naive use of Boolean or arithmetic circuits.

Despite all of the above, GINGER is not quite ready for the big leagues. However, PEPPER and GINGER have made argument systems far more practical (in some cases improving costs by 23 orders of magnitude over a naive implementation). We are thus optimistic about ultimately achieving true practicality.

2 Problem statement and background

Problem statement. A computer V , known as the *verifier*, has a computation Ψ and some desired input x that it wants a computer P , known as the *prover*, to perform. P returns y , the purported output of the computation, and then V and P conduct an efficient interaction. This interaction should be cheaper for V than locally computing $\Psi(x)$. Furthermore, if P returned the correct answer, it should be able to convince V of that fact; otherwise, V should be able to reject the answer as incorrect, with high probability. (The converse will not hold: rejection does not imply that P returned incorrect output, only that

it misbehaved somehow.) Our goal is that this guarantee be *unconditional*: it should hold regardless of whether P obeys the protocol (given standard cryptographic assumptions about P ’s computational power). If P deviates from the protocol at any point (computing incorrectly, proving incorrectly, etc.), we call it *malicious*.

2.1 Tools

In principle, we can meet our goal using PCPs. The PCP theorem [5, 6] says that if a set of constraints is satisfiable (see below), there exists a *probabilistically checkable* proof (a PCP) and a verification procedure that accepts the proof after querying it in only a small number of locations. On the other hand, if the constraints cannot be satisfied, then the verification procedure rejects *any* purported proof, with probability at least $1 - \epsilon$.

To apply the theorem, we represent the computation as a set of quadratic constraints over a finite field. A *quadratic constraint* is an equation of degree 2 that uses additions and multiplications (e.g., $A \cdot Z_1 + Z_2 - Z_3 \cdot Z_4 = 0$). A set of constraints is *satisfiable* if the variables can be set to make all of the equations hold simultaneously; such an assignment is called a *satisfying assignment*. In our context, a set of constraints \mathcal{C} will have a designated input variable X and output variable Y (this generalizes to multiple inputs and outputs), and $\mathcal{C}(X = x, Y = y)$ denotes \mathcal{C} with variable X bound to x and Y bound to y .

We say that a set of constraints \mathcal{C} is *equivalent* to a desired computation Ψ if: for all x, y , $\mathcal{C}(X = x, Y = y)$ is satisfiable if and only if $y = \Psi(x)$. As a simple example, increment-by-1 is equivalent to the constraint set $\{Y = Z + 1, Z = X\}$. (For convenience, we will sometimes refer to a given input x and purported output y implicitly in statements such as, “If constraints \mathcal{C} are satisfiable, then Ψ executed correctly”.) To verify a computation $y = \Psi(x)$, one could in principle apply the PCP theorem to the constraints $\mathcal{C}(X = x, Y = y)$.

Unfortunately, PCPs are too large to be transferred. However, if we assume a computational bound on the prover P , then *efficient arguments* apply [37, 38]: V issues its PCP queries to P (so V need not receive the entire PCP). For this to work, P must commit to the PCP *before* seeing V ’s queries, thereby simulating a fixed proof whose contents are independent of the queries. V thus extracts a cryptographic commitment to the PCP (e.g., with a collision-resistant hash tree [40]) and verifies that P ’s query responses are consistent with the commitment.

This approach can be taken a step further: not even P has to materialize the entire PCP. As Ishai et al. [35] observe, in some PCP constructions, which they call *linear PCPs*, the PCP itself is a linear function: the verifier submits queries to the function, and the function’s outputs serve as the PCP responses. Ishai et al. thus design a *linear commitment primitive* in which P can commit to

¹We elide some of these details for space; they are documented in a longer version of this paper [46].

²One may wonder why, if the verifier has this hardware, it needs to outsource. GPUs are amenable only to certain computations (which include the cryptographic underpinnings of GINGER).

a linear function (the PCP) and V can submit function inputs (the PCP queries) to P , getting back outputs (the PCP responses) as if P itself were a fixed function.

PEPPER [45] refines and implements the outline above. In the rest of the section, we summarize the linear PCPs that PEPPER incorporates, give an overview of PEPPER, and provide formal definitions. Additional details are in Appendix A.1.

2.2 Linear PCPs, applied to verifying computations

Imagine that V has a desired computation Ψ and desired input x , and somehow obtains purported output y . To use PCP machinery to check whether $y = \Psi(x)$, V compiles Ψ into equivalent constraints \mathcal{C} , and then asks whether $\mathcal{C}(X = x, Y = y)$ is satisfiable, by consulting an *oracle* π : a fixed function (that depends on \mathcal{C}, x, y) that V can query. A *correct* oracle π is the proof (or PCP); V should accept a correct oracle and reject an incorrect one.

A correct oracle π has three properties. First, π is a *linear function*, meaning that $\pi(a) + \pi(b) = \pi(a + b)$ for all a, b in the domain of π . A linear function $\pi: \mathbb{F}^n \rightarrow \mathbb{F}$ is determined by a vector w ; i.e., $\pi(a) = \langle a, w \rangle$ for all $a \in \mathbb{F}^n$. Here, \mathbb{F} is a finite field, and $\langle a, b \rangle$ denotes the inner (dot) product of two vectors a and b . The parameter n is the size of w ; in general, n is quadratic in the number of variables in \mathcal{C} [5], but we can sometimes tailor the encoding of w to make n smaller [45].

Second, one set of the entries in w must be a redundant encoding of the other entries. Third, w encodes the actual satisfying assignment to $\mathcal{C}(X = x, Y = y)$.

A surprising aspect of PCPs is that each of these properties can be tested by making a small number of queries to π ; if π is constructed incorrectly, the probability that the tests pass is upper-bounded by $\epsilon > 0$. A key test for us—we return to it in Section 3—is the *linearity test* [16]: V randomly selects q_1 and q_2 from \mathbb{F}^n and checks if $\pi(q_1) + \pi(q_2) = \pi(q_1 + q_2)$. The other two PCP tests are the *quadratic correction test* and the *circuit test*.

The completeness and soundness properties of linear PCPs are defined in Section 2.4. A detailed explanation of why the mechanics above satisfy those properties is outside our scope but can be found in [5, 13, 35, 45].

2.3 Our base: PEPPER

We now walk through the three phases of PEPPER [45], which is depicted in Figure 1. The approach is to compose a *linear PCP* and a *linear commitment primitive* that forces the prover to act like an oracle.

Specify and compute. V transforms its desired computation, Ψ , into a set of equivalent constraints, \mathcal{C} . V sends Ψ (or \mathcal{C}) to P , or P may come with them installed.

To gain from outsourcing, V must amortize the costs of compiling Ψ to \mathcal{C} and generating queries. Thus, V verifies computations in batches [45] (although they need not be

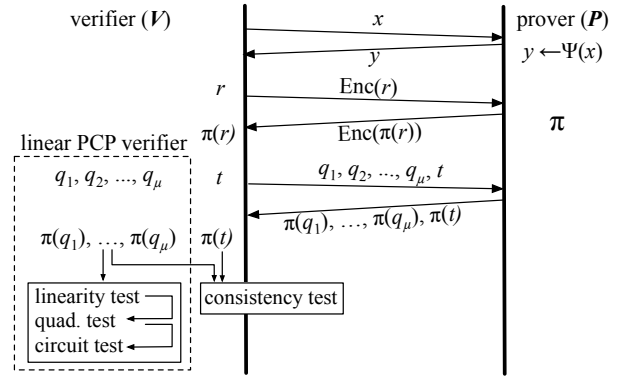


Figure 1—The PEPPER protocol [45], which is GINGER’s base. Though not depicted, many of the protocol steps happen in parallel, to facilitate batching.

executed in a batch). A *batch* (of size β) refers to a set of computations in which Ψ is the same but the inputs are different; a member of the batch is called an *instance*. In the protocol, V has inputs x_1, \dots, x_β that it sends to P (not necessarily all at once), which returns y_1, \dots, y_β ; for each instance i , y_i is supposed to equal $\Psi(x_i)$.

For each instance i , an honest P stores a proof vector w_i that encodes a satisfying assignment to $\mathcal{C}(X = x_i, Y = y_i)$; w_i is constructed as described in Section 2.2. Being a vector, w_i can also be regarded as a linear function π_i —or an oracle of the kind described above.

Extract commitment. V cannot inspect $\{\pi_i\}$ directly (they are functions; written out, they would have an entry for each value in a huge domain). Instead, V extracts a *commitment* to each π_i . To do so, V randomly generates a *commitment vector* $r \in \mathbb{F}^n$. V then homomorphically encrypts each entry of r under a public key pk to get a vector $\text{Enc}(pk, r) = (\text{Enc}(pk, r_1), \text{Enc}(pk, r_2), \dots, \text{Enc}(pk, r_n))$. We emphasize that $\text{Enc}(\cdot)$ need not be fully homomorphic encryption [27] (which remains unfeasibly expensive); PEPPER uses ElGamal [23, 45].

V sends $(\text{Enc}(pk, r), pk)$ to P . If P is honest, then π_i is linear, so P can use the homomorphism of $\text{Enc}(\cdot)$ to compute $\text{Enc}(pk, \pi_i(r))$ from $\text{Enc}(pk, r)$, without learning r . P replies with $(\text{Enc}(pk, \pi_1(r)), \dots, \text{Enc}(pk, \pi_\beta(r)))$, which is P ’s commitment to $\{\pi_i\}$. V then decrypts to get $(\pi_1(r), \dots, \pi_\beta(r))$.

Verify. V now generates PCP queries $q_1, \dots, q_\mu \in \mathbb{F}^n$, as described in Section 2.2. V sends these queries to P , along with a *consistency query* $t = r + \sum_{j=1}^{\mu} \alpha_j \cdot q_j$, where each α_j is randomly chosen from \mathbb{F} (here, \cdot represents scalar multiplication).

For ease of exposition, we focus on a single proof π_i ; however, the following steps happen β times in parallel, using the same queries for each of the β instances. If P is honest, it returns $(\pi_i(q_1), \dots, \pi_i(q_\mu), \pi_i(t))$. V checks that $\pi_i(t) = \pi_i(r) + \sum_{j=1}^{\mu} \alpha_j \cdot \pi_i(q_j)$; this is known as

the *consistency test*. If P is honest, then this test passes, by the linearity of π . Conversely, if this test passes then, *regardless* of P 's honesty, V can treat P 's responses as the output of an oracle (this is shown in previous work [35, 45]). Thus, V can use $\{\pi_i(q_1), \dots, \pi_i(q_\mu)\}$ in the PCP tests described in Section 2.2.

2.4 PCPs and arguments defined more formally

The definitions of PCPs [5, 6] and argument systems [19, 32] below are borrowed from [35, 45].

A *PCP protocol* with soundness error ϵ includes a probabilistic polynomial time verifier V that has access to a constraint set \mathcal{C} . V makes a constant number of queries to an oracle π . This process has the following properties:

- **PCP Completeness.** If \mathcal{C} is satisfiable, then there exists a linear function π such that, after V queries π , $\Pr\{V \text{ accepts } \mathcal{C} \text{ as satisfiable}\} = 1$, where the probability is over V 's random choices.
- **PCP Soundness.** If \mathcal{C} is not satisfiable, then $\Pr\{V \text{ accepts } \mathcal{C} \text{ as satisfiable}\} < \epsilon$ for *all* purported proof functions $\tilde{\pi}$.

An argument (P, V) with soundness error ϵ comprises P and V , two probabilistic polynomial time (PPT) entities that take a set of constraints \mathcal{C} as input and provide:

- **Argument Completeness.** If \mathcal{C} is satisfiable and P has access to a satisfying assignment z , then the interaction of $V(\mathcal{C})$ and $P(\mathcal{C}, z)$ makes $V(\mathcal{C})$ accept \mathcal{C} 's satisfiability, regardless of V 's random choices.
- **Argument Soundness.** If \mathcal{C} is not satisfiable, then for every malicious PPT P^* , the probability over V 's random choices that the interaction of $V(\mathcal{C})$ and $P^*(\mathcal{C})$ makes $V(\mathcal{C})$ accept \mathcal{C} as satisfiable is less than ϵ .

3 Protocol refinements in GINGER

In principle, PEPPER solves the problem of verified computation. The reality is less attractive: PEPPER's computational burden is high, its network costs are absurd, and its applicability is limited (to straight line numerical computations). Our system, GINGER, mitigates these issues: it lowers costs through protocol refinements (presented in this section), and it applies to a much wider class of computations (as we discuss in Section 4).

GINGER's refinements eliminate many queries, by relying on a new analysis of the base commitment primitive. To motivate this analysis, we note that there is something seemingly redundant in the base machinery (see Figure 1): why does the linear PCP require a linearity test (§2.2) if an honest prover depends on the linearity of its function π to pass the linear commitment protocol's consistency test (§2.3)? Can we remove one of these tests, or combine them somehow? The reason that

PEPPER appears to need both tests is that their guarantees are (so far) subtly different:

- *Consistency test* (§2.3): First, an honest prover is guaranteed to pass this test. Second, if the prover—even a cheating one—passes this test, then it is very likely bound to *some* function (as shown in [35, 45]).
- *Linearity test* (§2.2): This test is needed in case the prover cheats—it establishes that π is linear (as required by the rest of the PCP protocol). More accurately, if π is far from being linear, the test is somewhat likely to catch that case.

Yet, it seems unsatisfying that both tests are required when composing linear commitment and the linear PCP: can a prover really pass the consistency test systematically with a function that the linearity test would reject? In fact, our intuitive dissatisfaction is well-founded: this paper proves that the commitment primitive (which includes the consistency test) is far stronger than the linearity test. Put simply, even a cheating prover is very likely bound to a function that is linear, or almost so.

Practically, this result saves query generation and response costs. For one thing, we can eliminate linearity tests from the protocol. More significantly, we eliminate *amplification*: PEPPER needed to repeat the protocol to turn the linearity test's guarantee of "somewhat likely" into "very likely". In contrast, our result already gives a guarantee of "very likely", so no repetition is required.

More broadly, this result means that the commitment primitive is considerably more powerful than was realized—it efficiently commits an untrusted entity to a linear function and extracts evaluations of that function—and may apply elsewhere.

Details. The protocol refinements are rooted in a strengthened soundness analysis. *Soundness error* (for example, ϵ in Section 2.4) refers to the probability that a protocol or test succeeds when the condition that it is verifying or testing is actually false. The ideal is to have a small upper-bound on soundness error.

The soundness of the PCP protocol in Section 2.2 and Appendix A.1 is connected to the soundness of linearity testing [16]. Specifically, the base analysis proves that if the prover returns $y \neq \Psi(x)$, then the prover survives all tests (linearity, quadratic correction, circuit) with probability less than $7/9$ (requiring ρ runs to make $(7/9)^\rho$ small). The $7/9$ derives from a result [8] that if the proof oracle is not "somewhat close" to linear, then the linearity test passes with probability $< 7/9$ (though fascinating, this result is inconveniently weak in our context).

Our analysis, detailed in Appendix A.2, establishes that the commitment protocol binds the prover to a function that is extremely close to linear (otherwise, the prover could break the semantic security of the homo-

	PEPPER [45]	GINGER
PCP encoding size (n)	$s^2 + s$, in general	$s^2 + s$, in general
V's per-instance CPU costs		
Issue commit queries	$(e + 2c) \cdot n/\beta$	$(e + 2c) \cdot n/\beta$
Process commit responses	d	d
Issue PCP queries	$\rho \cdot (\chi \cdot f + \ell' \cdot f + 5c) \cdot n/\beta$	$(\chi \cdot f + \ell \cdot f + 2c) \cdot n/\beta$
Process PCP responses	$\rho \cdot (2\ell' + x + y) \cdot f$	$(2\ell + x + y) \cdot f$
P's per-instance CPU costs		
Issue commit responses	$h \cdot n$	$h \cdot n$
Issue PCP responses	$(\rho \cdot \ell' + 1) \cdot f \cdot n$	$(\ell + 1) \cdot f \cdot n$
Network cost (per instance)	$((\rho \cdot \ell' + 1) \cdot p + \xi) \cdot n/\beta$	$((\ell + 1) \cdot p + \xi) \cdot n/\beta$
PCP soundness error	$(7/9)^\rho = 2.3 \cdot 10^{-8}$	$\kappa = 2.6 \cdot 10^{-6}$
Overall soundness error	$2.4 \cdot 10^{-8}$	$4.5 \cdot 10^{-6}$

Figure 2—High-order costs and error in GINGER, compared to its base (PEPPER [45]), for a computation represented as χ constraints over s variables (§2.1). The soundness error depends on field size (Appendix A.2); the table assumes $|\mathbb{F}| = 2^{128}$. Many of the cryptographic costs enter through the commitment protocol (see Section 2.3 or Figure 12); Section 6 quantifies the parameters. The “PCP” row include the consistency query and check. The network costs slightly underestimate by not including query responses.

morphic encryption used by GINGER and PEPPER). This results in the PCP soundness error improving from $7/9$ to κ , where $\kappa \approx 4\sqrt[6]{1/|\mathbb{F}|}$; this analysis does not depend on linearity tests, so they can be dropped.

The soundness error is somewhat low by cryptographic standards, but in practice, a failure rate (when the prover is malicious) of 1 in 200,000 is reasonable.

A further optimization. GINGER reuses some queries across the quadratic correction and circuit tests; this refinement is detailed and justified in Appendix A.3.

Savings. Most significantly, V can take advantage of the lower soundness error to run $\rho = 1$ instead of $\rho = 70$ repetitions of the PCP protocol. Also, per repetition, V 's work to generate pseudorandom queries decreases by $3/5$ ($2/5$ coming from the elimination of linearity tests and $1/5$ from reusing queries). These gains are depicted in Figure 2, most notably in the reduction from $\rho \cdot \ell' \approx 500$ to $\ell = 3$ total PCP queries.

The total savings for the verifier depend on the relative cost of pseudorandom number generation (encapsulated by c) and encryption (encapsulated by e). These savings show up in β^* , the minimum batch size (§2.3) at which V gains from outsourcing. As shown in Section 6.1, the reduction in β^* can be several orders of magnitude (when e is small). Finally, taking $|p| = 128$ bits and $|\xi| = 2 \cdot 1024$ bits, the savings in network costs are 1–2 orders of magnitude (holding β constant).

4 Broadening the space of computations

GINGER extends to computations over floating-point fractional quantities and to a restricted general-purpose programming model that includes inequality tests, log-

ical expressions, conditional branching, etc. To do so, GINGER maps computations to the constraint-over-finite-field formalism (§2.1), and thus the core protocol in Section 3 applies. In fact, our techniques³ apply to the many protocols that use the constraint formalism or arithmetic circuits. Moreover, we have implemented a compiler (derived from Fairplay's [39]) that transforms high-level computations first into constraints and then into verifier and prover executables.

The challenges of representing computations as constraints over finite fields include: the “true answer” to the computation may live outside of the field; sign and ordering in finite fields interact in an unintuitive fashion; and constraints are simply equations, so it is not obvious how to represent comparisons, logical expressions, and control flow. To explain GINGER's solutions, we first present an abstract framework that illustrates how GINGER broadens the set of computations soundly and how one can apply the approach to further computations.

Framework to map computations to constraints. To map a computation Ψ over some domain D (such as the integers, \mathbb{Z} , or the rationals, \mathbb{Q}) to equivalent constraints over a finite field, the programmer or compiler performs three steps, as illustrated and described below:

$$\Psi \text{ over } D \xrightarrow{(C1)} \Psi \text{ over } U \xrightarrow{(C2)} \theta(\Psi) \text{ over } \mathbb{F}$$

$$\downarrow (C3)$$

$$\mathcal{C} \text{ over } \mathbb{F}$$

³We suspect that many of the individual techniques are known. However, when the techniques combine, the material is surprisingly hard to get right, so we will delve into (excruciating) detail, consistent with our focus on built systems.

$|x|, |y|$: # of elements in input, output
 n : # of components in linear function π (§2.2)
 s : # of variables in constraint set (§2.1)
 χ : # of constraints in constraint set (§2.1)
 $\ell = 3$: # of high-order PCP queries in GINGER (§A.2, §A.3)
 $\ell' = 7$: # of high-order PCP queries in PEPPER (§A.1)
 $\rho = 70$: # of PCP reps. in base scheme (§A.1)
 β : batch size (# of instances) (§2.3)
 e : cost of encrypting an element in \mathbb{F}
 d : cost of decrypting an encrypted element
 f : cost of multiplying in \mathbb{F}
 h : cost of ciphertext add plus multiply
 c : cost to generate 192-bit pseudorandom #
 $|p|$: length of an element in \mathbb{F}
 $|\xi|$: length of an encrypted element in \mathbb{F}

- C1 *Bound the computation.* Define a set $U \subset D$ and restrict the input to Ψ such that the output and intermediate values stay in U .
- C2 *Represent the computation faithfully in a suitable finite field.* Choose a finite field, \mathbb{F} , and a map $\theta: U \rightarrow \mathbb{F}$ such that computing $\theta(\Psi)$ over $\theta(U) \subset \mathbb{F}$ is isomorphic to computing Ψ over U . (By “ $\theta(\Psi)$ ”, we mean Ψ with all inputs and literals mapped by θ .)
- C3 *Transform the finite field version of the computation into constraints.* Write a set of constraints over \mathbb{F} that are equivalent (in the sense of Section 2.1) to $\theta(\Psi)$.

4.1 Signed integers and floating-point rationals

We now instantiate C1 and C2 for integer and rational number computations; the next section addresses C3.

Consider $m \times m$ matrix multiplication over N -bit signed integers. For step C1, each term in the output, $\sum_{k=1}^m A_{ik}B_{kj}$, has m additions of $2N$ -bit subterms so is contained in $[-m \cdot 2^{2N-1}, m \cdot 2^{2N-1}]$; this is our set U .

For step C2, take $\mathbb{F} = \mathbb{Z}/p$ (the integers mod a prime p , to be chosen shortly) and define $\theta: U \rightarrow \mathbb{Z}/p$ as $\theta(u) = u \bmod p$. Observe that θ maps negative integers to $\{\frac{p+1}{2}, \frac{p+3}{2}, \dots, p-1\}$, analogous to how processors represent negative numbers with a 1 in the most significant bit (this technique is standard [17, 50]). Of course, addition and multiplication in \mathbb{Z}/p do not “know” when their operands are negative. Nevertheless, the computation over \mathbb{Z}/p is isomorphic to the computation over U , provided that $|\mathbb{Z}/p| > |U|$ (as shown in Appendix B [46]).⁴ Thus, for the given U , we require $p > m \cdot 2^{2N}$. Note that a larger p brings larger costs (see Figure 2), so there is a three-way trade-off among p, m, N .

We now turn to rational numbers. For step C1, we restrict the inputs as follows: when written in lowest terms, their numerators are $(N_a + 1)$ -bit signed integers, and their denominators are in $\{1, 2, 2^2, 2^3, \dots, 2^{N_b}\}$. Note that such numbers are (primitive) floating-point numbers: they can be represented as $a \cdot 2^{-q}$, so the decimal point floats based on q . Now, for $m \times m$ matrix multiplication, the computation does not “leave” $U = \{a/b: |a| < 2^{N'_a}, b \in \{1, 2, 2^2, 2^3, \dots, 2^{N'_b}\}\}$, for $N'_a = 2N_a + 2N_b + \log_2 m$ and $N'_b = 2N_b$ [46, Appendix B].

For step C2, we take $\mathbb{F} = \mathbb{Q}/p$, the quotient field of \mathbb{Z}/p . Take $\theta(\frac{a}{b}) = (a \bmod p, b \bmod p)$. For any $U \subset \mathbb{Q}$, there is a choice of p such that the mapped computation over \mathbb{Q}/p is isomorphic to the original computation over \mathbb{Q} [46, Appendix B]. For our U above, $p > (m + 1)^2 \cdot 2^{4(N_a + N_b)}$ suffices.

Limitations and costs. To understand the limitations of GINGER’s floating-point representation, consider the number $a \cdot 2^{-q}$, where $|a| < 2^{N_a}$ and $|q| \leq N_q$.

To represent this number, the IEEE standard requires roughly $N_a + \log N_q$ bits [29] while GINGER requires $2 \cdot (\max(N_a, N_q) + 1)$ bits [46, Appendix B]. As a result, GINGER’s range is vastly more limited: with 64 bits, the IEEE standard can represent numbers on the order of 2^{1023} and 2^{-1022} (with $N_a = 53$ bits of precision) while 64 bits buys GINGER only numbers on the order of 2^{32} and 2^{-32} (with $N_a = 32$). Moreover, unlike the IEEE standard, GINGER does not support a division operation or rounding.

However, comparing GINGER’s floating-point representation to its *integer* representation, the extra costs are not terrible. First, the prover and verifier take an extra pass over the input and output (for implementation reasons; see Appendix B [46] for details). Second, a larger prime p is required. For example, $m \times m$ matrix multiplication with 32-bit integer inputs requires p to have at least $\log_2 m + 64$ bits; if the inputs are rationals with $N_a = N_q = 32$, then p requires $2 \log_2(m + 1) + 256$ bits. Roughly speaking, the end-to-end costs are $3 \times$ those of the integers case (see Section 6.2). Of course, the actual numbers depend on the computation. (Our compiler computes suitable bounds with static analysis.)

4.2 General-purpose program constructs

Case study: branch on order comparison. We now illustrate C3 with a case study of a computation, Ψ , that includes a less-than test and a conditional branch; pseudocode for Ψ is in Figure 3. For clarity, we will restrict Ψ to signed integers; handling rational numbers requires additional mechanisms [46, Appendix C].

How can we represent the test $x_1 < x_2$ using constraint *equations*? The solution is to use special *range constraints* that decompose a number into its bits to test whether it is in a given range; in this case, $C_<$, depicted in Figure 3, tests whether $e = \theta(x_1) - \theta(x_2)$ is in the “negative” range of \mathbb{Z}/p (see Section 4.1). Now, under the input restriction $x_1 - x_2 \in U$, $C_<$ is satisfiable if and only if $x_1 < x_2$ [46, Appendix C]. Analogously, we can construct C_{\geq} that is satisfiable if and only if $x_1 \geq x_2$.

Finally, we introduce a 0/1 variable M that encodes a choice of branch, and then arrange for M to “pull in” the constraints of that branch and “exclude” those of the other. (Note that the prover need not execute the untaken branch.) Figure 3 depicts the complete set of constraints, C_Ψ ; these constraints are satisfiable if and only if the prover correctly computes Ψ [46, Appendix C].

Logical expressions and conditionals. Besides order comparisons and if-else, GINGER can represent $==$, $\&\&$, and $||$ as constraints. An interesting case is $! =$: we can represent $Z1 != Z2$ with $\{M \cdot (Z_1 - Z_2) - 1 = 0\}$ because this constraint is satisfiable when $(Z_1 - Z_2)$ has a multiplicative inverse and hence is not zero. These constructs and others are detailed in Appendix D [46].

⁴For space, Appendices B–E appear only in the extended version [46].

$$\begin{array}{l}
\Psi : \\
\text{if } (X_1 < X_2) \\
\quad Y = 3 \\
\text{else} \\
\quad Y = 4
\end{array}
\quad
\mathcal{C}_\zeta = \left\{ \begin{array}{l} B_0(1 - B_0) = 0, \\ B_1(2 - B_1) = 0, \\ \vdots \\ B_{N-2}(2^{N-2} - B_{N-2}) = 0, \\ \theta(X_1) - \theta(X_2) - (p - 2^{N-1}) - \sum_{i=0}^{N-2} B_i = 0 \end{array} \right\}
\quad
\mathcal{C}_\Psi = \left\{ \begin{array}{l} M\{\mathcal{C}_\zeta\}, \\ M(Y - 3) = 0, \\ (1 - M)\{\mathcal{C}_{>=}\}, \\ (1 - M)(Y - 4) = 0 \end{array} \right\}$$

Figure 3—Pseudocode for our case study of Ψ , and corresponding constraints \mathcal{C}_Ψ . Ψ 's inputs are signed integers x_1, x_2 ; per steps C1 and C2 (§4.1), we assume $x_1 - x_2 \in U \subset [-2^{N-1}, 2^{N-1}]$, where $p > 2^N$. The constraints \mathcal{C}_ζ test $x_1 < x_2$ by testing whether the bits of $\theta(x_1) - \theta(x_2)$ place it in $[p - 2^{N-1}, p)$. $M\{\mathcal{C}\}$ means multiplying all constraints in \mathcal{C} by M and then reducing to degree-2.

Limitations and costs. We compile a subset of SFDL, the language of the Fairplay compiler [39]. Thus, our limitations are essentially those of SFDL; notably, loop bounds have to be known at compile time.

How efficient is our representation? The program constructs above mostly have concise constraint representations. Consider, for instance, `comp1==comp2`; the equivalent constraint set \mathcal{C} consists of the constraints that represent `comp1`, the constraints that represent `comp2`, and an additional constraint to relate the outputs of `comp1` and `comp2`. Thus, \mathcal{C} is the same size as its two components, as one would expect.

However, two classes of computations are costly. First, inequality comparisons require variables and a constraint for every bit position; see Figure 3. Second, the constraints for `if-else` and `||`, as written, seem to be degree-3; notice, for instance, the $M\{\mathcal{C}\}$ in Figure 3. To be compatible with the core protocol, these constraints must be rewritten to be degree-2 (§2.1), which carries costs. Specifically, if \mathcal{C} has s variables and χ constraints, an equivalent degree-2 representation of $M\{\mathcal{C}\}$ has $s + \chi$ variables and $2 \cdot \chi$ constraints [46, Appendix D].

5 Parallelization and implementation

Many of GINGER's remaining costs are in the cryptographic operations in the commitment protocol (see Appendix A.1). To address these costs, we distribute the prover over multiple machines, leveraging GINGER's inherent parallelism. We also implement the prover and verifier on GPUs, which raises two questions. (1) Isn't this just moving the problem? Yes, and this is good: GPUs are optimized for the types of operations that bottleneck GINGER. (2) Why do we assume that the *verifier* has a GPU? Desktops are more likely than servers to have GPUs, and the prevalence of GPUs is increasing. Also, this setup models a future in which specialized hardware for cryptographic operations is common.

Parallelization. To distribute GINGER's prover, we run multiple copies of it (one per host), each copy receiving a fraction of the batch (Section 2.3). In this configuration, the provers use the Open MPI [2] message-passing library to synchronize and exchange data.

To further reduce latency, each prover offloads work to a GPU (see also [49] for an independent study of GPU

hardware in the context of [21]). We exploit three levels of parallelism here. First, the prover performs a ciphertext operation for each component in the commitment vector (§2.3); each operation is (to first approximation) separate. Second, each operation computes two independent modular exponentiations (the ciphertext of an ElGamal encryption has two elements). Third, modular exponentiation itself admits a parallel implementation (each input is a multiprecision number encoded in multiple machine words). Thus, in our GPU implementation, a group of CUDA [1] threads computes each exponentiation.

We also parallelize the verifier's encryption work during the commitment phase (§2.3), using the approach above plus an optimization: the verifier's exponentiations are fixed base, letting us memoize intermediate squares. We implement exponentiations for the prover and verifier with the `libgpcrypto` library of `SSLShader` [36], modified to implement the memoization.

Implementation details. Our compiler consists of two stages, which a future publication will detail. The front-end compiles a subset of Fairplay's SFDL [39] to constraints; it is derived from Fairplay and is implemented in 5294 lines of Java, starting from Fairplay's 3886 lines (per [51]). The back-end transforms constraints into C++ code that implements the verifier and prover and then invokes `gcc`; this component is 1105 lines of Python code.

For efficiency, PEPPER [45] introduced specialized PCP protocols for certain computations. For some experiments we use specialized PCPs in GINGER also; in these cases we write the prover and verifier manually, which typically requires a few hundred lines of C++. Automating the compilation of specialized PCPs is future work.

The verifier and prover are separate processes that exchange data using Open MPI [2]. GINGER uses the ElGamal cryptosystem [23] with 1024-bit keys.

6 Experimental evaluation

Our evaluation answers the following questions:

- What is the effect of the protocol refinements (§3)?
- What are the costs of supporting rational numbers and the additional program structures (§4)?
- What is GINGER's speedup from parallelizing (§5)?

Figure 4 summarizes the results.

GINGER’s protocol refinements reduce per-instance network costs by 25–30× (to hundreds of KBs for the computations we study), prover CPU costs by about 10–14% (leaving them still high), and break-even batch size (β^*) by about 4×.	§6.1
With accelerated encryption GINGER breaks even from outsourcing short computations at small batch sizes; for 400×400 matrix multiplication, the verifier gains from outsourcing at a batch size of 20 (tens of seconds of computation).	§6.1
Rational arithmetic costs roughly 3× integer arithmetic under GINGER (but much more than native floating-point).	§6.2
Parallelizing results in near-linear reduction in the prover’s latency.	§6.3

Figure 4—Summary of main evaluation results.

computation (Ψ)	$O(\cdot)$	input domain (see §4.1)	size of \mathbb{F}	s	n	default	local
matrix mult.	$O(m^3)$	32-bit signed integers	128 bits	$2m^2$	m^3	$m = 200$	800 ms
matrix mult. (\mathbb{Q})	$O(m^3)$	rationals ($N_a = 32, N_b = 32$)	320 bits	$2m^2$	m^3	$m = 100$	5.90 ms
deg-2 poly. eval.	$O(m^2)$	32-bit signed integers	128 bits	m	m^2	$m = 100$	0.40 ms
deg-3 poly. eval.	$O(m^3)$	32-bit signed integers	192 bits	m	m^3	$m = 200$	160 ms
m -Hamming dist.	$O(m^2)$	32-bit unsigned	128 bits	$2m^2 + m$	$2m^3$	$m = 100$	0.90 ms
bisection method	$O(m^2)$	rationals ($N_a = 32, N_b = 5$)	320 bits	$16 \cdot (m + \mathcal{C}_<)$	$256 \cdot (m + \mathcal{C}_<)^2$	$m = 25$	180 ms

Figure 5—Benchmark computations. s is the number of constraint variables; s affects n , which is the size of V ’s queries and of P ’s linear function π (see Figure 2). Only high-order terms are reported for n . The latter two columns give our experimental defaults and the cost of local computation (i.e., no outsourcing) at those defaults. In polynomial evaluation, V and P hold a polynomial; the input is values for the m variables. The latter two computations exercise the program constructs in Section 4.2. In m -Hamming distance, V and P hold a fixed set of strings; the input is a length m string, and the output is a vector of the Hamming distance between the input and the set of strings. Bisection method refers to root-finding via bisection: both V and P hold a degree-2 polynomial in m variables, the input is two m -element endpoints that bracket a root, and the output is a small interval that contains the root.

We use six benchmark computations, summarized in Figure 5 (Appendix E [46] has details). For bisection method and degree-2 polynomial evaluation, V and P were produced by our compiler; for the other computations, we use tailored encodings (see Section 5). We implemented and analyzed other computations (e.g., edit distance and circle packing) but found that V gained from outsourcing only at implausibly large batch sizes.

Method and setup. We measure latency and computation cycles used by the verifier and the prover, and the amount of data exchanged between them. We account for the prover’s cost in per-instance terms. Because the verifier amortizes costs over a batch (§2.3), we focus on the *break-even batch size*, β^* : the batch size at which the verifier’s CPU cost from GINGER equals the cost of computing the batch locally. We measure local computation using implementations built on the GMP library (except for matrix multiplication over rationals, where we use native floating-point).

For each result that we report, we run at least three experiments and take the averages (the standard deviations are always within 5% of the means). We measure CPU time using `getrusage`, latency using PAPI’s real time counter [3], and network costs by recording the number of application-level bytes transferred.

Our experiments use a cluster at the Texas Advanced Computing Center (TACC). Each machine is configured identically and runs Linux on an Intel Xeon processor E5540 2.53 GHz with 48GB of RAM. Experiments with GPUs use machines with an NVIDIA Tesla M2070. Each

GPU has 448 CUDA cores and 6GB of memory.

Validating the cost model. We will sometimes predict β^* , V ’s costs, and P ’s costs by using our cost model (Figure 2), so we now validate this model. We run microbenchmarks to quantify the model’s parameters— e is reported in this section; Appendix E [46] quantifies the other parameters—and then compare the parameterized model to GINGER’s measured performance. GINGER’s empirical results are at most 2%–15% more than are predicted by the model. However, local computation costs about 1.2–4.0 times more than is predicted; we think that the divergence results from adverse caching effects that increase the cost of a multiplication. Thus, we expect the verifier to break even at batch sizes that are about a factor of 1.2–4.0 smaller than predicted by the model.

6.1 The effect of GINGER’s protocol refinements

We begin with $m \times m$ matrix multiplication ($m = 100, 200$) and degree-3 polynomial evaluation ($m = 100, 200$), and batch size of $\beta = 5000$. We report *per-instance* network and CPU costs: the total network and CPU costs over the batch, divided by β .

Figure 6 depicts network costs. For matrix multiplication, these are about the same as the cost to send the inputs and receive the outputs; for polynomial evaluation, these are about 10 times the size of the inputs and outputs. Also, GINGER improves on PEPPER by 20–30×.

In this experiment, GINGER’s prover incurs about 10–14% less CPU time compared to PEPPER (estimated using a cost model from [45]) but still takes tens of minutes per-instance; this is obviously a lot, but we reduce

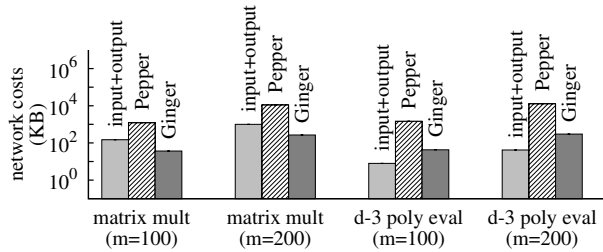


Figure 6—Per-instance network costs of GINGER and its base (PEPPER [45]), compared to the size of the inputs and outputs. At this batch size ($\beta = 5000$), GINGER’s refinements reduce per-instance network costs by a factor of 25–30 compared to PEPPER. GINGER’s network costs here are hundreds of KB or less. The y-axis is log-scaled.

	PEPPER	GINGER	
local	1.1 s	1.1 s	
CPU	β^*	13000	4100
	verifier aggregate	3.9 hr	1.3 hr
	prover aggregate	5.0 yr	1.6 yr
	prover per-instance	3.5 hr	3.3 hr
GPU	β^*	8700	2300
	verifier aggregate	2.7 hr	43.4 min
	prover aggregate	3.5 yr	320 days
	prover per-instance	3.5 hr	3.3 hr
crypto hardware	β^*	3900	20
	verifier aggregate	1.2 hr	22.3 s
	prover aggregate	1.6 yr	2.8 days
	prover per-instance	3.5 hr	3.3 hr

Figure 7—Break-even batch sizes (β^*) and predicted running times of prover and verifier at $\beta = \beta^*$, for matrix multiplication ($m = 400$), under three models of the encryption cost. The verifier’s per-instance work is not depicted because it equals the local running time, by definition of β^* . The local running time is high in part because the local implementation uses GMP.

latency by parallelizing (§6.3). For this computation and at this batch size ($\beta = 5000$), GINGER’s verifier takes a few hundreds of milliseconds per-instance, less than locally computing using our baseline of GMP.

Amortizing the verifier’s costs. Batching is both a limitation and a strength of GINGER: GINGER’s verifier *must* batch to gain from outsourcing but *can* batch to drive per-instance overhead arbitrarily low. Nevertheless, we want break-even batch sizes (β^*) to be as small as possible. But β^* mostly depends on e , the cost of encryption (Figure 2), because after our refinements the verifier’s main burden is creating $\text{Enc}(pk, r)$ (see §2.3), the cost of which amortizes over the batch.

What values of e make sense? We consider three scenarios: (1) the verifier uses a CPU for encryptions, (2) the verifier offloads encryptions to a GPU, and (3) the verifier has special-purpose hardware that can *only* perform encryptions. (See Section 5 for motivation.) Under scenario (1), we measure $e = 72.1\mu\text{s}$ on a 2.5 GHz CPU.

	mat. mult.	mat. mult. (\mathbb{Q})
local	17.6 ms	5.90 ms
verifier per-instance	17.6 ms	80.2 ms
verifier aggregate	76.1 s	5.7 min
prover per-instance	3.1 min	9.4 min
prover aggregate	9.3 days	28 days

Figure 8—Predicted running times of GINGER’s verifier and prover for matrix multiplication ($m = 100$), under integer and floating-point inputs, at $\beta = 4300$ (the break-even batch size for this computation over integers). The “local” row refers to GMP arithmetic for \mathbb{Z} and native floating-point arithmetic for \mathbb{Q} . Handling rationals costs GINGER roughly $3\times$ more than handling integers, but both are still far from native.

computation (Ψ)	# Boolean gates (est.)	# constraint vars.
m -Hamming dist.	$1.3 \cdot 10^6$	$2 \cdot 10^4$
bisection method	$3.0 \cdot 10^8$	1528

Figure 9—GINGER’s constraints compared to Boolean circuits, for m -Hamming distance ($m = 100$) and bisection method ($m = 25$). The Boolean circuits are estimated using the unmodified Fairplay [39] compiler. GINGER’s constraints are not concise but are far more so than Boolean circuits.

Under scenario (3), we take $e = 0\mu\text{s}$. What about scenario (2)? Our cost model concerns *CPU* costs, so we need an exchange rate between GPU and CPU exponentiations. We make a crude estimate: we measure the number of encryptions per second achievable on an NVIDIA Tesla M2070 (which is 180,000) and on an Intel 2.5 GHz CPU (which is 13,700), normalize by the dollar cost of the chips, and obtain that their throughput-per-dollar ratio is $1.8\times$. We thus (very conservatively) take $e = 72.1/1.8 = 40\mu\text{s}$.

We plug these three values of e into the cost model in Figure 2, set the cost under GINGER equal to the cost of local computing, and solve for β^* . The values of β^* are 4150 (CPU), 2300 (crude GPU estimate), and 20 (crypto hardware). We also use the model to predict V ’s and P ’s costs at β^* , under PEPPER and GINGER. Figure 7 summarizes. GINGER is very sensitive to the value of e because its refinements have eliminated many of the other costs. Moreover, the aggregate verifier computing time drops significantly under all three cost models. The prover’s per-instance work is mostly unaffected, but as the batch size decreases, so does its aggregate work.

6.2 Evaluating GINGER’s computational model

To understand the costs of the floating-point representation (§4.1), we compare it to two baselines: GINGER’s signed integer representation and the computation executed locally, using the CPU’s floating point unit. Our benchmark application is matrix multiplication ($m = 100$). Figure 8 details the comparison.

We also consider GINGER’s general-purpose program constructs (§4). Our baseline is *Boolean* circuits (we are

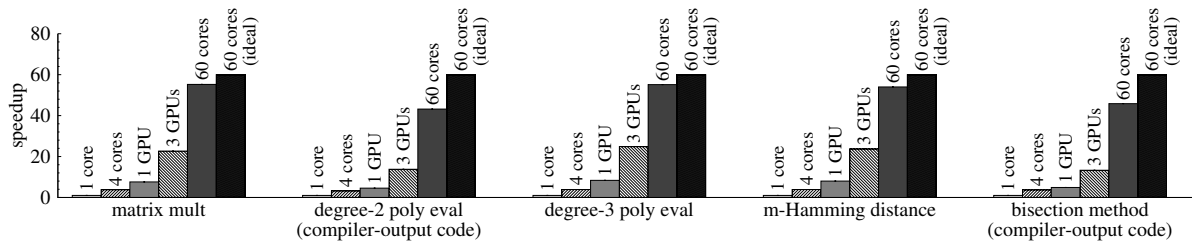


Figure 10—Latency speedup observed by GINGER’s verifier when the prover is parallelized. We run with $m = 100, \beta = 150$ for matrix multiplication and degree-3 polynomial evaluation; $m = 100, \beta = 1500$ for degree-2 polynomial evaluation; $m = 100, \beta = 15$ for m -Hamming distance; and $m = 25, \beta = 15$ for bisection method. GINGER’s prover achieves near-linear speedups except when the problem sizes are small and hence the overhead from parallelizing is significant (e.g., degree-2 polynomial evaluation).

unaware of efficient arithmetic representations of these constructs). We compare the number of Boolean circuit *gates* and the number of GINGER’s arithmetic constraint *variables*, since these determine the proving and verifying costs under the respective formalisms (see [5, 45]). Taken individually, GINGER’s constructs ($<=$, $\&\&$, etc.) are the same cost or more than those of Boolean circuits (e.g., $||$ introduces auxiliary variables). However, Boolean circuits are in general far more verbose: they represent quantities by their bits (which GINGER does only when computing inequalities). Figure 9 gives a rough end-to-end comparison.

6.3 Scalability of the parallel implementation

To demonstrate the scalability of GINGER’s parallelization, we run the prover using many CPU cores, many GPUs, and many machines. We measure end-to-end latency, as observed by the verifier. Figure 10 summarizes the results for various computations. In most cases, the speedup is near-linear.

7 Related work

A substantial body of work achieves two of our goals—it is general-purpose and practical—but it makes strong assumptions about the servers (e.g., trusted hardware). There is also a large body of work on protocols for special-purpose computation. We regard this work as orthogonal to our efforts; for a survey of this landscape, see [45]. Herein, we focus on approaches that are general-purpose and unconditional.

Homomorphic encryption and secure multi-party protocols. Homomorphic encryption (which enables computation over ciphertext) and secure multi-party protocols (in which participants compute over private data, revealing only the result [34, 39, 52]) provide only *privacy* guarantees, but one can build on them for verifiable computation. For instance, the Boneh-Goh-Nissim homomorphic cryptosystem [18] can be adapted to evaluate circuits, Groth uses homomorphic commitments to produce a zero-knowledge argument protocol [33], and Applebaum et al. use secure multi-party protocols for ver-

ifying computations [4]. Also, Gentry’s fully homomorphic encryption [27] has engendered protocols for verifiable non-interactive computation [20, 24, 26]. However, despite striking improvements [28, 42, 47], the costs of hiding inputs (among other expenses) prevent any of the aforementioned verified computation schemes from getting close to practical (even by our relaxed standards).

PCPs, argument systems, and interactive proofs. Applying proof systems to verifiable computation is standard in the theory community [5–7, 10, 15, 32, 37, 38, 41], and the asymptotics continue to improve [13, 14, 22, 43]. However, none of this work has paid much attention to building systems.

Very recently, researchers have begun to explore using this theory for practical verified outsourced computation. In a recent preprint, Ben-Sasson et al. [12] investigate when PCP protocols might be beneficial for outsourcing. Since many of the protocols require representing computations as constraints, Ben-Sasson et al. [11] study improved reductions to constraints from a RAM model of computation. And Gennaro et al. [25] give a new characterization of NP to provide asymptotically efficient arguments without using PCPs.

However, as far as we know, only two research groups have made serious efforts toward practical systems. Our previous work [44, 45] built upon the efficient argument system of Ishai et al. [35]. In contrast, Cormode, Mitzenmacher, and Thaler [21] (hereafter, CMT) built upon the protocol of Goldwasser et al. [31], and a follow-up effort studies a GPU-based parallel implementation [49].

Comparison of GINGER and CMT [21, 49]. We compared three different implementations: *CMT-native*, *CMT-GMP*, and GINGER. *CMT-native* refers to the code and configuration released by Thaler et al. [49]; it works over a small field and thereby exploits highly efficient machine arithmetic but restricts the inputs to the computation unrealistically (see Section 4.1). *CMT-GMP* refers to an implementation based on *CMT-native* but modified by us to use the GMP library for multi-precision arithmetic; this allows more realistic computation sizes and inputs, as well as rational numbers.

m	domain	component	CMT-native	CMT-GMP	GINGER
256	\mathbb{Z}	verifier	40 ms	0.6 s	0.3 s
		prover	22 min	2.5 hr	36 min
		network	88 KB	0.3 MB	1.1 MB
128	\mathbb{Q}	verifier	–	260 ms	190 ms
		prover	–	1.0 hr	21 min
		network	–	1.8 MB	1.4 MB

Figure 11—CMT [21] compared to GINGER, in terms of *amortized* CPU and network costs (GINGER’s total costs are divided by a batch size of $\beta=5000$ instances), for $m \times m$ matrix multiplication. CMT-native uses native data types but is restricted to small problem sizes and domains. CMT-GMP uses the GMP library for multi-precision arithmetic (as does GINGER).

We perform two experiments using $m \times m$ matrix multiplication. Our testbed is the same as in Section 6. In the first one, we run with $m = 256$ and integer inputs. For CMT-GMP and GINGER, the inputs are 32-bit unsigned integers, and the prime (the field modulus) is 128 bits. For CMT-native, the prime is $2^{61} - 1$. In the second experiment, m is 128, the inputs are rational numbers (with $N_a = N_b = 32$; see Section 4.1), the prime is 320 bits, and we experiment only with CMT-GMP and GINGER.

We measure total CPU time and network cost; for CMT, we measure “network” traffic by counting bytes (the CMT verifier and prover run in the same process and hence the same machine). Each reported datum is an average over 3 sample runs; there is little experimental variation (less than 5% of the means).

Figure 11 depicts the results. CMT incurs a significant penalty when moving from native to GMP (and hence to realistic problem sizes). Comparing CMT-GMP and GINGER, the network and prover costs are similar (although network costs for CMT reflect high fixed overhead for their circuit). The *per-instance* verifier costs are also similar, but GINGER is batch verifying whereas CMT does not need to do so (a significant advantage).

A qualitative comparison is as follows. On the one hand, CMT does not require cryptography, has better asymptotic prover and network costs, and for some computations the verifier does not need batching to gain from outsourcing [49]. On the other hand, CMT applies to a smaller set of computations: if the computation is not efficiently parallelizable or does not naturally map to arithmetic circuits (e.g., it has order comparisons or conditionality), then CMT in its current form will be inapplicable or inefficient, respectively. Ultimately, GINGER and CMT should be complementary, as one can likely ease or eliminate some of the restrictions on CMT by incorporating the constraint formalism together with batching [48].

8 Summary and conclusion

This paper is a contribution to the emerging area of practical PCP-based systems for unconditional verifiable

computation. GINGER has combined theoretical refinements (slashing query costs and network overhead); a general computational model (including fractions and standard program constructs) with a compiler; and a massively parallel implementation that takes advantage of modern hardware. Together, these changes have brought us closer to a truly deployable system. Nevertheless, much work remains: the efficiency of the verifier depends on special hardware, the costs for the prover are still too high, and looping cannot yet be handled concisely.

Acknowledgments

Detailed attention from Edmund L. Wong substantially clarified this paper. Yuval Ishai, Mike Lee, Bryan Parno, Mark Silberstein, Chung-chieh (Ken) Shan, Sara L. Su, Justin Thaler, and the anonymous reviewers gave useful comments that improved this draft. The Texas Advanced Computing Center (TACC) at UT supplied computing resources. We thank Jane-ellen Long, of USENIX, for her good nature and inexhaustible patience. The research was supported by AFOSR grant FA9550-10-1-0073 and by NSF grants 1055057 and 1040083.

Our code and experimental configurations are available at <http://www.cs.utexas.edu/pepper>

References

- [1] CUDA (<http://developer.nvidia.com/what-cuda>).
- [2] Open MPI (<http://www.open-mpi.org>).
- [3] PAPI: Performance Application Programming Interface.
- [4] B. Applebaum, Y. Ishai, and E. Kushilevitz. From secrecy to soundness: efficient verification via secure computation. In *ICALP*, 2010.
- [5] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *J. of the ACM*, 45(3):501–555, May 1998.
- [6] S. Arora and S. Safra. Probabilistic checking of proofs: a new characterization of NP. *J. of the ACM*, 45(1):70–122, Jan. 1998.
- [7] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *STOC*, 1991.
- [8] M. Bellare, D. Coppersmith, J. Håstad, M. Kiwi, and M. Sudan. Linearity testing in characteristic two. *IEEE Transactions on Information Theory*, 42(6):1781–1795, Nov. 1996.
- [9] M. Bellare, S. Goldwasser, C. Lund, and A. Russell. Efficient probabilistically checkable proofs and applications to approximations. In *STOC*, 1993.
- [10] M. Ben-Or, S. Goldwasser, J. Kilian, and A. Wigderson. Multi-prover interactive proofs: how to remove intractability assumptions. In *STOC*, 1988.
- [11] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems. Feb. 2012. Cryptology eprint 071.
- [12] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. On the concrete-efficiency threshold of probabilistically-checkable proofs. *ECCC*, (045), Apr. 2012.
- [13] E. Ben-Sasson, O. Goldreich, P. Harsha, M. Sudan, and S. Vadhan. Robust PCPs of proximity, shorter PCPs and applications to coding. *SIAM J. on Comp.*, 36(4):889–974, Dec. 2006.

- [14] E. Ben-Sasson and M. Sudan. Short PCPs with polylog query complexity. *SIAM J. on Comp.*, 38(2):551–607, May 2008.
- [15] M. Blum and S. Kannan. Designing programs that check their work. *J. of the ACM*, 42(1):269–291, 1995.
- [16] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *J. of Comp. and Sys. Sciences*, 47(3):549–595, Dec. 1993.
- [17] D. Boneh and D. M. Freeman. Homomorphic signatures for polynomial functions. In *EUROCRYPT*, 2011.
- [18] D. Boneh, E. J. Goh, and K. Nissim. Evaluating 2-DNF formulas on ciphertexts. In *TCC*, 2005.
- [19] G. Brassard, D. Chaum, and C. Crépeau. Minimum disclosure proofs of knowledge. *J. of Comp. and Sys. Sciences*, 37(2):156–189, 1988.
- [20] K.-M. Chung, Y. Kalai, and S. Vadhan. Improved delegation of computation using fully homomorphic encryption. In *CRYPTO*, 2010.
- [21] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In *ITCS*, 2012.
- [22] I. Dinur. The PCP theorem by gap amplification. *J. of the ACM*, 54(3), June 2007.
- [23] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31:469–472, 1985.
- [24] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, 2010.
- [25] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. Apr. 2012. Cryptology eprint 215.
- [26] R. Gennaro and D. Wichs. Fully homomorphic message authenticators. May 2012. Cryptology eprint 290.
- [27] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.
- [28] C. Gentry, S. Halevi, and N. Smart. Homomorphic evaluation of the AES circuit. In *CRYPTO*, 2012.
- [29] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, Mar. 1991.
- [30] O. Goldreich. *Foundations of Cryptography: II Basic Applications*. Cambridge University Press, 2004.
- [31] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. In *STOC*, 2008.
- [32] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. on Comp.*, 18(1):186–208, 1989.
- [33] J. Groth. Linear algebra with sub-linear zero-knowledge arguments. In *CRYPTO*, 2009.
- [34] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*, 2011.
- [35] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short PCPs. In *Conference on Computational Complexity (CCC)*, 2007.
- [36] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL acceleration with commodity processors. In *NSDI*, 2011.
- [37] J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *STOC*, 1992.
- [38] J. Kilian. Improved efficient arguments (preliminary version). In *CRYPTO*, 1995.
- [39] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay—a secure two-party computation system. In *USENIX Security*, 2004.
- [40] R. C. Merkle. Digital signature based on a conventional encryption function. In *CRYPTO*, 1987.
- [41] S. Micali. Computationally sound proofs. *SIAM J. on Comp.*, 30(4):1253–1298, 2000.
- [42] M. Naehrig, K. Lauter, and V. Vaikuntanathan. Can homomorphic encryption be practical? In *ACM Workshop on Cloud Computing Security*, 2011.
- [43] A. Polishchuk and D. A. Spielman. Nearly-linear size holographic proofs. In *STOC*, 1994.
- [44] S. Setty, A. J. Blumberg, and M. Walfish. Toward practical and unconditional verification of remote computations. In *HotOS*, 2011.
- [45] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *NDSS*, 2012.
- [46] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality (extended version). Technical Report TR-12-14, Dept. of CS, UT Austin, June 2012.
- [47] N. Smart and F. Vercauteren. Fully homomorphic SIMD operations. Aug. 2011. Cryptology eprint 133.
- [48] J. Thaler. Personal communication, June 2012.
- [49] J. Thaler, M. Roberts, M. Mitzenmacher, and H. Pfister. Verifiable computation with massively parallel interactive proofs. In *USENIX HotCloud Workshop*, June 2012. Full paper at <http://arxiv.org/abs/1202.1350>, Feb. 2012.
- [50] C. Wang, K. Ren, J. Wang, and K. M. R. Urs. Harnessing the cloud for securely outsourcing large-scale systems of linear equations. In *Intl. Conf. on Dist. Computing Sys. (ICDCS)*, 2011.
- [51] D. A. Wheeler. SLOccount.
- [52] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, 1986.

A Efficient arguments with linear PCPs but no linearity tests

Whereas previous work [35, 45] established that the commitment protocol in phases 2 and 3 of PEPPER (§2.3) binds the prover to a particular function, there were no constraints on that function. The principal result of this section is that the prover is actually bound to a function that is linear, or very nearly so. As a consequence, we can eliminate linearity testing from the PCP protocol. Furthermore, the error bound from one run of this modified PCP protocol is far stronger (lower) than was known.

This section describes the base protocols (A.1), states the refinements and proves their soundness (A.2), and describes a few other optimizations (A.3).

A.1 Base protocols

GINGER uses a linear commitment protocol that is borrowed from PEPPER [45]; this protocol is depicted in Figure 12.⁵ As described in Section 2.3, PEPPER composes this protocol and a linear PCP; that PCP is depicted in Figure 13. The purpose of $\{\gamma_0, \gamma_1, \gamma_2\}$ in this figure is to make a maliciously constructed oracle unlikely to pass

⁵Like PEPPER, GINGER verifies in batches (§2.3), which changes the protocols a bit; see [45, Appendix C] for details.

Commit+Multidecommit

The protocol assumes an additive homomorphic encryption scheme $(\text{Gen}, \text{Enc}, \text{Dec})$ over a finite field, \mathbb{F} .

Commit phase

Input: Prover holds a vector $w \in \mathbb{F}^n$, which defines a linear function $\pi: \mathbb{F}^n \rightarrow \mathbb{F}$, where $\pi(q) = \langle w, q \rangle$.

1. Verifier does the following:
 - Generates public and secret keys $(pk, sk) \leftarrow \text{Gen}(1^k)$, where k is a security parameter.
 - Generates vector $r \in_R \mathbb{F}^n$ and encrypts r component-wise, so $\text{Enc}(pk, r) = (\text{Enc}(pk, r_1), \dots, \text{Enc}(pk, r_n))$.
 - Sends $\text{Enc}(pk, r)$ and pk to the prover.
2. Using the homomorphism in the encryption scheme, the prover computes $e \leftarrow \text{Enc}(pk, \pi(r))$ without learning r . The prover sends e to the verifier.
3. The verifier computes $s \leftarrow \text{Dec}(sk, e)$, retaining s and r .

Decommit phase

Input: the verifier holds $q_1, \dots, q_\mu \in \mathbb{F}^n$ and wants to obtain $\pi(q_1), \dots, \pi(q_\mu)$.

4. The verifier picks μ secrets $\alpha_1, \dots, \alpha_\mu \in_R \mathbb{F}$ and sends to the prover (q_1, \dots, q_μ, t) , where $t = r + \alpha_1 q_1 + \dots + \alpha_\mu q_\mu \in \mathbb{F}^n$.
5. The prover returns $(a_1, a_2, \dots, a_\mu, b)$, where $a_i, b \in \mathbb{F}$. If the prover behaved, then $a_i = \pi(q_i)$ for all $i \in [\mu]$, and $b = \pi(t)$.
6. The verifier checks: $b \stackrel{?}{=} s + \alpha_1 a_1 + \dots + \alpha_\mu a_\mu$. If so, it outputs (a_1, a_2, \dots, a_μ) . If not, it rejects, outputting \perp .

Figure 12—The commitment protocol of PEPPER [45], which generalizes a protocol of Ishai et al. [35]. q_1, \dots, q_μ are the PCP queries, and n is the size of the proof encoding. The protocol is written in terms of an additive homomorphic encryption scheme, but as stated elsewhere [35, 45], the protocol can be modified to work with a multiplicative homomorphic scheme, such as ElGamal [23].

the circuit test; to generate the $\{\gamma_i\}$, V multiplies each constraint by a random value and collects like terms, a process described in [5, 13, 35, 45]. The completeness and soundness of this PCP are explained in those sources, and our notation is borrowed from [45]. Here we just assert that the soundness error of this PCP is $\epsilon = (7/9)^\rho$; that is, if the proof π is incorrect, the verifier detects that fact with probability greater than $1 - \epsilon$. To make ϵ small, PEPPER takes $\rho = 70$.

A.2 Stronger soundness analysis and consequences

GINGER retains the (P, V) argument system of PEPPER [45] but uses a modified PCP protocol (depicted in Figure 14) that makes the following changes to the base PCP protocol (Figure 13):

- Remove the linearity queries and tests.
- Set $\rho = 1$.

Theorem A.1. The (P, V) described above is an argument system with soundness $\epsilon_G \approx \sqrt[6]{1/|\mathbb{F}|}$. (The exact value of ϵ_G depends on intermediate lemmas and will be given at the end of the section.)

We will prove this theorem at the end of this section. To build up to the proof, we first strengthen the definition of a linear commitment primitive. We note that only the third property (linearity) in the definition is new; the rest is taken from [45, Appendix B], which itself heavily borrows framing, notation, and text from Ishai et al. [35].

Definition A.1 (Commitment to a function with multiple decommitments (CFMD)). Define a two-phase experiment between two probabilistic polynomial time ac-

tors (S, R) (a sender and receiver, which correspond to our prover and verifier) in an environment \mathcal{E} that generates \mathbb{F} , w and $Q = (q_1, \dots, q_\mu)$. In the first phase, the *commit phase*, S has w , and S and R interact, based on their random inputs. In the *decommit phase*, \mathcal{E} gives Q to R , and S and R interact again, based on further random inputs. At the end of this second phase, R outputs $A = (a_1, \dots, a_\mu) \in \mathbb{F}^\mu$ or \perp . A CFMD meets the following properties:

- **Correctness.** At the end of the decommit phase, R outputs $\pi(q_i) = \langle w, q_i \rangle$ (for all i), if S is honest.
- **ϵ_B -Binding.** Consider the following experiment. The environment \mathcal{E} produces two (possibly distinct) μ -tuples of queries: $Q = (q_1, \dots, q_\mu)$ and $\hat{Q} = (\hat{q}_1, \dots, \hat{q}_\mu)$. R and a cheating S^* run the commit phase once and two independent instances of the decommit phase. In the two instances R presents the queries as Q and \hat{Q} , respectively. We say that S^* *wins binding* if R 's outputs at the end of the respective decommit phases are $A = (a_1, \dots, a_\mu)$ and $\hat{A} = (\hat{a}_1, \dots, \hat{a}_\mu)$, and for some i, j , we have $q_i = \hat{q}_j$ but $a_i \neq \hat{a}_j$. We say that the protocol meets the ϵ_B -Binding property if for all \mathcal{E} and for all efficient S^* , the probability of S^* winning binding is less than ϵ_B . The probability is taken over three sets of independent randomness: the commit phase and the two runnings of the decommit phase.
- **ϵ_L -Linearity.** Consider the same experiment above. We say that S^* *wins linearity* if R 's outputs at the end of the respective decommit phases are $A = (a_1, \dots, a_\mu)$ and $\hat{A} = (\hat{a}_1, \dots, \hat{a}_\mu)$, and for some i, j, k , we have $\hat{q}_k = q_i + q_j$ but $\hat{a}_k \neq a_i + a_j$. We say that

The linear PCP from [5]

Loop ρ times:

- Generate linearity queries: Select $q_1, q_2 \in_R \mathbb{F}^s$ and $q_4, q_5 \in_R \mathbb{F}^{s^2}$. Take $q_3 \leftarrow q_1 + q_2$ and $q_6 \leftarrow q_4 + q_5$.
- Generate quadratic correction queries: Select $q_7, q_8 \in_R \mathbb{F}^s$ and $q_{10} \in_R \mathbb{F}^{s^2}$. Take $q_9 \leftarrow (q_7 \otimes q_8 + q_{10})$.
- Generate circuit queries: Select $q_{12} \in_R \mathbb{F}^s$ and $q_{14} \in_R \mathbb{F}^{s^2}$. Take $q_{11} \leftarrow \gamma_1 + q_{12}$ and $q_{13} \leftarrow \gamma_2 + q_{14}$.
- Issue queries. Send q_1, \dots, q_{14} to oracle π , getting back $\pi(q_1), \dots, \pi(q_{14})$.
- Linearity tests: Check that $\pi(q_1) + \pi(q_2) = \pi(q_3)$ and that $\pi(q_4) + \pi(q_5) = \pi(q_6)$. If not, **reject**.
- Quadratic correction test: Check that $\pi(q_7) \cdot \pi(q_8) = \pi(q_9) - \pi(q_{10})$. If not, **reject**.
- Circuit test: Check that $(\pi(q_{11}) - \pi(q_{12})) + (\pi(q_{13}) - \pi(q_{14})) = -\gamma_0$. If not, **reject**.

If V makes it here, **accept**.

Figure 13—The linear PCP that PEPPER uses. It is from [5]. The notation $x \otimes y$ refers to the outer product of two vectors x and y (meaning the vector or matrix consisting of all pairs of components from the two vectors). The values $\{\gamma_0, \gamma_1, \gamma_2\}$ are described briefly in the text.

the protocol meets the ϵ_L -linearity property if for all \mathcal{E} and for all efficient S^* , the probability of S^* winning linearity is less than ϵ_L . As with the prior property, the probability is taken over three sets of independent randomness: the commit phase and the two runnings of the decommit phase.

Prior work proved that Commit+Multidecommit (Figure 12) meets the first two properties above [45]. We will now show that it also meets the third property.

Lemma A.1. Commit+Multidecommit meets the definition of ϵ_L -linearity, with $\epsilon_L = 1/|\mathbb{F}| + \epsilon_S$, where ϵ_S comes from the semantic security of the homomorphic encryption scheme.

Proof. We will show that if S^* can systematically cheat, then an adversary \mathcal{A} could use S^* to break the semantic security of the encryption scheme.

Let $r \in_R \mathbb{F}^n$ and $Z_1, Z_2 \in_R \mathbb{F}$ (we use \in_R to mean “drawn uniformly at random from”). Semantic security (see [30], definitions 5.2.2, 5.2.8 and Exercise 17) implies that for all PPT \mathcal{A} (\mathcal{A} can be non-uniform),

$$\Pr_{\text{Gen, Enc, } r, Z_1, Z_2} \{ \mathcal{A}(pk, \text{Enc}(pk, r), r + Z_1 q, r + Z_2 q) = Z_1 \} < 1/|\mathbb{F}| + \epsilon_S. \quad (1)$$

This holds for all $q \in \mathbb{F}^n$.⁶

⁶We are being loose here. Under the actual definition of semantic security, (a) ϵ_S should be replaced with a negligible function of n , and (b) the claim holds only for n sufficiently large.

GINGER’s PCP protocol

- Generate quadratic correction queries: Select $q_1, q_2 \in_R \mathbb{F}^s$ and $q_4 \in_R \mathbb{F}^{s^2}$. Define $q_3 \leftarrow (q_1 \otimes q_2 + q_4)$. Note that q_3 will not travel, as P can derive it.
- Generate circuit queries: Take $q_5 \leftarrow \gamma_1 + q_1$. Take $q_6 \leftarrow \gamma_2 + q_4$.
- Issue queries. Send $(q_1, q_2, q_4, q_5, q_6)$ to oracle π , getting back $\pi(q_1), \pi(q_2), \pi(q_3), \pi(q_4), \pi(q_5), \pi(q_6)$.
- Quadratic correction test: Check that $\pi(q_1) \cdot \pi(q_2) = \pi(q_3) - \pi(q_4)$. If not, **reject**.
- Circuit test: Check that $(\pi(q_5) - \pi(q_1)) + (\pi(q_6) - \pi(q_4)) = -\gamma_0$. If so, **accept**.

Figure 14—GINGER’s PCP protocol, which refines PEPPER’s protocol (Figure 13). This protocol eliminates linearity testing and repetition, and recycles queries [9].

Now, assume to the contrary that Commit+Multidecommit does not meet the definition of ϵ_L -linearity. Then there exists an environment \mathcal{E} producing $q_i, q_j, i, j, k, Q, \hat{Q}, S^*$ (where Q has q_i, q_j in the i th and j th positions and \hat{Q} has $q_i + q_j$ in the k th position) such that $\Pr_{\text{all 3 phases}} \{ S^* \text{ wins linearity under } \mathcal{E} \} > 1/|\mathbb{F}| + \epsilon_S$. Let $q' \triangleq \hat{q}_k = q_i + q_j$.

We now describe an algorithm \mathcal{A} that, when given input $I = (pk, \text{Enc}(pk, r), r + Z_1 q', r + Z_2 q')$, can recover Z_1 with probability more than $1/|\mathbb{F}| + \epsilon_S$. \mathcal{A} has $Q, \hat{Q}, q_i, q_j, i, j, k$ hard-wired (because it is working under environment \mathcal{E}) and works as follows:

- (a) \mathcal{A} gives $(pk, \text{Enc}(pk, r))$ to S^* and ignores the reply.
- (b) \mathcal{A} randomly generates $\alpha_1, \dots, \alpha_\mu$ and sends to S^* the input $(Q, r + \alpha_1 q_1 + \dots + (\alpha_i + Z_1) q_i + \dots + (\alpha_j + Z_1) q_j + \dots + \alpha_\mu q_\mu)$. \mathcal{A} is able to construct this input because \mathcal{A} was given $r + Z_1 q' = r + Z_1 q_i + Z_1 q_j$. In response, S^* returns $(b, a_1, \dots, a_i, \dots, a_j, \dots, a_\mu)$.
- (c) \mathcal{A} randomly generates $\hat{\alpha}_1, \dots, \hat{\alpha}_\mu$. \mathcal{A} sends to S^* the input $(\hat{Q}, r + \hat{\alpha}_1 \hat{q}_1 + \dots + Z_2 \hat{q}_k + \dots + \hat{\alpha}_\mu \hat{q}_\mu)$. \mathcal{A} is able to construct this input because \mathcal{A} was given $r + Z_2 q' = r + Z_2 \hat{q}_k$. \mathcal{A} gets back $(\hat{b}, \hat{a}_1, \dots, \hat{a}_k, \dots, \hat{a}_\mu)$.

At this point, \mathcal{A} assumes that the responses from S^* pass the decommitment phase; that is, \mathcal{A} acts as if $b = s + \alpha_1 a_1 + \dots + (\alpha_i + Z_1) a_i + \dots + (\alpha_j + Z_1) a_j + \dots + \alpha_\mu a_\mu$ and $\hat{b} = s + \hat{\alpha}_1 \hat{a}_1 + \dots + Z_2 \hat{a}_k + \dots + \hat{\alpha}_\mu \hat{a}_\mu$. \mathcal{A} can write

$$K_1 = Z_2 \hat{a}_k - Z_1 (a_i + a_j), \quad (2)$$

where \mathcal{A} can derive $K_1 = \hat{b} - b - \sum_{\iota \neq k} \hat{\alpha}_\iota \hat{a}_\iota + \sum_\iota \alpha_\iota a_\iota$. Now, let $t = r + Z_1 q'$ and let $\hat{t} = r + Z_2 q'$ (both of these were supplied as input to \mathcal{A}). These two equations concern vectors. However, by choosing an index ι in the vector q' where q' is not zero (if the vector is zero everywhere, then r is revealed), \mathcal{A} can derive

$$K_2 = Z_2 - Z_1, \quad (3)$$

where $K_2 = (\hat{t}^{(\iota)} - t^{(\iota)})/q'^{(\iota)}$.

Now, observe that if $\hat{a}_k \neq a_i + a_j$ (as happens when S^* wins), then \mathcal{A} can recover Z_1 by solving equations (2) and (3). Thus,

$$\begin{aligned} & \Pr_{\text{Gen, Enc, } r, Z_1, Z_2, \vec{\alpha}, \vec{\alpha}'} \{ \mathcal{A}(I) = Z_1 \} \\ & \geq \Pr_{\text{Gen, Enc, } r, Z_1, Z_2, \vec{\alpha}, \vec{\alpha}'} \{ S^* \text{ wins linearity under } \mathcal{E} \} \\ & = \Pr_{\text{all 3 phases}} \{ S^* \text{ wins linearity under } \mathcal{E} \} \\ & > 1/|\mathbb{F}| + \epsilon_S. \end{aligned} \quad (4)$$

The equality holds because the distribution of $(\alpha_1, \dots, \alpha_i + Z_1, \dots, \alpha_j + Z_1, \dots, \alpha_\mu)$ and $(\hat{\alpha}_1, \dots, Z_2, \dots, \hat{\alpha}_\mu)$ is equivalent to the distribution from which R selects in the decommit phases of the three-phase experiment, under Commit+Multidecommit. Meanwhile, inequality (4) contradicts inequality (1). \square

The lemmas ahead show that, under Commit+Multidecommit, S is bound to a *nearly linear* function, $\tilde{f}(\cdot)$; specifically, $\tilde{f}(\cdot)$ is δ^* -close to linear for small δ^* . By contrast, previous work [35, 45] showed only that S was bound to *some* function $\tilde{f}(\cdot)$.

We now give some notation and restate two claims from [45]. Let ζ be the event that R 's output is a vector (a_1, \dots, a_μ) ; equivalently, ζ is the event that R 's output is non- \perp . Below, we sometimes write $\Pr_{\text{commit}}\{\cdot\}$ or $\Pr_{\text{decommit}}\{\cdot\}$ to mean the probability over the random choices of the commit or decommit phases.

Lemma A.2 (Existence of an extractor function [45]).

Let (S, R) be a CFMD protocol with binding error ϵ_B . Let $\epsilon_C = \mu \cdot 2 \cdot (2\sqrt[3]{9/2} + 1) \cdot \sqrt[3]{\epsilon_B}$. Let $v = (v_{S^*}, v_R)$ represent the views of S^* and R after the commit phase (v captures the randomness of the commit phase). For every efficient S^* and for every v , there exists a function $\tilde{f}_v : \mathbb{F}^\mu \rightarrow \mathbb{F}$ such that the following holds.⁷ For any environment \mathcal{E} , the output of R at the end of the decommit phase is, except with probability ϵ_C , either \perp or satisfies $a_i = \tilde{f}_v(q_i)$ for all $i \in [\mu]$, where (q_1, \dots, q_μ) are the decommitment queries generated by \mathcal{E} , and the probability is over the random inputs of S^* and R in both phases.

Lemma A.3. Let $\epsilon_3 = (2\sqrt[3]{9/2} + 1) \cdot \sqrt[3]{\epsilon_B}$. Label the i th query in Q as q_i and the i th response as a_i . For all Q, i , we have $\Pr_{\text{commit, decommit}} \{ \zeta \cap \{ a_i \neq \tilde{f}_v(q_i) \} \} < 2\epsilon_3$.

Proof. Follows from a claim in [45] (Claim B.4). \square

Lemma A.4. For all $q_1, q_2 \in \mathbb{F}^\mu$, $\Pr_{\text{commit}} \{ \tilde{f}_v(q_1) + \tilde{f}_v(q_2) \neq \tilde{f}_v(q_1 + q_2) \} < \epsilon_F \triangleq \epsilon_L + 6\epsilon_3$.

⁷Note that after the commit phase, $\tilde{f}_v(\cdot)$ is deterministic. $(\tilde{f}_v(\cdot))$ is defined [35, 45] to map q to the value that R is most likely to successfully output in the decommit phase.)

Proof. Assume otherwise. Then for some q_1 and q_2 , we have $\Pr_{\text{commit}} \{ \tilde{f}_v(q_1) + \tilde{f}_v(q_2) \neq \tilde{f}_v(q_1 + q_2) \} \geq \epsilon_F$, which implies $\Pr_{\text{all 3 phases}} \{ \tilde{f}_v(q_1) + \tilde{f}_v(q_2) \neq \tilde{f}_v(q_1 + q_2) \} \geq \epsilon_F$, since we can “add coin flips that don’t matter”, namely those of the two decommit phases.

Now, consider the game in the definition of ϵ_L -linearity, and set $Q = (q_1, q_2, \dots)$ and $\tilde{Q} = (q_1 + q_2, \dots)$. Let η be the event that S^* wins in this game. Let ν be the event that the outputs a_1, a_2, \hat{a}_1 are given by the function $\tilde{f}_v(\cdot)$. Then $\Pr_{\text{all 3 phases}} \{ \neg \nu \} < 6\epsilon_3$, by Lemma A.3, by the union bound, and by (again) “adding coin flips that don’t matter” to get from a probability over two phases to one over three phases. Now, note that $\Pr_{\text{all 3 phases}} \{ \eta | \nu \} \geq \epsilon_F$, by the contrary hypothesis. This implies that $\Pr_{\text{all 3 phases}} \{ \eta \} \geq \epsilon_F - 6\epsilon_3 = \epsilon_L$, which contradicts the definition of ϵ_L -linearity. \square

Lemma A.4 almost talks about a linearity test [16]! But linearity testing theory [8] relates (a) the probability *over randomly chosen queries* that the test fails and (b) the closeness-to-linearity of the tested function. Thus, to apply the theory, we line up Lemma A.4 and (a).

Lemma A.5. With probability greater than $1 - \sqrt{\epsilon_F}$ over the commit phase, the fraction of (q_1, q_2) pairs that cause $\tilde{f}_v(\cdot)$ to fail the linearity test is $\leq \sqrt{\epsilon_F}$.

Proof. Let I_{v, q_1, q_2} be an indicator random variable that equals 1 if, in view v (that is, given the randomness of the commit phase), $\tilde{f}_v(q_1 + q_2) \neq \tilde{f}_v(q_1) + \tilde{f}_v(q_2)$. The lemma is equivalent to the statement

$$\Pr_{\text{commit}} \{ \Pr_{q_1, q_2} \{ I_{v, q_1, q_2} = 1 \} > \sqrt{\epsilon_F} \} < \sqrt{\epsilon_F}.$$

Now, define a random variable $Y_v = \frac{1}{Q^2} \sum_{q_1, q_2} I_{v, q_1, q_2}$, where $Q = |\mathbb{F}^\mu|$ is the number of possibilities for each of q_1 and q_2 . By linearity of expectation, $E_{\text{commit}}[Y_v] = \frac{1}{Q^2} \cdot (E[I_{v, i}] + \dots + E[I_{v, Q^2}])$, where $E[I_{v, i}]$ is the probability, over the commit phase, that a particular (q_j, q_k) pair causes $\tilde{f}_v(\cdot)$ to fail the linearity test. Lemma A.4 implies that $E[I_{v, i}] < \epsilon_F$ for all i ; hence, $E_{\text{commit}}[Y_v] < \epsilon_F$. We now apply a Markov bound to Y_v :

$$\Pr_{\text{commit}} \{ Y_v > \sqrt{\epsilon_F} \} < \frac{E_{\text{commit}}[Y_v]}{\sqrt{\epsilon_F}} < \frac{\epsilon_F}{\sqrt{\epsilon_F}} = \sqrt{\epsilon_F}.$$

But Y_v is equivalent to $\Pr_{q_1, q_2} \{ I_{v, q_1, q_2} = 1 \}$; making this substitution immediately above yields the lemma. \square

Lemma A.6. Let δ^* be the lesser root of $6\delta^2 - 3\delta + \sqrt{\epsilon_F} = 0$. If $\sqrt{\epsilon_F} < \frac{2}{9}$, then with probability greater than $1 - \sqrt{\epsilon_F}$ over the commit phase, $\tilde{f}_v(\cdot)$ is δ^* -close to linear.

Proof. We use the linearity testing results of Bellare et al. [8, 9] and the terminology of [8]. Define $\text{Dist}(f, g)$ to be the fraction of inputs on which f and g disagree.

Define $\text{Dist}(f)$ to be the fraction of inputs on which f disagrees with its “closest linear function” [8]. Define $\text{Rej}(f)$ to be the probability, over uniformly random choices of x and y from the domain of f , that $f(x)+f(y) \neq f(x+y)$; $\text{Rej}(f)$ is the probability that f fails the linearity test. As stated by Bellare et al. [8]:

- If $\text{Dist}(f) = \delta$, then $\text{Rej}(f) \geq 3\delta - 6\delta^2$.
- If $\text{Dist}(f) \geq \frac{1}{4}$, then $\text{Rej}(f) \geq \frac{2}{9}$.

The above implies the following claim: for all $\delta' \in \{\delta' \mid 3\delta' - 6\delta'^2 < \frac{2}{9} \text{ and } 0 \leq \delta' \leq \frac{1}{4}\}$, if $\text{Rej}(f) \leq 3\delta' - 6\delta'^2$, then $\text{Dist}(f) \leq \delta'$. (To see this, fix δ' . Assume to the contrary that $\delta = \text{Dist}(f) > \delta'$. There are two cases, and both contradict the given. If $\delta < \frac{1}{4}$, then $\text{Rej}(f) \geq 3\delta - 6\delta^2 > 3\delta' - 6\delta'^2$. If $\delta \geq \frac{1}{4}$, then $\text{Rej}(f) \geq \frac{2}{9} > 3\delta' - 6\delta'^2$.)

From lemma A.5, the probability is greater than $1 - \sqrt{\epsilon_F}$ over the commit phase that $\text{Rej}(\tilde{f}_v) \leq \sqrt{\epsilon_F}$. We call such commit phases *usual*. Under a *usual* commit phase, we can apply the claim just above. To do so, we assume that $\sqrt{\epsilon_F} < \frac{2}{9}$, and we set δ^* so that $\sqrt{\epsilon_F} = 3\delta^* - 6\delta^{*2}$ and $\delta^* \leq \frac{1}{4}$ (such a δ^* is guaranteed to exist because the parabola is symmetric about $\delta = \frac{1}{4}$). The claim implies that $\text{Dist}(\tilde{f}_v) \leq \delta^*$, or that \tilde{f}_v is δ^* -close to linear. \square

Lemma A.7. If the PCP oracle π is known to be δ^* -close to linear, then the linear PCP (Section A.1) with linearity testing removed has soundness error $\kappa > \max\{4\delta^* + \frac{2}{|\mathbb{F}|}, 4\delta^* + \frac{1}{|\mathbb{F}|\}\}$.

Proof. This follows from the proof flow that establishes the soundness of linear PCPs, as in [5]. (A self-contained example is in Appendix D of [45].) Those proofs first establish that if the linearity test passes with probability higher than the soundness error, then π is δ -close to linear, for some δ . However, if we are *given* that π is δ^* -close to linear, then we can start those proofs midway and obtain the soundness of π as κ . \square

Proof of Theorem A.1. Lemma A.2 implies that there exists an extractor function that determines a (possibly incorrect) oracle $\tilde{\pi}$ such that, if V' does not reject during decommit, then with all but probability ϵ_C , V' receives back $\tilde{\pi}(q_1), \dots, \tilde{\pi}(q_\mu)$. We can thus “pay” probability ϵ_C in the union bound (below) to assume that V' hears back from $\tilde{\pi}$ itself. This allows us to apply Lemma A.6, at which point we can “pay” $\sqrt{\epsilon_F}$ more probability (again in the union bound below) to get that $\tilde{\pi}$ is δ^* -close to linear. (Applying the lemma requires that $\sqrt{\epsilon_F} < \frac{2}{9}$, and we will verify below that this bound holds.) Now, we can apply Lemma A.7 to ρ runs of the PCP protocol, giving a PCP soundness error of κ^ρ . Thus, the probability that V' wrongly accepts a proof is bounded from above by:

$$\epsilon_G = \epsilon_C + \sqrt{\epsilon_F} + \kappa^\rho.$$

By inspection (of the lemmas), the dominant contributor to ϵ_G , namely $\sqrt{\epsilon_F}$, is proportional to $\sqrt[6]{1/|\mathbb{F}|}$. \square

We compute a bound on ϵ_G as follows.

- ϵ_C is given in Lemma A.2. We take $\mu = 6$ (per Figure 14). We also take $\epsilon_B = 1/|\mathbb{F}|$ (following [45]; this amounts to ignoring the error from the semantic security of the homomorphic encryption scheme) and $|\mathbb{F}| = 2^{128}$, giving $\epsilon_C < 7.4 \cdot 10^{-12}$.
- $\epsilon_F = \epsilon_L + 6\epsilon_3$ (from Lemma A.4). ϵ_3 is given in Lemma A.3. We set $\epsilon_L = 1/|\mathbb{F}|$ (which again amounts to ignoring ϵ_S). Again taking $|\mathbb{F}| = 2^{128}$, we get $\sqrt{\epsilon_F} < 1.9 \cdot 10^{-6}$. Thus, $\sqrt{\epsilon_F} < 2/9$, as required.
- $\kappa = 4\delta^* + \frac{2}{|\mathbb{F}|}$, where δ^* is the lesser root of $6\delta^2 - 3\delta + \sqrt{\epsilon_F}$. This gives $\delta^* = 6.4 \cdot 10^{-7}$ and $\kappa = 2.6 \cdot 10^{-6}$.

Since κ and $\sqrt{\epsilon_F}$ are roughly the same, there is not much point to taking $\rho > 1$. Thus, we take $\rho = 1$, giving $\epsilon_G < 4.5 \cdot 10^{-6}$ when $|\mathbb{F}| = 2^{128}$. When $|\mathbb{F}| = 2^{192}$, we get $\epsilon_G < 2.8 \cdot 10^{-9}$.

A.3 Optimizing out queries

GINGER’s PCP protocol includes two further refinements. First, the protocol reuses q_4 and q_1 from test to test. This reuse is sound because the PCP soundness lemma [5] is of the form, “if all tests pass with probability greater than X , then the proof oracle π has a certain desired property”; meanwhile, as Bellare et al. [9] observe, the tests need not be independent! One can observe the savings by comparing Figure 13 (minus the linearity queries) to Figure 14. The protocol goes from 8 queries (the original 14 minus 6 linearity queries) to 6 queries, though the real savings for the prover is in reducing the 4 high-order queries (that is, queries to the \mathbb{F}^{s^2} component of π) to 3. Moreover, the verifier saves because it goes from generating pseudorandomness for 3 high-order queries (including γ_2) to 2. Second, V avoids transmitting a query (q_3) that P can generate for itself. This optimization offsets the consistency query, which is computed over \mathbb{Z} not \mathbb{Z}/p (owing to the details of our use of ElGamal [45, Appendix E]) and thus has roughly twice as many bits as a PCP query.

Optimally Robust Private Information Retrieval*

Casey Devet Ian Goldberg
University of Waterloo
{cjdevet,iang}@cs.uwaterloo.ca

Nadia Heninger
University of California, San Diego
nadiah@cs.ucsd.edu

Abstract

We give a protocol for multi-server information-theoretic private information retrieval which achieves the theoretical limit for Byzantine robustness. That is, the protocol can allow a client to successfully complete queries and identify server misbehavior in the presence of the maximum possible number of malicious servers. We have implemented our scheme and it is extremely fast in practice: up to thousands of times faster than previous work. We achieve these improvements by using decoding algorithms for error-correcting codes that take advantage of the practical scenario where the client is interested in multiple blocks of the database.

1 Introduction and related work

Private information retrieval (PIR) is a way for a client to look up information in an online database without letting the database servers learn the query terms or responses. A simple if inefficient way to do this is for the database server to send a copy of the entire database to the client, and let the client look up the information for herself. This is called *trivial download*. The goal of PIR is to transmit less data while still protecting the privacy of the query. PIR is a fundamental building block for many proposed privacy-sensitive applications in the literature, including patent databases [2], domain name registration [28], anonymous email [33], and improving the scalability of anonymous communication networks [26].

The simplest kind of query one can make with PIR is to consider the database to be composed of a number of blocks of equal size, and to retrieve a particular block from the database by its absolute position [10]. Although this simple type of query does not appear to be very useful in practice, it turns out that it can be used as a black-box building block to construct more complex and use-

ful queries, such as searching for keywords [9] or private SQL queries [28].

PIR protocols can be grouped into two classes corresponding to the security guarantees they provide. One class is *computational* PIR [8], in which the database servers can learn the client's query if they can apply sufficient computational power to break a particular cryptographic system. The other class of protocols — those we will consider in this work — is *information-theoretic* PIR [10, 11], in which no amount of computation will allow the reconstruction of the client's query. In these protocols, the query is protected by splitting it among multiple database servers. (Chor et al. [10] show that information-theoretic PIR with less data transfer than the trivial download scheme is impossible with only one server.) As is common in many distributed privacy-enhancing technologies, such as mix networks [7], Tor [14], or some forms of electronic voting [6], we must assume that some fraction of the servers above some threshold are not colluding against the client.

While much of the theoretical work on PIR focuses strictly on minimizing the amount of data transferred [15, 38], in a practical setting we must take other aspects, particularly the computational performance, into account. In 2007, Sion and Carbunar [36] opined that, given trends in computational power and network speeds, it would always be faster to send the whole database to the client than to use PIR to process it. However, they only considered one kind of computational PIR [23] in their analysis.

In fact, recent work by Olumofin and Goldberg [29] demonstrates that a more recent computational PIR scheme by Aguilar Melchor and Gaborit [1] is an order of magnitude faster than trivial download, while information-theoretic (IT) PIR can be two to three orders of magnitude faster. These PIR protocols are well matched to deployment on mobile clients as they require low data transfer, low client-side computation, and moderate server-side computation [30]. For example, to retrieve one 32 KiB block from a 1 GiB database, an IT-

*An extended version of this paper is available. [13]

PIR client would send one block of data to, and receive one block of data from, each server. The servers each perform about 1.4 CPU seconds of computation, and the client performs about 140 ms of computation.

1.1 Byzantine robustness

An important practical consideration with multi-server PIR is how to deal with servers that *do not respond* to a client’s queries, or that *respond incorrectly*, either through malice or error. These are respectively termed the *robustness* and *Byzantine robustness* problems.

The main result of this paper is to improve the Byzantine robustness of information-theoretic PIR. In order to guarantee information-theoretic PIR, one must have multiple servers in the protocol; Byzantine robustness guarantees that the protocol still functions correctly even if some of the servers fail to respond or give incorrect or malicious responses. Byzantine robustness makes no assumptions on the type of errors that can appear—the model covers spurious or random errors as well as malicious interference—and the bounds are given in terms of the number of servers which ever give incorrect responses. The client must still be able to determine the answer to her query, even when some number of the servers fail to respond, or give incorrect answers; further, in the latter case, the client would like to learn *which* servers misbehaved so that they can be avoided in the future. (In the single-server case, the owner of the database can provide a cryptographic signature on each block in order to ensure integrity, as PIR-Tor [26] does. Without computational assumptions or some kind of shared secret, it does not make much sense to consider robustness or Byzantine robustness in a single-server PIR setting.)

Beimel and Stahl [3, 4] were the first to consider robustness and Byzantine robustness for PIR. Consider an ℓ -server information-theoretic PIR setting, where only k of the servers respond, v of the servers respond incorrectly, and the system can withstand up to t colluding servers without revealing the client’s query (t is called the *privacy level*). (This is termed “ t -private v -Byzantine robust k -out-of- ℓ PIR”.) Then the protocol of Beimel and Stahl works when $v \leq t < k/3$. Under those conditions, the protocol will always output to the client a unique block, which will be the correct one; this is called *unique decoding*.

In 2007, Goldberg [19] observed that by allowing for the possibility of *list decoding* — that is, that the protocol may sometimes output a small number of blocks instead of just one — the privacy level and the number of misbehaving servers can be substantially increased, up to $t < k$ and $v < k - \lfloor \sqrt{kt} \rfloor$. He also showed that in many scenarios, the probability of more than one block being output by the protocol is vanishingly small, while in others, one

can employ standard techniques to convert list decoding to unique decoding [25] at the cost of slightly increasing the size of the database. The communication overhead of Goldberg’s protocol is $k + \ell$; that is, to retrieve one block of data (say b bits), the protocol transfers a total of $(k + \ell)b$ bits, for the optimal choice of block size b .

1.2 Our contributions

- We change only the client side of Goldberg’s 2007 protocol to improve its Byzantine robustness from $v < k - \lfloor \sqrt{kt} \rfloor$ to $v < k - t - 1$, which is the theoretically maximum possible value. Depending on the deployment scenario, the communication overhead of our protocol ranges from a factor of $k + \ell$ to a maximum of $v(k + \ell)$.
- Our protocol is considerably faster than Goldberg’s protocol for many reasonable parameter choices. We implemented our protocol on top of Goldberg’s open-source Percy++ [18] distribution and find that our new protocol can be up to 3–4 orders of magnitude (thousands of times) faster than the original in reconstructing the correct response to a query in the presence of Byzantine servers.

The robustness and efficiency improvements to the PIR protocol given in this paper mean that recovering from Byzantine errors even in an extremely adversarial or noisy setting is not just academically feasible, but is completely reasonable for user-facing applications.

Goldberg’s protocol uses Shamir secret sharing to hide the query; since Shamir secret sharing is based off of polynomial interpolation, the problem of recovering the response in the case of Byzantine failures corresponds to noisy polynomial reconstruction, which is exactly the problem of decoding Reed-Solomon codes. The theoretical contribution of this work is to observe that the practical setting of clients performing multiple queries allows us to use sophisticated decoding algorithms that can decode multiple queries *simultaneously* and achieve an enormous improvement in both performance and the level of robustness.

1.3 Organization

The remainder of the paper is organized as follows. In Section 2 we will introduce the tools that we need to present our protocol: Shamir secret sharing, Reed-Solomon codes, and decoding algorithms for multipolynomial extensions of these codes. In Section 3 we review the PIR protocols that form the foundation for our work. We present our protocol and algorithms in Section 4, and give experimental results in Section 5. We conclude the paper in Section 6.

2 Preliminaries

2.1 Notation

We will use the following variables throughout the paper:

- ℓ denotes the total number of servers
- t is the privacy level: no coalition of t or fewer servers can learn the client's query
- k is the number of servers that respond
- v is the number of Byzantine servers that respond and h is the number of honest servers that respond (so $h + v = k$). Byzantine servers may respond with any maliciously chosen value.
- D is the database
- r is the number of blocks in the database
- s is the number of words in each database block
- w is the number of bits per word

We denote by \mathbf{e}_j the standard basis vector $\langle 0, \dots, 0, 1, 0, \dots, 0 \rangle$ where the 1 is in the j^{th} place. $x \in_R X$ means selecting the element x uniformly at random from the space X .

2.2 Shamir secret sharing

The classic Shamir secret sharing scheme [34] allows a *dealer* to choose a secret value σ , and distribute *shares* of that secret to ℓ *players*. If t or fewer of the players come together, they learn no information about σ , but if more than t pool their shares, they can easily recover the secret. (t and ℓ are parameters of the scheme, with $t < \ell$.)

The scheme works as follows: let σ be an arbitrary element of some finite field \mathbb{F} (not necessary uniformly distributed). The dealer selects ℓ arbitrary distinct non-zero indices $\alpha_1, \dots, \alpha_\ell \in \mathbb{F}$, and selects t elements $a_1, \dots, a_t \in_R \mathbb{F}$ uniformly at random. The dealer constructs the polynomial $f(x) = \sigma + a_1x + a_2x^2 + \dots + a_tx^t$, and gives to player i the share $(\alpha_i, f(\alpha_i)) \in \mathbb{F} \times \mathbb{F}$ for $1 \leq i \leq \ell$. Note that the secret σ is just $f(0)$. Now any $t + 1$ or more players can use Lagrange interpolation to reconstruct the polynomial f , and evaluate $f(0)$ to yield σ . However, t or fewer players learn absolutely no information about σ .

Complications arise during reconstruction, however, when some of the shares being brought together to reconstruct f are incorrect. Dealing with this case involves working with error-correcting codes, and will be discussed in Section 2.3, next.

Sharing a vector of elements in \mathbb{F}^r rather than a single field element is done in the straightforward way: each coordinate of the vector is secret shared separately, using r independent random polynomials.

2.3 Error-correcting codes

We will use error-correcting codes to handle Byzantine robustness. In the case of servers that merely fail to respond, we could try to use an erasure code — an error-correcting code which can be decoded when some symbols are erased by the channel. In order to handle Byzantine failures, we will use error-correcting codes that can handle both corrupted and missing symbols. Our scheme will transform malicious errors into random errors, which will allow us to achieve much higher robustness (with high probability) than was efficiently possible before. In addition, the use of these error-correcting codes allows us to identify servers that cheat during the protocol, and not use them in the future.

The error-correcting codes that we will use in our protocol are based off of Reed-Solomon codes. [32] This is a natural choice to use with Shamir secret sharing, as they both use polynomial interpolation. If a message of length $t + 1$ consists of elements $\{a_0, a_1, \dots, a_t\}$ in some field \mathbb{F} then we can define the degree- t polynomial $f(x) = a_0 + a_1x + \dots + a_tx^t$. Fix k distinct field elements $\alpha_1, \dots, \alpha_k$. A Reed-Solomon codeword consists of the evaluations of f at each point: $\{f(\alpha_1), \dots, f(\alpha_k)\}$.

The Berlekamp-Welch [5] algorithm can efficiently decode a Reed-Solomon codeword with up to $v < (k - t)/2$ errors, which is the theoretical maximum for unique decoding. However, if one is willing to accept the possibility of decoding to multiple valid codewords, the Guruswami-Sudan algorithm [22] improves the *decoding radius* to $v < k - \sqrt{kt}$. This is known as *list decoding*: the algorithm returns a list of all valid codewords.

2.3.1 Multi-polynomial reconstruction

The above decoding algorithms all consider the case of noisy interpolation of a single polynomial. More recently, Parvaresh and Vardy [31], and Guruswami and Rudra [21] designed codes that could be efficiently list decoded, approaching the asymptotic limit of $v < k - t - 1$. These codes are based around the idea of extending the Reed-Solomon code to evaluate *multiple polynomials* simultaneously, and using clever constructions of the polynomials in order to efficiently decode the codewords. One of the main contributions of this paper is to adapt these ideas to a cryptographic setting. We cannot directly use their constructions, as their polynomials have a special structure that would make them unsuitable for secret sharing. However, using a randomized construction we can nonetheless efficiently decode such multi-polynomial codes in practice with high probability, yielding a secret sharing system robust to many errors.

The codes that we will use will reconstruct several polynomials simultaneously from noisy evaluation

points. Define m polynomials

$$\begin{aligned} f_1(x) &= a_{10} + a_{11}x + \dots + a_{1t}x^t, \\ &\vdots \\ f_m(x) &= a_{m0} + a_{m1}x + \dots + a_{mt}x^t. \end{aligned}$$

Then a codeword will consist of the evaluations of each of these polynomials at points $\alpha_1, \dots, \alpha_k$:

$$\begin{aligned} f_1(\alpha_1), \dots, f_m(\alpha_1), \\ \vdots \\ f_1(\alpha_k), \dots, f_m(\alpha_k) \end{aligned}$$

This general case is considered by Cohn and Heninger [12], who give an algorithm that heuristically reconstructs every polynomial as long as there are no more than $v < k - t^{m/(m+1)}k^{1/(m+1)}$ values of i for which the received value of some $f_p(\alpha_i)$ is incorrect. In our application to PIR, each polynomial will correspond to a column of the database matrix D , and each value α_i will correspond to a PIR server; therefore, we will be able to tolerate v dishonest servers.

2.3.2 Linear multi-polynomial decoding

The list-decoding algorithms of Guruswami-Sudan, Parvaresh-Vardy, and Cohn-Heninger all work by constructing a polynomial which vanishes to high multiplicity at the codeword. If one simply uses multiplicity one, one can obtain a “linear” variant of the Cohn-Heninger algorithm [12] which is extremely fast in practice. It reconstructs each polynomial uniquely when no more than $v \leq \frac{m}{m+1}(k-t-1)$ values of i have incorrect received values of some $f_p(\alpha_i)$. This algorithm works with high probability in practice as long as the errors are randomized. We will show later how to set up our protocol to enforce that even malicious servers can only insert random errors.

Since this linear variant is not explicitly described in their work, we provide a brief outline in Algorithm 1.

Polynomial lattice basis reduction. Step 4 in the algorithm uses a “polynomial lattice basis row reduction algorithm”. This is an algorithm which takes as input a matrix M of *polynomials* and applies elementary row operations over the ring of polynomials to produce a matrix M' whose coefficient polynomials have minimal degree. [37] There are several polynomial-time polynomial lattice basis reduction algorithms. (This is a refreshing contrast to the case of *integer* lattices where finding exact shortest vectors is NP-hard and efficient algorithms

Algorithm 1 Fast multi-polynomial reconstruction

Input: km points (α_i, y_{ip}) $1 \leq i \leq k$, $1 \leq p \leq m$, degree bound t , and minimum number of correct points $h = k - v$.

Output: m polynomials f_1, \dots, f_m of degree at most t such that for at least h values of i , $f_p(\alpha_i) = y_{ip}$ for all $1 \leq p \leq m$

- 1: Use Lagrange interpolation to construct m polynomials f_p^* of degree at most $k-1$ s.t. $f_p^*(\alpha_i) = y_{ip}$ for each $1 \leq i \leq k$.
- 2: Construct the degree- k polynomial

$$N(x) = \prod_{i=1}^k (x - \alpha_i)$$

- 3: Construct the $(m+1) \times (m+1)$ polynomial matrix

$$M = \begin{bmatrix} x^t & & & -f_1^*(x) \\ & x^t & & -f_2^*(x) \\ & & \ddots & \\ & & & x^t & -f_m^*(x) \\ & & & & N(x) \end{bmatrix}$$

- 4: Run a polynomial lattice basis row reduction algorithm on M .
- 5: Discard the largest-degree row in the reduced matrix. If any remaining row has degree larger than h , abort.
- 6: Write the remaining $m \times (m+1)$ matrix as $[\mathbf{A}|\mathbf{b}]$, where \mathbf{A} is an $m \times m$ matrix, and \mathbf{b} is an $m \times 1$ column vector.
- 7: Solve the linear system of equations

$$\left(\frac{1}{x^t} \mathbf{A} \right) \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{bmatrix} = \mathbf{b}$$

- 8: **return** (f_1, \dots, f_m)
-

such as LLL [24] can only obtain an exponential approximation.) The algorithm of Giorgi et al. [17] runs in time $O(\delta n^{\omega+o(1)})$ where δ is the maximum degree of the input basis, n is the dimension, and ω is the exponent of matrix multiplication. Our implementation uses the algorithm of Mulders and Storjohann [27] which runs in time $O(n^3 \delta^2)$ but is much simpler and easier to implement, and yields excellent running times for the input sizes we care about.

If step 5 does not abort, then there is guaranteed to be a unique set of polynomials satisfying the requirements; that is, we are in the unique decoding case. As we will

see later, if the errors are random, this step aborts only with very low probability.

Without any imposed structure on the codeword polynomials, the Cohn-Heninger algorithm is *heuristic*; that is, they conjecture that it will succeed for sufficiently random input. The linear version that we use here is also heuristic: there are adversarial inputs on which it may fail. However, we observe in experiments (see Section 5.3) that the heuristic assumption holds with high probability for *random* inputs, which is the situation we need for our cryptographic purposes here. We conjecture based on the experimental evidence presented in Section 5.3 that the probability of failure depends only on the size of the underlying field \mathbb{F} . In particular, the algorithm will work with high probability for random polynomials if the errors are uncorrelated. We will see later that we can enforce this restriction in our protocol even in the case of Byzantine servers.

2.3.3 Optimality

Relating this to our PIR application, we will be able to use this algorithm to correctly decode the results of t -private PIR queries. If k servers respond to us, of which v are Byzantine (so $h = k - v$ are honest), then this algorithm will succeed with high probability after we query for m blocks, satisfying $v \leq \frac{m}{m+1}(k-t-1)$, or equivalently, $m \geq \frac{v}{h-t-1}$. That is, for m large enough, we can handle any number of Byzantine servers $v < k-t-1$. We note that this bound on v is *optimal*—if $v = k-t-1$, or equivalently, $h = t+1$, then *any* subset of $t+1$ servers' responses will form a polynomial of degree at most t . This means that the number of possible valid blocks will always be exponential, and no polynomial-time algorithm could hope to address this case.

2.4 Dynamic programming

In practice, each of the algorithms we have described above has different performance characteristics for different inputs. Thus in our implementation, we achieve the best performance by assembling all of them together into a *portfolio algorithm*. This algorithm optimistically attempts to decode a given input using Lagrange interpolation, and if that fails, uses a dynamic program with timing measurements to fall back to an optimal sequence of decoding algorithms. See Section 5.2 for more details.

3 Protocols for PIR

In this section, we will introduce the ideas from previous PIR protocols that will form the basis for our protocol.

3.1 Database queries as linear algebra

We begin with a general mathematical setting of the PIR schemes we will be considering.

Our database D is structured as an $r \times s$ matrix with r rows. Each row represents one block of the database, and consists of s words of w bits each. The database D resides on a remote server. The client wishes to retrieve one block (row) of the database from the server.

$$D = \begin{bmatrix} \mathbf{w}_{11} & \mathbf{w}_{12} & \cdots & \mathbf{w}_{1s} \\ \mathbf{w}_{21} & \mathbf{w}_{22} & \cdots & \mathbf{w}_{2s} \\ \vdots & \vdots & & \vdots \\ \mathbf{w}_{r1} & \mathbf{w}_{r2} & \cdots & \mathbf{w}_{rs} \end{bmatrix}$$

One non-private protocol for the client to retrieve row β of the database would be to transmit the vector \mathbf{e}_β consisting of all zeros except for a single 1 in coordinate β to the server. The server considers \mathbf{e}_β as a row vector and computes the product $\mathbf{e}_\beta \cdot D$, which it sends back to the client.

$$\begin{bmatrix} 0 & 0 & \cdots & 1 & \cdots & 0 \end{bmatrix} \begin{bmatrix} \mathbf{w}_{11} & \mathbf{w}_{12} & \cdots & \mathbf{w}_{1s} \\ \mathbf{w}_{21} & \mathbf{w}_{22} & \cdots & \mathbf{w}_{2s} \\ \vdots & \vdots & & \vdots \\ \mathbf{w}_{r1} & \mathbf{w}_{r2} & \cdots & \mathbf{w}_{rs} \end{bmatrix} = [\mathbf{w}_{\beta 1} \quad \mathbf{w}_{\beta 2} \quad \cdots \quad \mathbf{w}_{\beta s}]$$

We will show how to construct two information-theoretic PIR schemes that modify this basic scheme to retrieve blocks from the database without revealing the query or result to an adversary.

3.2 A simple PIR scheme due to Chor et al.

We next present a simple PIR scheme due to Chor et al. [10] We begin with the same setup as above. In this protocol, the words will be single bits, so $w = 1$, and D is an $r \times s$ matrix of bits. Since we will be constructing information-theoretic PIR, we will be querying more than one server. We will require that not all of the servers are colluding to reveal the client's query. Each of the $\ell \geq 2$ servers gets a copy of D .

A client wishing to retrieve block β of the database generates the basis vector \mathbf{e}_β as above to select coordinate β . Then in order to hide this query vector from the servers, the client picks $\ell - 1$ vectors $\mathbf{v}_1, \dots, \mathbf{v}_{\ell-1}$ uniformly at random from $GF(2)^r$ (that is, $\ell - 1$ uniformly random r -bit binary strings), and computes $\mathbf{v}_\ell = \mathbf{e}_\beta \oplus (\mathbf{v}_1 \oplus \cdots \oplus \mathbf{v}_{\ell-1})$. \mathbf{v}_ℓ will be a uniformly random (though not independent) r -bit string, as $\ell \geq 2$.

The client sends \mathbf{v}_i to server i for each $1 \leq i \leq \ell$. Server i computes the product $\mathbf{r}_i = \mathbf{v}_i \cdot D$, which is the same as

setting \mathbf{r}_i to be the XOR of those blocks j in the database for which the j^{th} bit of \mathbf{v}_i is 1. Each server i returns \mathbf{r}_i to the client.

The client XORs the results to obtain $\mathbf{r} = \mathbf{r}_1 \oplus \dots \oplus \mathbf{r}_\ell = (\mathbf{v}_1 \oplus \dots \oplus \mathbf{v}_\ell) \cdot D = \mathbf{e}_\beta \cdot D$, which is the β^{th} block of the database, as required.

Note that this scheme is $(\ell - 1)$ -private; that is, no combination of $\ell - 1$ or fewer servers has enough information to determine i from the information they receive from, or send to, the client. Choosing $r = s = \sqrt{n}$ yields a total communication of $2\ell\sqrt{n}$ bits to privately retrieve a block of size \sqrt{n} bits.

3.3 Goldberg's PIR scheme

Chor's scheme, above, is not robust; if even one server fails to respond, the client cannot reconstruct her answer. Further, it is not Byzantine robust; if one server gives the wrong answer, then the client not only will reconstruct the wrong block, but the client will be unable to determine which server misbehaved.

Goldberg [19] modified Chor's scheme to achieve both robustness and Byzantine robustness. Rather than working over $GF(2)$ (binary arithmetic), his scheme works over a larger field \mathbb{F} , where each element can represent w bits (so $w = \lceil \lg |\mathbb{F}| \rceil$). The database D is then an $r \times s$ matrix of elements of \mathbb{F} . In Goldberg's simplest construction, as with Chor's scheme, each of $\ell \geq 2$ servers gets a copy of the database.

To transform this into a t -private PIR protocol, the client uses (ℓ, t) Shamir secret sharing to share the vector $\mathbf{e}_\beta \in \mathbb{F}^r$ into ℓ independent shares $(\alpha_1, \mathbf{v}_1), \dots, (\alpha_\ell, \mathbf{v}_\ell)$. That is, the client creates r random degree- t polynomials f_1, \dots, f_r satisfying $f_j(0) = \mathbf{e}_\beta[j]$ and chooses ℓ distinct non-zero elements $\alpha_i \in \mathbb{F}$. Server i 's share will be the vector $\mathbf{v}_i = \langle f_1(\alpha_i), \dots, f_r(\alpha_i) \rangle$.

Each server then computes the product $\mathbf{r}_i = \mathbf{v}_i \cdot D = \langle \sum_j f_j(\alpha_i) \mathbf{w}_{j1}, \dots, \sum_j f_j(\alpha_i) \mathbf{w}_{js} \rangle \in \mathbb{F}^s$.

$$\begin{aligned} & \begin{bmatrix} f_1(\alpha_i) & \dots & f_r(\alpha_i) \end{bmatrix} \begin{bmatrix} \mathbf{w}_{11} & \mathbf{w}_{12} & \dots & \mathbf{w}_{1s} \\ \mathbf{w}_{21} & \mathbf{w}_{22} & \dots & \mathbf{w}_{2s} \\ \vdots & \vdots & & \vdots \\ \mathbf{w}_{r1} & \mathbf{w}_{r2} & \dots & \mathbf{w}_{rs} \end{bmatrix} \\ &= \left[\sum_j f_j(\alpha_i) \mathbf{w}_{j1} \quad \dots \quad \sum_j f_j(\alpha_i) \mathbf{w}_{js} \right] \end{aligned}$$

By the linearity property of Shamir secret sharing, since $\{(\alpha_i, \mathbf{v}_i)\}_{i=1}^\ell$ is a set of Shamir secret shares of \mathbf{e}_β , $\{(\alpha_i, \mathbf{r}_i)\}_{i=1}^\ell$ will be a set of Shamir secret shares of $\mathbf{e}_\beta \cdot D$, which is the β^{th} block of the database. Looking at it another way, the vector $\langle \mathbf{r}_1[q], \mathbf{r}_2[q], \dots, \mathbf{r}_\ell[q] \rangle$ is a Reed-Solomon codeword encoding the polynomial

$g_q = \sum_j f_j \mathbf{w}_{jq}$, and the client wishes to compute $g_q(0)$ for each $1 \leq q \leq s$.

However, some of the servers may be down or Byzantine, so some of the shares returned by these servers may be missing or incorrect. Goldberg's scheme first optimistically assumes that all of the servers that replied gave correct responses, and uses Lagrange interpolation to attempt to reconstruct the database row (his EASYRECOVER algorithm). He bases this optimistic assumption on the fact that Byzantine servers are discovered by his scheme, which disincentivizes servers to act maliciously. If the optimism is not justified, however, his scheme then uses the Guruswami-Sudan algorithm [22] (his HARDRECOVER algorithm) to do error correction (see Section 2.3).

This scheme is t -private, and the Guruswami-Sudan algorithm can correct $v < k - \lfloor \sqrt{kt} \rfloor$ incorrect server responses. Choosing $r = s = \sqrt{nw}/w$ yields a total communication of $(k + \ell) \sqrt{nw}$ bits to privately retrieve a block of size \sqrt{nw} bits.

Goldberg's scheme also allows for an extension called τ -independence [16], in which the database itself is secret shared among the ℓ servers, so that no coalition of τ or fewer servers can learn the contents of the database. We will omit the details for ease of presentation, but our scheme extends naturally to this scenario as well.

4 Our algorithm

Our algorithm follows the same general idea as Goldberg during the client-server interaction and Shamir secret sharing. We change the way that queries are randomized and make improvements to the client-side processing to greatly improve robustness and the speed of processing.

Goldberg's block reconstruction technique uses the Guruswami-Sudan algorithm to reconstruct the block a single word at a time. However, we can achieve better error-correction bounds with the algorithm of Section 2.3.2 by considering multiple blocks simultaneously. This takes advantage of the observation that a server is either Byzantine or not; if it is not, it will give correct results for every query.

If the Reed-Solomon codewords the client expects to receive from the servers are $\langle R_1^*[q], R_2^*[q], \dots, R_\ell^*[q] \rangle$ for $1 \leq q \leq s$, what it actually receives may differ because some number of servers may be down, and some further number may be Byzantine. Of the ℓ servers, it may only receive a response from k of them, and of those, v may be incorrect.

If the client receives $\langle R_1[q], R_2[q], \dots, R_\ell[q] \rangle$ for $1 \leq q \leq s$, then:

$$\begin{bmatrix} f_1(\alpha_1) & \dots & f_r(\alpha_1) \\ \vdots & & \vdots \\ f_1(\alpha_\ell) & \dots & f_r(\alpha_\ell) \end{bmatrix} \begin{bmatrix} \mathbf{w}_{11} & \mathbf{w}_{12} & \dots & \mathbf{w}_{1s} \\ \mathbf{w}_{21} & \mathbf{w}_{22} & \dots & \mathbf{w}_{2s} \\ \vdots & \vdots & & \vdots \\ \mathbf{w}_{r1} & \mathbf{w}_{r2} & \dots & \mathbf{w}_{rs} \end{bmatrix} = \begin{bmatrix} R_1[1] & R_1[2] & \dots & R_1[s] \\ \vdots & \vdots & & \vdots \\ R_\ell[1] & R_\ell[2] & \dots & R_\ell[s] \end{bmatrix} = \begin{bmatrix} g_1(\alpha_1) & g_2(\alpha_1) & \dots & g_s(\alpha_1) \\ \vdots & \vdots & & \vdots \\ g_1(\alpha_\ell) & g_2(\alpha_\ell) & \dots & g_s(\alpha_\ell) \end{bmatrix}$$

Figure 1: Our PIR protocol illustrated. Each row of the leftmost matrix corresponds to a Shamir secret share of the database row being queried; each column of the rightmost two matrices corresponds to a Reed-Solomon codeword encoding a word of the queried database row. The client sends the i^{th} row of the leftmost matrix to server i and expects to receive the i^{th} row of the rightmost matrix in reply.

- For $\ell - k$ values of i , $R_i[q] = \perp$ for all q (these are the down servers)
- For at least $h = k - v$ values of i , $R_i[q] = R_i^*[q]$ for all q (these are the honest servers)
- For the remaining at most v values of i , $R_i[q] = R_i^*[q] + \Delta_{iq}$ for error terms Δ_{iq} (these are the Byzantine servers)

4.1 Randomizing queries

In order to use the algorithm of section 2.3.2, we need to ensure that the Byzantine servers produce *random* errors; that is, that the Δ_{iq} terms are randomly and independently chosen in the m codewords we supply to that algorithm. We make no a priori assumptions on the types of errors that the Byzantine servers may produce, but we will randomize the algorithm to cause any kind of spurious or malicious error to appear random.

To accomplish this, we make the following modification to Goldberg’s protocol: the client chooses a uniformly random non-zero element $c_i \in_R \mathbb{F}^*$ for each server and sends the server a “blinded” query $c_i Q_i = \langle c_i f_1(\alpha_i), \dots, c_i f_r(\alpha_i) \rangle$ instead of just Q_i . When the server i returns a vector R'_i , the client unblinds it by dividing by c_i to yield $R_i = c_i^{-1} R'_i$.

This ensures that if the server’s response R'_i was the correct response to query $c_i Q_i$, then R_i will be the correct response to query Q_i . Further, for a Byzantine server, the error $\Delta'_{iq} = R'_i[q] - c_i R_i^*[q]$ it maliciously introduces will be randomized unpredictably by the client to $\Delta_{iq} = c_i^{-1} \Delta'_{iq} = R_i[q] - R_i^*[q]$.

Note, however, that different errors within the same server’s response to a single query are *not* independently randomized; if a Byzantine server just adds a constant C to each word of the correct result $c_i R_i^*$ before returning it to the client, the client will see a result R_i that has had the constant $c_i^{-1} C$ added to each word of the correct result.

Errors from different servers, or from different queries, though, *are* independent, and it is this independence we leverage to get the linear multi-polynomial al-

gorithm in Section 2.3.2 to work: after m queries, we will have m responses each with independently random errors, and we can use the algorithm to decode them simultaneously with high probability.

4.2 Reconstructing responses

After unblinding, the client possesses responses R_i from k servers; for ease of notation, suppose they are servers 1 through k . Each R_i is a vector $\langle R_i[1], \dots, R_i[s] \rangle$ where s is the number of words (elements of \mathbb{F}) in one database block. Each $\langle R_1[q], \dots, R_k[q] \rangle$ for $1 \leq q \leq s$ is a Reed-Solomon codeword with errors (the $\ell - k$ non-responding servers’ entries having been removed) encoding a polynomial g_q ; see Figure 1. The client’s desired block is $\langle g_1(0), \dots, g_s(0) \rangle$.

As with Goldberg’s scheme, the client first optimistically attempts to reconstruct each g_q using Lagrange interpolation on the points $\{(\alpha_1, R_1[q]), \dots, (\alpha_k, R_k[q])\}$ to see if the resulting polynomial has degree at most t . If there were no Byzantine servers, this will be successful. For any g_j for which Lagrange interpolation fails, we apply an escalating sequence of error-correction algorithms from Section 2.3 to attempt to recover g_j . Our implementation ties these together in a portfolio algorithm; see Section 5.2. If at any time, the error correction algorithm identifies a particular server as Byzantine, that server’s results are discarded for all future computations.

If there is still at least one g_j which was not yet able to be reconstructed, any one such unsuccessfully decoded codeword $\langle R_1[q], \dots, R_k[q] \rangle$ is stored for later reconstruction, along with the current state of the computation. The client’s requested block will not be available at this time.

The client can then do PIR requests for more blocks of the database. If it was interested in multiple blocks, it can just request those. Otherwise, it can re-request blocks it has not yet successfully decoded. Note that the properties of PIR ensure that the servers cannot tell whether a request is for a repeated block or a fresh one. Each time, the client either receives its desired block (from the Lagrange interpolation or error correcting portfolio algo-

rithms) or another codeword gets stored for later reconstruction.

When $m = \lceil \frac{v}{h-t-1} \rceil \leq v$ such codewords have been collected, we can apply the algorithm of Section 2.3.2. Since the stored codewords have independent errors, the algorithm will succeed with high probability. At that point, all m stored computations can be concluded, the m blocks will be returned to the client, and the $v < k-t-1$ Byzantine servers will be identified. The client can then avoid those servers in the future.

Note that the decoding algorithm is randomized, so there is a small chance of failure even when the client has collected the results of m queries. In this case, the client can continue to collect queries and construct new codewords until the algorithm succeeds.

Algorithm 2 summarizes the process.

Algorithm 2 Robust PIR Protocol

Goal: Client wishes to query row β from database D stored on ℓ servers.

- 1: Client chooses ℓ distinct non-zero elements $\alpha_1, \dots, \alpha_\ell \in \mathbb{F}^*$.
- 2: Client chooses r random degree- t polynomials $f_1, \dots, f_r \in \mathbb{F}[x]$ satisfying $f_j(0) = 1$ for $j = \beta$ and $f_j(0) = 0$ otherwise.
- 3: Client chooses ℓ random non-zero elements $c_1, \dots, c_\ell \in \mathbb{F}^*$.
- 4: Client sends the vector

$$Q_i = \langle c_i f_1(\alpha_i), c_i f_2(\alpha_i), \dots, c_i f_r(\alpha_i) \rangle$$

to server i .

- 5: Server i receives vector Q_i .
 - 6: Server i sends the product $R'_i = Q_i \cdot D$ to client.
 - 7: Client receives R'_1, \dots, R'_ℓ .
 - 8: Client computes $R_i = c_i^{-1} R'_i$ for each i .
 - 9: Client considers vectors $S_q = \langle R_1[q], \dots, R_\ell[q] \rangle$ as received Reed-Solomon codewords and uses the algorithms from Section 2.3 to recover word q of row β of D .
 - 10: If the recovery algorithm fails, postpone decoding until $m = \lceil \frac{v}{h-t-1} \rceil \leq v$ blocks have been requested (requesting blocks multiple times if necessary). Then use the algorithm from Section 2.3.2 to recover all of the blocks simultaneously.
-

5 Implementation and experiments

We implemented the algorithm described in this paper as an extension of Goldberg's implementation of his protocol, available as the Percy++ project on Source-

Forge [18]. The software is implemented in C++ using the NTL library [35].

In this paper we are concerned with the speed of the client-side block reconstruction operation in the presence of Byzantine servers. Our work does not change the server side of Goldberg's protocol in any way; to see speeds for the server-side operations, see Olumofin and Goldberg's 2011 paper [29].

5.1 Choice of underlying field

Goldberg's 2007 work used a 128-bit prime field as the field \mathbb{F} . Subsequent releases of Percy++, however, were able to use different fields, including prime fields of different sizes as well as $GF(2^8)$. This last field turns out to be a very efficient choice, as additions in this field can be implemented as XOR operations and multiplications are simple lookups in a 64 KB table.

Our implementation of our protocols uses C++ templates to abstract the field \mathbb{F} , making it very easy to work over any desired field.

5.2 Portfolio algorithms for decoding

Our implementation assembles the error correction algorithms described in Section 2.3 into a portfolio algorithm [20] to do efficient decoding. We use dynamic programming to choose an optimal sequence of decoding algorithms to try.

Each of the error correction algorithms we use is characterized by the tuple (k, t, h) with $k \geq h > t$: we wish to find a polynomial of degree at most t that passes through at least h of the k input points.

We have a few different choices of algorithm to solve this problem directly:

Berlekamp-Welch: If $h > \frac{k+t}{2}$, we can use the Berlekamp-Welch algorithm to find the unique polynomial solution, if it exists. This algorithm is quite fast, and we use it whenever it is applicable.

Guruswami-Sudan: If $h > \sqrt{kt}$, we can use the Guruswami-Sudan algorithm to find all solutions. It turns out this algorithm is very inefficient if $h^2 - kt$ is small; for the parameter sizes we care about, we avoid this algorithm if this value is less than 10.

Brute force: Lagrange interpolate each subset of $t+1$ points to form a polynomial of degree t , and see if it passes through at least h points. This works for any $h > t$, but is inefficient if $\binom{k}{t+1}$ is large.

In addition, we have three strategies to attempt to solve a particular instance with parameters (k, t, h) by recursively solving smaller instances and combining the results. Let $C(k, t, h)$ represent the expected time cost to

solve an instance of this size; we will bound this cost as a function of the costs of solving smaller instances.

Guess g incorrect points: If the client can guess that a particular point is wrong (that is, that the server that provided that point is Byzantine), then it can just throw away the point, and solve the remaining problem, with parameters $(k-1, t, h)$. In general, it might guess g points to be wrong, and solve a problem of size $(k-g, t, h)$. Since at least h of the original k points are correct, for any set of $h+g$ points, there exists a subset of g points that can be removed from the original k so that there are at least h correct points in the $k-g$ points remaining. Thus by recursively solving $\binom{h+g}{g}$ smaller instances and combining the results, we are guaranteed to find all solutions to our original problem. Thus we see that

$$C(k, t, h) \leq \min_g \binom{h+g}{g} \cdot C(k-g, t, h)$$

Guess g correct points: Conversely, the client might guess that a particular subset of g points are all correct (that is, that the servers that provided them are honest), and recursively try to find a polynomial of degree at most $t-g$ that passes through at least $h-g$ of the remaining $k-g$ points.¹ Similar to the above, we get that we need to recursively solve $\binom{k-h+g}{g}$ subproblems with parameters $(k-g, t-g, h-g)$, so we get

$$C(k, t, h) \leq \min_g \binom{k-h+g}{g} \cdot C(k-g, t-g, h-g)$$

Guess whether d points are correct or incorrect: The above strategies may not be helpful if the binomial coefficients are large. Our final strategy is to pick a fixed set of d points. For all g , and for all choices of g correct and $d-g$ incorrect points within that set, we recursively try to find polynomials of degree at most $t-g$ that pass through at least $h-g$ of the remaining $k-d$ points. As before, we get that

$$C(k, t, h) \leq \min_d \sum_g \binom{d}{g} \cdot C(k-d, t-g, h-g)$$

Given these three algorithms to directly solve the problem, and three strategies to indirectly solve it by combining solutions to smaller instances, we use dynamic programming to build a table of the best strategy to use to

¹The remaining points are actually slightly modified before solving recursively. If (α^*, y^*) is guessed to be correct, then each other point (α_i, y_i) is modified to $(\alpha_i, \frac{y_i - y^*}{\alpha_i - \alpha^*})$ before recursively solving. If $f(x)$ interpolates at least $h-1$ points of the latter form, then $f(x) \cdot (x - \alpha^*) + y^*$ interpolates the corresponding $h-1$ original points and also the guessed point.

minimize the expected run time for inputs of each combination of parameters (k, t, h) . We measure the runtimes of the direct algorithms experimentally, and compute the times for the indirect strategies. We pick the lowest result of the six, and set $C(k, t, h)$ to that value. We currently do this in a precomputation step for all $k \leq 25$ and give the PIR client access to this table.

5.3 Multi-polynomial decoding

We also implemented the linear multi-polynomial decoding algorithm described in Section 2.3.2 as an extension of Percy++ using C++ and the NTL library. For the lattice reduction step, our implementation uses the lattice reduction algorithm by Mulders and Storjohann [27]. Although its theoretical runtime is not the fastest known, we chose this algorithm because of its simplicity. After the lattice reduction, the implementation then solves the resulting system of linear equations using Gaussian elimination.

As previously mentioned, if the errors are random, then there is a very low probability that this algorithm will fail. Based on experimental investigation, we conjecture the probability of failure is, to first order, $\left(\frac{1}{|\mathbb{F}|}\right)^{m(h-t-1)-v+1}$. (Recall from Section 2.3.3 that in order for the algorithm to work, $m \geq \frac{v}{h-t-1}$, or equivalently, $m(h-t-1) - v \geq 0$.) See the appendix for more details on these experiments. This probability of failure falls within the confidence intervals for all of our tests with $|\mathbb{F}| \geq 256$. We also ran tests with an extremely small field of $|\mathbb{F}| = 16$, and found that failures in that field occurred slightly (but statistically significantly) more often than our conjecture predicts. This leads us to believe that there is a missing second-order term in our conjecture, which is negligible for reasonable field sizes, but significant for tiny fields. We hope to nail down the missing term in future work.

In the cases where the linear multi-polynomial algorithm does fail, our algorithm will wait until another block is requested and then try again. This increases m by one and reduces the probability that the algorithm will fail by a factor of $|\mathbb{F}|^{h-t-1}$; therefore, since $h-t \geq 2$, the probability it will fail a second time is extremely tiny.

5.4 Measuring improvements to Percy++

In his 2007 paper, Goldberg measures the performance of his protocols using a Lenovo T60p laptop computer with a 2.16 GHz Intel dual-core CPU running Ubuntu Linux [19]. For the purposes of comparison, we have performed our measurements on a machine of the same model and similar Ubuntu Linux configuration. Goldberg reports that the implementation of

Table 1: Measuring improvements to Percy++’s client-side decoding algorithms. For these measurements we ran 100 trials using the parameters $(k, t, h) = (20, 10, 15)$.

Implementation	Algorithm	Field	Time
timing reported by Goldberg [19]	Guruswami-Sudan in MuPAD	128-bit prime	“several minutes”
Percy++	Guruswami-Sudan in C++	128-bit prime	9000 ± 3000 ms
Percy++	Guruswami-Sudan in C++	$GF(2^8)$	3000 ± 900 ms
this work	Cohn-Heninger in C++ with $m = 2$ blocks	128-bit prime	2.2 ± 0.9 ms
this work	Cohn-Heninger in C++ with $m = 2$ blocks	$GF(2^8)$	1.3 ± 0.4 ms

his HARDRECOVER algorithm takes “several minutes” when using the values $(k, t, h) = (20, 10, 15)$ [19].

Since the writing of that paper, the Percy++ software has improved. The first improvement was to implement the parts of the HARDRECOVER subroutine previously written using MuPAD in native C++. We timed HARDRECOVER 100 times using only this change, and found the running time reduced to 9000 ± 3000 ms.

The other improvement in the latest version of Percy++ is to use $GF(2^8)$ as the underlying field, rather than a 128-bit prime field. With this change, we again measured HARDRECOVER 100 times, and found the running time further reduced to 3000 ± 900 ms.

Finally, using the implementation of our algorithm described in Section 4 we further improve the running time, again tested with 100 trials using multi-polynomial decoding with just $m = 2$ blocks. With a 128-bit prime field, our algorithm completes in 2.2 ± 0.9 ms; with $GF(2^8)$, in just 1.3 ± 0.4 ms.

This is a reduction of over three orders of magnitude in client-side decoding time versus the latest software, and of over five orders of magnitude versus Goldberg’s 2007 reported measurements. This comes at a cost of fetching just two blocks instead of one — something the client is likely to have done anyway. The results are summarized in Table 1.

5.5 New client-side measurements

We next outline the results of measurements taken of the implementation of our new algorithm described in Section 4. These measurements are only on the client-side decoding operations. For these measurements we used a server with a 2.40 GHz Intel dual-core CPU running Ubuntu Linux. For each case, we ran at least 100 trials, all using only a single core. We used the field $GF(2^8)$ for all experiments in this section.

To illustrate the improvements that our algorithm provides, we compare time measurements for four algorithms in Figure 2:

- the potentially exponential-time brute force decoding algorithm

- the Guruswami-Sudan list decoding algorithm from the latest release of Percy++
- the single-polynomial dynamic programming algorithm described in Section 5.2; and
- the linear multi-polynomial algorithm described in Section 5.3

Note that the data is plotted on a log scale so that the results can be easily compared, even though they span five orders of magnitude.

Observe that the Guruswami-Sudan algorithm only works when the number of Byzantine servers $v = k - h$ is less than $k - \lfloor \sqrt{kt} \rfloor$, and its running time blows up as v nears that bound. Past that bound, with more Byzantine servers, the only prior way for the client to decode the result was to use the brute-force decoding algorithm. Now, we can see that our single-polynomial dynamic programming algorithm and our multi-polynomial decoding algorithm both outperform the brute-force algorithm, often substantially. For example, in Figure 2(c) we see that for eight Byzantine servers with $(k, t) = (20, 10)$, the Guruswami-Sudan algorithm is ineffective, and the brute-force algorithm takes about 10 seconds. Meanwhile, our dynamic programming algorithm takes about 1.5 seconds, and our multi-polynomial decoding algorithm takes about 6 *milliseconds*.

The multi-polynomial algorithm comes at a cost, however, of forcing the client to fetch multiple blocks. In this case, $m = \lceil \frac{v}{h-t-1} \rceil = 8$ blocks. If the client were going to fetch that many blocks anyway, there is no additional overhead to the scheme. Otherwise, the client may have to request some blocks multiple times. In the worst case, the client only wishes to fetch one block, and there are $v = k - t - 2$ Byzantine servers. In this worst case, $h = t + 2$, and the client must request its desired block $m = v = k - t - 2$ times before it will be able to decode it. Note that, even when multiple blocks are retrieved from the servers, our multi-polynomial algorithm is run only *once*, in order to distinguish the honest servers from the misbehaving ones.

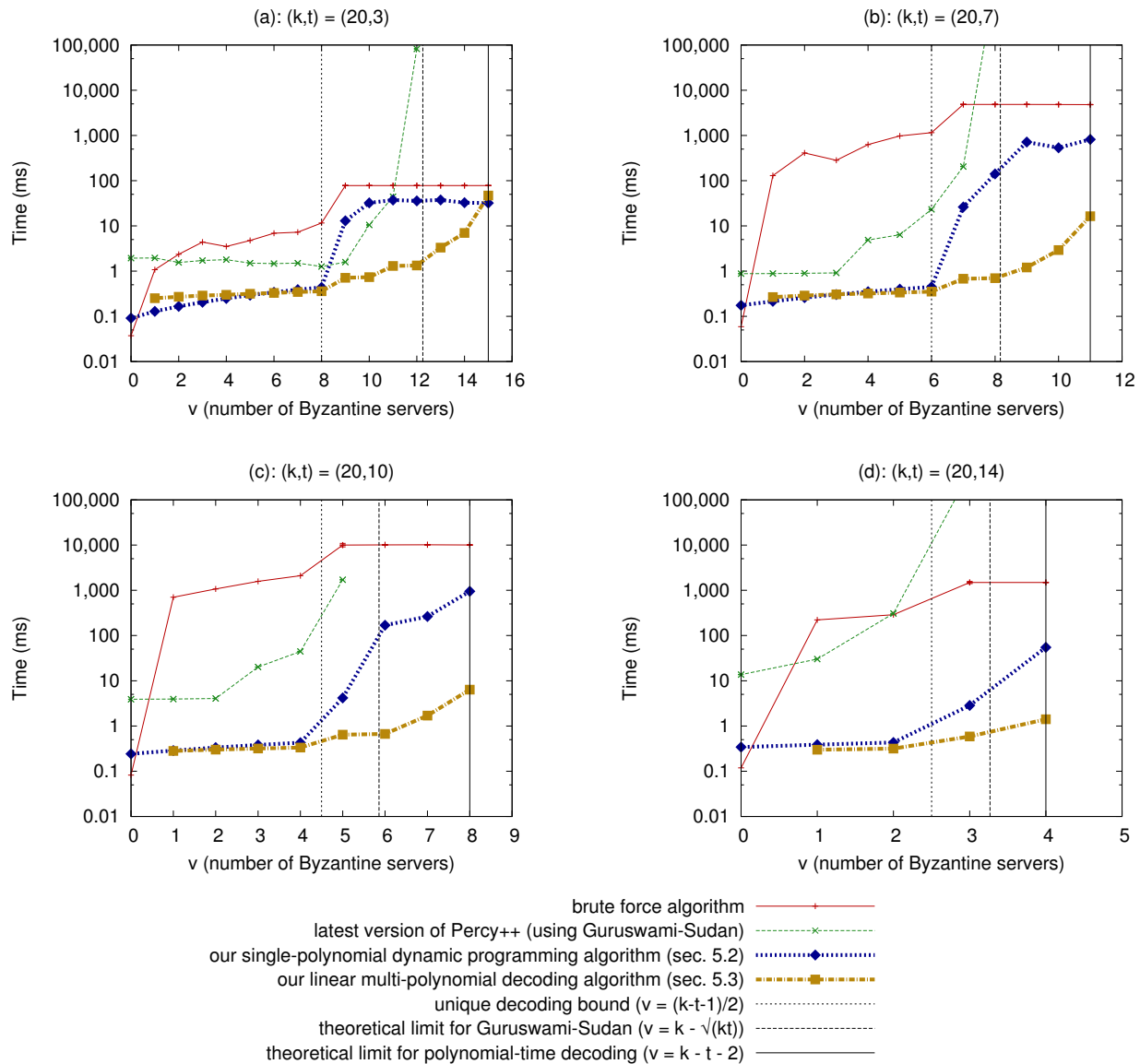


Figure 2: Timing measurements for the client-side decoding algorithms discussed in this paper for different parameters. We ran each algorithm 100 times for each choice of feasible parameters and plot the mean running times in milliseconds. Note that times are plotted on a log scale.

The three vertical lines on each plot show the unique decoding radius for Reed-Solomon codes, on the left, the theoretical bound past which the Guruswami-Sudan algorithm used in Percy++ fails, in the middle, and the theoretical bound past which efficient decoding with any algorithm is impossible, on the right.

The main results of this paper are to give two client-side decoding algorithms that outperform Guruswami-Sudan in its feasible region, and allow us to extend the range of efficient client-side decoding to the region of interest between the two vertical lines on the right. Note that the Guruswami-Sudan algorithm performs much slower in practice than the Berlekamp-Welch algorithm used by our dynamic programming portfolio algorithm within the unique decoding radius, and its running time blows up very quickly for parameters approaching its theoretical limit.

The running times of the brute force algorithm within the unique decoding radius have extremely high variance; we do not plot error bars for those timings as they obscure the entire rest of the plot. We *do* plot error bars for all other points, but they are generally too small to see.

6 Conclusions

We have improved the client side of Goldberg’s 2007 Byzantine-robust information-theoretic private information retrieval protocol to use state-of-the-art decoding algorithms to improve the Byzantine robustness of the protocol to its theoretical limit. We did this using decoding algorithms that are able to take advantage of decoding information in multiple blocks of data simultaneously, observing that in practical scenarios, clients will often be interested in more than one block at a time.

We implemented our protocol and found that it is very fast in practice: *several thousand times* faster than previous protocols, and usually less than 10 ms for the parameter choices in our experiments.

Combined with fast processing on the server side [29] and scenarios in which the database servers are not in collusion [28], we can see that information-theoretic private information retrieval can be practical even in highly adversarial settings.

Acknowledgements

We thank Dan Bernstein for pointing out the connections between multi-polynomial error correction and some kinds of PIR. We thank Mark Giesbrecht and Arne Storjohann for their pointers on implementing polynomial lattice basis reduction. This material is based upon work supported by NSERC, Mprime, the National Science Foundation under Award No. DMS-1103803, and the MURI program under AFOSR Grant No. FA9550-08-1-0352. Finally, we thank the Shared Hierarchical Academic Research Computing Network (SHARCNET) and Compute/Calcul Canada for the computing cluster on which we ran the experiments in Section 5.3 and the appendix.

References

- [1] C. Aguilar Melchor and P. Gaborit. A lattice-based computationally-efficient private information retrieval protocol. In *Western European Workshop on Research in Cryptology (WEWoRC2007), Bochum, Germany. Book of Abstracts*, pages 50–54, 2007.
- [2] D. Asonov. Private Information Retrieval: An overview and current trends. In *ECDPvA Workshop*, 2001.
- [3] A. Beimel and Y. Stahl. Robust information-theoretic private information retrieval. In *Proceedings of the 3rd International Conference on Security in Communication Networks (SCN’02)*, pages 326–341, 2003.
- [4] A. Beimel and Y. Stahl. Robust information-theoretic private information retrieval. *Journal of Cryptology*, 20:295–321, 2007.
- [5] E. Berlekamp and L. Welch. Error correction of algebraic block codes. US Patent Number 4,633,470, 1986.
- [6] R. Carback, D. Chaum, J. Clark, J. Conway, A. Essex, P. S. Herrnson, T. Mayberry, S. Popoveniuc, R. L. Rivest, E. Shen, A. T. Sherman, and P. L. Vora. Scantegrity II Municipal Election at Takoma Park: The First E2E Binding Governmental Election with Ballot Privacy. In *19th USENIX Security Symposium*, pages 291–306, 2010.
- [7] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–90, Feb. 1981.
- [8] B. Chor and N. Gilboa. Computationally private information retrieval (extended abstract). In *29th annual ACM Symposium on Theory of Computing (STOC’97)*, pages 304–313, 1997.
- [9] B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. Technical Report TR CS0917, Department of Computer Science, Technion, Israel, 1997.
- [10] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *36th Annual IEEE Symposium on Foundations of Computer Science (FOCS’95)*, pages 41–50, oct 1995.
- [11] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *J. ACM*, 45:965–981, November 1998.
- [12] H. Cohn and N. Heninger. Approximate common divisors via lattices. Cryptology ePrint Archive, Report 2011/437, 2011. <http://eprint.iacr.org/>.
- [13] C. Devet, I. Goldberg, and N. Heninger. Optimally Robust Private Information Retrieval. Cryptology ePrint Archive, Report 2012/083, 2012.
- [14] R. Dingledine, N. Mathewson, and P. Syverson. Tor: the second-generation onion router. In *13th USENIX Security Symposium*, 2004.
- [15] W. I. Gasarch. A survey on private information retrieval (column: Computational complexity). *Bulletin of the EATCS*, 82:72–107, 2004.

- [16] Y. Gertner, S. Goldwasser, and T. Malkin. A Random Server Model for Private Information Retrieval. In *2nd International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 200–217, 1998.
- [17] P. Giorgi, C.-P. Jeannerod, and G. Villard. On the complexity of polynomial matrix computations. In *2003 International Symposium on Symbolic and Algebraic Computation*, pages 135–142, 2003.
- [18] I. Goldberg. Percy++ project on sourceforge. <http://percy.sourceforge.net>. Accessed February 2012.
- [19] I. Goldberg. Improving the robustness of private information retrieval. In *2007 IEEE Symposium on Security and Privacy*, pages 131–148, 2007.
- [20] C. Gomes and B. Selman. Algorithm Portfolios. *Artificial Intelligence*, 126(1):43–62, 2001.
- [21] V. Guruswami and A. Rudra. Explicit codes achieving list decoding capacity: Error-correction with optimal redundancy. *IEEE Transactions on Information Theory*, 54(1):135–150, 2008.
- [22] V. Guruswami and M. Sudan. Improved decoding of Reed-Solomon and algebraic-geometric codes. *39th Annual IEEE Symposium on Foundations of Computer Science (FOCS'98)*, pages 28–39, 1998.
- [23] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *38th Annual Symposium on Foundations of Computer Science (FOCS'97)*, pages 364–373, 1997.
- [24] H. W. Lenstra, A. K. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.
- [25] S. Micali, C. Peikert, M. Sudan, and D. A. Wilson. Optimal Error Correction Against Computationally Bounded Noise. In *2nd Theory of Cryptography Conference*, pages 1–16, February 2005.
- [26] P. Mittal, F. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg. PIR-Tor: Scalable Anonymous Communication Using Private Information Retrieval. In *20th USENIX Security Symposium*, pages 475–490, 2011.
- [27] T. Mulders and A. Storjohann. On lattice reduction for polynomial matrices. *Journal of Symbolic Computation*, 35(4):377 – 401, 2003.
- [28] F. Olumofin and I. Goldberg. Privacy-preserving queries over relational databases. In *10th International Privacy Enhancing Technologies Symposium*, pages 75–92, 2010.
- [29] F. Olumofin and I. Goldberg. Revisiting the Computational Practicality of Private Information Retrieval. In *15th International Conference on Financial Cryptography and Data Security*, pages 158–172, 2011.
- [30] F. Olumofin, P. Tysowski, I. Goldberg, and U. Hengartner. Achieving efficient query privacy for location based services. In *10th International Privacy Enhancing Technologies Symposium*, pages 93–110, 2010.
- [31] F. Parvaresh and A. Vardy. Correcting errors beyond the Guruswami-Sudan radius in polynomial time. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*, pages 285–294, 2005.
- [32] I. S. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics (SIAM)*, 8(2):300–304, 1960.
- [33] L. Sassaman, B. Cohen, and N. Mathewson. The Pynchon Gate: a Secure Method of Pseudonymous Mail Retrieval. In *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society (WPES '05)*, pages 1–9, 2005.
- [34] A. Shamir. How to share a secret. *Commun. ACM*, 22:612–613, November 1979.
- [35] V. Shoup. NTL, a library for doing number theory. <http://www.shoup.net/ntl/>, 2005. Accessed February 2012.
- [36] R. Sion and B. Carbunar. On the computational practicality of private information retrieval. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2007.
- [37] J. von zur Gathen. Hensel and Newton methods in valuation rings. *Math. Comp.*, 42(166):637–661, 1984.
- [38] S. Yekhanin. Towards 3-query locally decodable codes of subexponential length. *J. ACM*, 55(1):1–16, 2008.

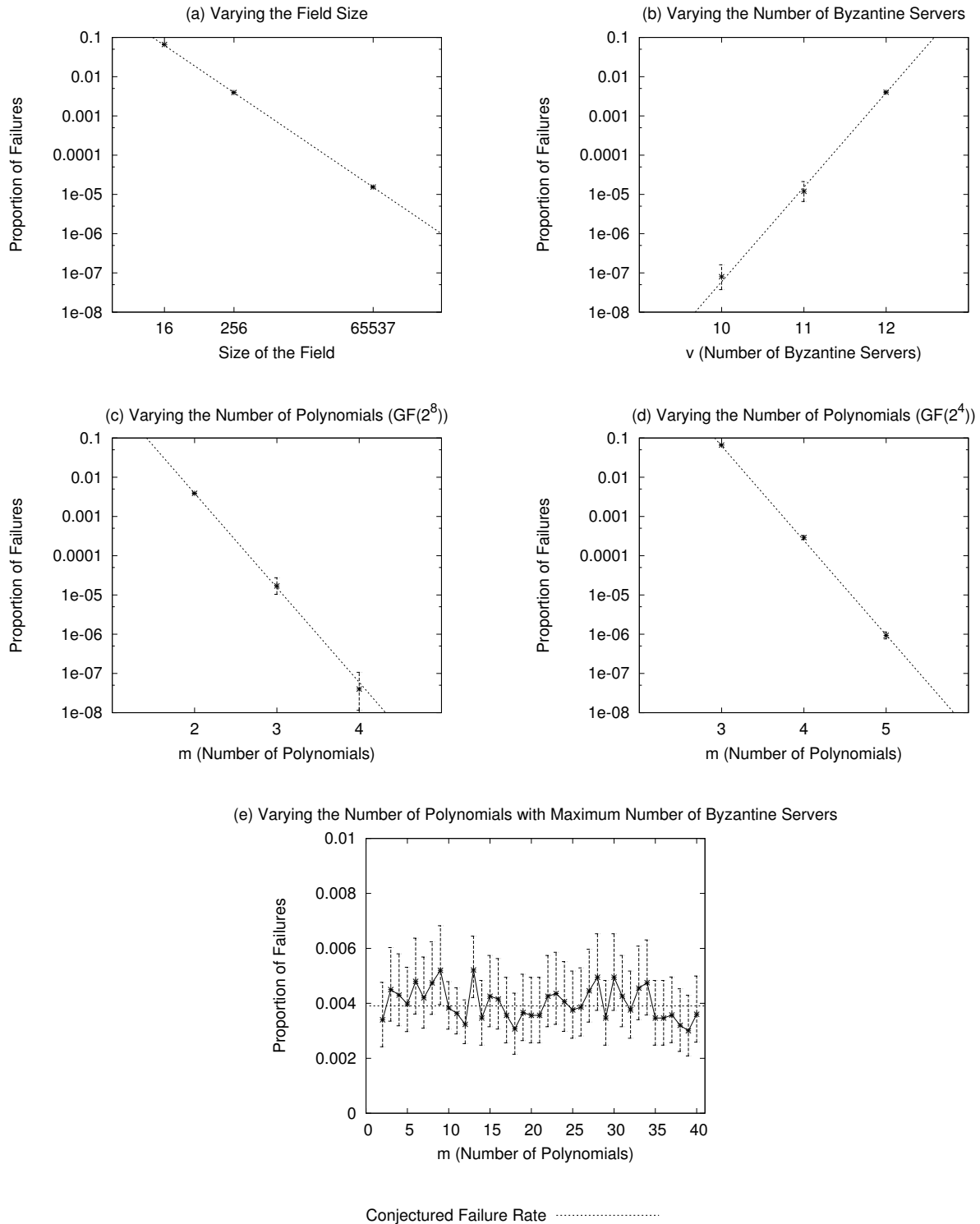


Figure 3: In Section 5.3, we conjectured that the linear multi-polynomial algorithm we present will fail with probability $\left(\frac{1}{|\mathbb{F}|}\right)^{m(h-t-1)-v+1}$. We ran several hundred million trials in order to test this conjecture. In plots (a)-(d), we varied a single parameter, keeping other parameters fixed, and plotted the observed proportion of failures for our linear multi-polynomial algorithm (black dots, with error bars at the 95% confidence interval) along with the conjectured value (dotted line). For plot (e) we varied m and used the maximum possible value of v for the given number of polynomials.

Appendix: Failure rate of Algorithm 1

The linear multi-polynomial algorithm described in Section 2.3.2 is probabilistic and may fail with some probability. We conjecture that the probability of failure is $\left(\frac{1}{|\mathbb{F}|}\right)^{m(h-t-1)-v+1}$ but do not have a proof. We ran hundreds of millions of tests, varying each of the parameters in the expression, in order to validate this conjecture experimentally. Figure 3 contains plots of the results; for details, see the extended version of this paper [13].

We observe that for large fields ($|\mathbb{F}| \geq 256$), our conjectured failure rate falls into the 95% confidence interval of our experimentally observed failure rate for all data points except 2 of the 39 data points in Figure 3(e). This is as expected from 95% confidence intervals. However, for a small field ($\mathbb{F} = GF(2^4)$) our conjecture appears to consistently underestimate the actual failure rate. This suggests the presence of an unknown second-order term in the failure rate; we will be exploring this in future work.

Billion-Gate Secure Computation with Malicious Adversaries

Benjamin Kreuter
brk7bx@virginia.edu
University of Virginia

abhi shelat
abhi@virginia.edu
University of Virginia

Chih-hao Shen
cs6zb@virginia.edu
University of Virginia

Abstract

The goal of this paper is to assess the feasibility of two-party secure computation in the presence of a malicious adversary. Prior work has shown the feasibility of billion-gate circuits in the semi-honest model, but only the 35k-gate AES circuit in the malicious model, in part because security in the malicious model is much harder to achieve. We show that by incorporating the best known techniques and parallelizing almost all steps of the resulting protocol, evaluating billion-gate circuits is feasible in the malicious model. Our results are in the standard model (i.e., no common reference strings or PKIs) and, in contrast to prior work, we do not use the random oracle model which has well-established theoretical shortcomings.

1 Introduction

Protocols for secure computation allow two or more mutually distrustful parties to collaborate and compute some function on each other's inputs, with privacy and correctness guarantees. Andrew Yao showed that secure two-party protocols can be constructed for any computable function [33]. Yao's protocol involves representing the function as a boolean circuit and having one party (called the *generator*) encrypt the circuit in such a way that it can be selectively decrypted by the other party (called the *evaluator*) to compute the output, a process called *garbling*. In particular, oblivious transfers are used for the evaluator to obtain a subset of the decryption keys that are needed to compute the output of the function.

Yao's protocol is of great practical significance. In many real-world situations, the inputs to a function may be too valuable or sensitive to share. Huang et al. explored the use of secure computation for biometric identification [14] in national security applications, in which it is desirable for individual genetic data to be kept private but still checked against a classified list. In a similar

security application, Osadchy et al. described how face recognition could be performed in a privacy-preserving manner [29]. The more general case of multiparty computation has already seen real-world use in computing market clearing prices in Denmark [2].

Yao's original protocol ensures the privacy of each party's input and the correctness of the output under the *semi-honest* model, in which both parties follow the protocol honestly. This model has been the basis for several scalable secure computation systems [4, 10, 12, 13, 17, 22, 26]. It is conceivable, however, that one of the parties may deviate from the protocol in an attempt to violate privacy or correctness. Bidders may attempt to manipulate the auction output in their favor; spies may attempt to obtain sensitive information; and a computer being used for secure computation may be infected with malware. Securing against *malicious* participants, who may deviate arbitrarily from pre-agreed instructions, in an efficient manner is of more practical importance.

There have been several attempts on practical systems with security against active, malicious adversaries. Lindell and Pinkas presented an approach based on garbled circuits that uses the cut-and-choose technique [23], with an implementation of this system having been given by Pinkas et al. [30]. Nielsen et al. presented the LEGO+ system [28], which uses efficient oblivious transfers and authenticated bits to enforce honest behaviors from participants. shelat and Shen proposed a hybrid approach that integrates sigma protocols into the cut-and-choose technique [32]. The protocol compiler presented by Ishai, Prabhakaran, and Sahai [16] also uses an approach based on oblivious transfer, and was implemented by Lindell, Oxman, and Pinkas [21]. In all these cases, AES was used as a benchmark for performance tests.

Protocols for general multiparty computation with security against a malicious *majority* have also been presented. Canetti et al. gave a construction of a *universally composable* protocol in the *common reference string* model [5]. The protocol compiler of Ishai et al.,

mentioned above, can be used to construct a multiparty protocol with security against a dishonest majority in the UC model [16]. Bendlin et al. showed a construction based on homomorphic encryption [1], which was improved upon by Damgård et al. [7]; these protocols were also proved secure in the UC model, and thus require additional setup assumptions. The protocol of Damgård et al. (dubbed “SPDZ” and pronounced “speedz”) is based on a preprocessing model, which improves the amortized performance. Damgård et al. presented an implementation of their protocol, which could evaluate the function $(x \times y) + z$ in about 3 seconds with a 128 bit security level, but with an amortized time of a few milliseconds.

This paper presents a scalable two-party secure computation system which guarantees privacy and correctness in the presence of a malicious party. The system we present can handle circuits with hundreds of millions or even billions of gates, while requiring relatively modest computing resources. Our system follows the Fairplay framework, allowing general purpose secure computation starting from a high level description of a function. We present a system with numerous technical advantages over the Fairplay system, both in our compiler and in the secure computation protocol. Unlike previous work, we do not rely solely on AES circuits as our benchmark; our goal is to evaluate circuits that are orders of magnitude larger than AES in the malicious model, and we use AES only as a comparison with other work. We prove the security of our protocol assuming *circular 2-correlation robust* hash functions and the hardness of the elliptic curve discrete logarithm problem, and require neither additional setup assumptions nor the random oracle model.

2 Contributions

Our principal contribution is to build a high performance secure two-party computation system that integrates state-of-the-art techniques for dealing with *malicious* adversaries efficiently. Although some of these techniques have been reported individually, we are not aware of any attempt to incorporate them all into one system, while ensuring that a security proof can still be written for that system. Even though some of the techniques are claimed to be compatible, it is not until everything is put together and someone has gone through all the details can a system as a whole be said to be provably secure.

System Framework We start by using Yao’s garbled circuit [33] protocol for securely computing functions in the presence of semi-honest adversaries, and shelat and Shen’s cut-and-choose-based transformation [32] that converts Yao’s garbled circuit protocol into one that

is secure against malicious adversaries.

We then modify the above to use Ishai et al.’s oblivious transfer extension [15] that has efficient amortized computation time for oblivious transfers secure against malicious adversaries, and Lindell and Pinkas’ random combination technique [23] that defends against selective failure attacks. We implement Kiraz’s randomized circuit technique [18] that guarantees that the generator gets either no output or an authentic output, i.e., the generator cannot be tricked into accepting arbitrary output.

Optimization Techniques For garbled circuit generation and evaluation, we incorporate Kolesnikov and Schneider’s free-XOR technique that minimizes the computation and communication cost for XOR gates in a circuit [20]. We also adopt Pinkas et al.’s garbled-row-reduction technique that reduces the communication cost for k -fan-in non-XOR gates by $1/2^k$ [30], which means at least a 25% communication saving in our system since we only have gates of 1-fan-in or 2-fan-in. Finally, we implement Goyal et al.’s technique for reducing communication as follows: during the cut-and-choose step, the check circuits are given to the evaluator by revealing the random seeds used to produce them rather than the check circuits themselves [11]. Combined with the 60%-40% check-evaluation ratio proposed by shelat and Shen [32], this technique provides a near 60% saving in communication. As far as we know, although these techniques exist individually, ours is the first system to incorporate all of these mutually compatible state-of-the-art techniques.

Circuit-Level Parallelism The most important new technique that we use is to exploit the embarrassingly parallel nature of shelat and Shen’s protocol for achieving security in the malicious model. Exploiting this, however, requires careful engineering in order to achieve good performance while maintaining security. We parallelize all computation-intensive operations such as oblivious transfers or circuit construction by splitting the generator-evaluator pair into hundreds of slave pairs. Each of the pairs works on an independently generated copy of the circuit in a parallel but synchronized manner as synchronization is required for shelat and Shen’s protocol [32] to be secure.

Computation Complexity For the computation time of a secure computation, there are two main contributing factors: the input processing time I (due to oblivious transfers) and the circuit processing time C (due to garbled circuit construction and evaluation). In the semi-honest model, the system’s computation time is simply $I+C$. Security in the malicious model, however, requires several extra checks. In the first instantiation of our sys-

tem, through heavy use of circuit-level parallelism, our system needs roughly $I + 2C$ to compute hundreds of copies of the circuit. Thus when the circuit size is sufficiently larger than the input size, our system (secure in the malicious model) needs roughly twice as much computation time as that needed by the original Yao protocol (secure in the semi-honest model). This is a tremendous improvement over prior work [30,32] which needed 100x more time than the semi-honest Yao. In the second instantiation of our scheme, we are able to achieve $I + C$ computation time, albeit at the cost of moderately more communication overhead.

Large Circuits In the Fairplay system, a garbled circuit is fully constructed before being sent over a network for the other party to evaluate. This approach is particularly problematic when hundreds of copies of a garbled circuit are needed against malicious adversaries. Huang et al. [13] pointed out that keeping the whole garbled circuit in memory is unnecessary, and that instead, the generation and evaluation of garbled gates could be conducted in a “pipelined” manner. Consequently, not only do both parties spend less time idling, only a small number of garbled gates need to reside in memory at one time, even when dealing with large circuits. However, this pipelining idea does not work trivially with other optimization techniques for the following two reasons:

- The cut-and-choose technique requires the generator to finish constructing circuits before the coin flipping (which is used to determine check circuits and evaluation circuits), but the evaluator cannot start checking or evaluating before the coin flipping. A naive approach would ask the evaluator to hold the circuits and wait for the results of the coin flipping before she proceeds to do her jobs. When the circuit is of large size, keeping hundreds of copies of such a circuit in memory is undesirable.
- Similarly, the random seed checking technique [11] requires the generator to send the hash for each garbled circuit, and later on send the random seeds for check circuits so that the communication for check circuits is vastly reduced. Note that the hash for an evaluation circuit is given away before the garbled circuit itself. However, a hash is calculated only after the whole circuit is generated. So the generation-evaluation pipelining cannot be applied directly.

Our system, however, integrates this pipelining idea with the optimization techniques mentioned above, and is capable of handling circuits of billions of gates.

AES-NI Besides the improvements by the algorithmic means, we also incorporate the Intel Advanced En-

ryption Standard Instructions (AES-NI) in our system. While the encryption is previously suggested to be

$$\text{Enc}_{X,Y}(Z) = H(X||Y) \oplus Z$$

in the literature [6, 20], where H is a 2-circular correlation robust function instantiated either with SHA-1 [13] or SHA-256 [30], we propose an alternative that

$$\text{Enc}_{X,Y}^k(Z) = \text{AES-256}_{X||Y}(k) \oplus Z,$$

where k is the index of the garbled gate. With the help of the latest instruction set, an AES-256 operation could take as little as 30% of the time for SHA-256. Since this operation is heavily used in circuit operations, with the help of AES-NI instructions, we are able to reduce the circuit computation time C by at least 20%.

Performance To get a sense of our improvements, we list the experimental results of the benchmark function—AES—from the most recent literature and our system. The latest reported system in the semi-honest model was built by Huang et al. [13] and needs 1.3 seconds (where $I = 1.1$ and $C = 0.2$) to complete a block of secure AES computation. The fastest known system in the malicious model was proposed by Nielson et al. [28] and has an amortized performance 1.6 seconds per block (or more precisely, $I = 79$ and $C = 6$ for 54 blocks). Our system provides security in the malicious model and needs 1.4 ($= I + 2C$, where $I = 1.0$ and $C = 0.2$) seconds per block. Note that both the prior systems require the full power of a random oracle, while ours requires a weaker cryptographic primitive, 2-circular correlation robust functions, which was recently shown to be sufficient to prove the security of the free-XOR technique. It should also be noted that our system benefits greatly from parallel computation, which was not tested for LEGO+.

Scalable Circuit Compiler One of the major bottlenecks that prevents large-scale secure computation is the need for a scalable compiler that generates a circuit description from a function written in a high-level programming language. Prior tools could barely handle circuits with 50,000 gates, requiring significant computational resources to compile such circuits. While this is just enough for an AES circuit, it is not enough for the large circuits that we evaluate in this paper.

We present a scalable boolean circuit compiler that can be used to generate circuits with billions of gates, with moderate hardware requirements. This compiler performs some simple but highly effective optimizations, and tends to favor XOR gates. The toolchain is flexible, allowing for different levels of optimizations and can be parameterized to use more memory or more CPU time when building circuits.

As a first sign that our compiler advances the state of the art, we observe that it automatically generates a smaller boolean circuit for the AES cipher than the hand-optimized circuit reported by Pinkas et al. [30]. AES plays an important role in secure computation, and oblivious AES evaluation can be used as a building block in cryptographic protocols. Not only is it one of the most popular building blocks in cryptography and real life security, it is often used as a benchmark in secure computation. With the textbook algorithm, the well-known Fairplay compiler can generate an AES circuit that has 15,316 non-XOR gates. Pinkas et al. were able to develop an optimized AES circuit that has 11,286 non-XOR gates. By applying an efficient S-box circuit [3] and using our compiler, we were able to construct an AES circuit that has 9,100 non-XOR gates. As a result, our AES circuit only needs 59% and 81% of the communication needed by the other two, respectively.

Most importantly, with our system and the scalable compiler, we are able to run experiments on circuits with sizes in the range of billions of gates. To the best of our knowledge, secure computation with such large circuits has never been run in the malicious model before. These circuits include 256-bit RSA (266,150,119 gates) and 4095x4095-bit edit distance (5,901,194,475). As the circuit size grows, resource management becomes crucial. A circuit of billions of gates can easily result in several GB of data stored in memory or sent over the network. Special care is required to handle these difficulties.

Paper Organization The organization of this paper is as follows. A variety of security decisions and optimization techniques will be covered in Section 3 and Section 4, respectively. Then, our system, including a compiler, will be introduced in Section 5. Finally, the experimental results are presented in Section 6 followed by the conclusion and future work in Section 7.

3 Techniques Regarding Security

The Yao protocol, while efficient, assumes honest behavior from both parties. To achieve security in the malicious model, it is necessary to enforce honest behavior. The *cut-and-choose* technique is one of the most efficient methods in the literature and is used in our system. Its main idea is for the generator to prepare multiple copies of the garbled circuit with independent randomness, and the evaluator picks a random fraction of the received circuits, whose randomness is then revealed. If any of the chosen circuits (called *check circuits*) is not consistent with the revealed randomness, the evaluator aborts; otherwise, she evaluates the remaining circuits (called *eval-*

uation circuits) and takes the majority of the outputs, one from each evaluation circuit, as the final output.

The intuition is that to pass the check, a malicious generator can only sneak in a few faulty circuits, and the influence of these (supposedly minority) faulty circuits will be eliminated by the majority operation at the end. On the other hand, if a malicious generator wants to manipulate the final output, she needs to construct faulty majority among evaluation circuits, but then the chance that none of the faulty circuits is checked will be negligible. So with the help of the cut-and-choose method, a malicious generator either constructs many faulty circuits and gets caught with high probability, or constructs merely a few and has no influence on the final output.

However, the cut-and-choose technique is not a cure-all. Several subtle attacks have been reported and would be a problem if not properly handled. These attacks include the *generator's input inconsistency attack*, the *selective failure attack*, and the *generator's output authenticity attack*, which are discussed in the following sections. Note that in this section, n denotes the input size and s denotes the number of copies of the circuit.

Generator's Input Consistency Recall that in the cut-and-choose step, multiple copies of a circuit are constructed and then evaluated. A malicious generator is therefore capable of providing altered inputs to different evaluation circuits. It has been shown that for some functions, there are simple ways for the generator to extract information about the evaluator's input [23]. For example, suppose both parties agree to compute the inner-product of their input, that is, $f([a_2, a_1, a_0], [b_2, b_1, b_0]) \mapsto a_2b_2 + a_1b_1 + a_0b_0$ where a_i and b_i is the generator's and evaluator's i -th input bit, respectively. Instead of providing $[a_2, a_1, a_0]$ to all evaluation circuits, the generator could send $[1, 0, 0]$, $[0, 1, 0]$, and $[0, 0, 1]$ to different copies of the evaluation circuits. After the majority operation from the cut-and-choose technique, the generator learns $\text{major}(b_2, b_1, b_0)$, the majority bit in the evaluator's input, which is not what the evaluator agreed to reveal in the first place.

There exist several approaches to deter this attack. Mohassel and Franklin [27] proposed the equality-checker that needs $O(ns^2)$ commitments to be computed and exchanged. Lindell and Pinkas [23] developed an approach that also requires $O(ns^2)$ commitments. Later, Lindell and Pinkas [24] proposed a pseudorandom synthesizer that relies on efficient zero-knowledge proofs for specific hardness assumptions and requires $O(ns)$ group operations. shelat and Shen [32] suggested the use of malleable claw-free collections, which also uses $O(ns)$ group operations, but they showed that witness-indistinguishability suffices, which is more efficient than zero-knowledge proofs by a constant factor.

In our system, we incorporate the malleable claw-free collection approach because of its efficiency. Although the commitment-based approaches can be implemented using lightweight primitives such as collision-resistant hash functions, they incur high communication overhead for the extra complexity factor s , that is, the number of copies of the circuit. On the other hand, the group-based approach could be more computationally intensive, but this discrepancy is compensated again due to the parameter s .¹ Hence, with similar computation cost, group-based approaches enjoy lower communication overhead.

Selective Failure A more subtle attack is *selective failure* [19, 27]. A malicious generator could use inconsistent keys to construct the garbled gate and OT so that the evaluator’s input can be inferred from whether or not the protocol completes. In particular, a cheating generator could assign (K_0, K_1) to an input wire in the garbled circuit while using (K_0, K_1^*) instead in the corresponding OT, where $K_1 \neq K_1^*$. As a result, if the evaluator’s input is 0, she learns K_0 from OT and completes the evaluation without complaints; otherwise, she learns K_1^* and gets stuck during the evaluation. If the protocol expects the evaluator to share the result with the generator at the end, the generator learns whether or not the evaluation failed, and therefore, the evaluator’s input is leaked.

Lindell and Pinkas [23] proposed the random input replacement approach that involves replacing each of the evaluator’s input bits with an XOR of s additional input bits, so that whether the evaluator aborts due to a selective failure attack is almost independent (up to a bias of 2^{1-s}) of her actual input value. Both Kiraz [18] and shelat and Shen [32] suggested a solution that exploits committing OTs so that the generator commits to her input for the OT, and the correctness of the OTs can later be checked by opening the commitments during the cut-and-choose. Lindell and Pinkas [24] also proposed a solution to this problem using cut-and-choose OT, which combines the OT and the cut-and-choose steps into one protocol to avoid this attack.

Our system is based on the random input replacement approach due to its scalability. It is a fact that the committing OT or the cut-and-choose OT does not alter the circuit while the random input replacement approach inflates the circuit by $O(sn)$ additional gates. However, it has been shown that $\max(4n, 8s)$ additional gates suffice [30]. Moreover, both the committing OT and the cut-

and-choose OT require $O(ns)$ group operations, while the random input replacement approach needs only $O(s)$ group operations. Furthermore, we observe that the random input replacement approach is in fact compatible with the OT extension technique. Therefore, we were able to build our system which has the group operation complexity independent of the evaluator’s input size, and as a result, our system is particularly attractive when handling a circuit with a large evaluator input.

Generator’s Output Authenticity It is not uncommon that *both* the generator and evaluator receive outputs from a secure computation, that is, the goal function is $f(x, y) = (f_1, f_2)$, where the generator with input x gets output f_1 , and the evaluator with input y gets f_2 .² In this case, the security requires that *both* the input and output are hidden from each other. In the semi-honest setting, the straightforward solution is to let the generator choose a random number c as an extra input, convert $f(x, y) = (f_1, f_2)$ into a new function $f^*((x, c), y) = (\lambda, (f_1 \oplus c, f_2))$, run the original Yao protocol for f^* , and instruct the evaluator to pass the encrypted output $f_1 \oplus c$ back to the generator, who can then retrieve her real output f_1 with the secret input c chosen in the first place. However, the situation gets complicated when either of the participants could potentially be malicious. In particular, a malicious evaluator might claim an arbitrary value to be the generator’s output coming from the circuit evaluation. Note that the two-output protocols we consider are not *fair* since the evaluator always learns her own output and may refuse to send the generator’s output. However, they can satisfy the notion that the evaluator cannot trick the generator into accepting arbitrary output.

Many approaches have been proposed to ensure the generator’s output authenticity. Lindell and Pinkas [23] proposed a solution similar to the aforementioned solution in the semi-honest setting, where the goal function is modified to compute $f_1 \oplus c$ and its MAC so that the generator can verify the authenticity of her output. This approach incurs a cost of adding $O(n^2)$ gates to the circuit. Kiraz [18] presented a two-party computation protocol in which a zero knowledge proof of size $O(s)$ is conducted at the end. shelat and Shen [32] suggested a signature-based solution which, similar to Kiraz’s, adds n gates to the circuit, and requires a proof of size $O(s + n)$ at the end. However, they observed that witness-indistinguishable proofs are sufficient.

Lindell and Pinkas’ approach, albeit straightforward, might introduce greater communication overhead than the description function itself. We therefore employ the approach that takes the advantages of the remaining two solutions. In particular, we implement Kiraz’s approach

¹To give concrete numbers, with an Intel Core i5 processor and 4GB DDR3 memory, a SHA-256 operation (from OpenSSL) requires 1,746 cycles, while a group operation (160-bit elliptic curve from the PBC library with preprocessing) needs 322,332 cycles. It is worth mentioning that s is at least 256 in order to achieve security level 2^{-80} . The gap between a symmetric operation and an asymmetric one becomes even smaller when modern libraries such as RELIC are used instead of PBC.

²Here f_1 and f_2 are short for $f_1(x, y)$ and $f_2(x, y)$ for simplicity.

(smaller proof size), but only a witness-indistinguishable proof is performed (weaker security property).

4 Techniques Regarding Performance

Yao’s garbled circuit technique has been studied for decades. It has drawn significant attention for its simplicity, constant round complexity, and computational efficiency (since circuit evaluation only requires fast symmetric operations). The fact that it incurs high communication overhead has provoked interest that has led to the development of fruitful results.

In this section, we will first briefly present the Yao garbled circuit, and then discuss the optimization techniques that greatly reduce the communication cost while maintaining the security. These techniques include free-XOR, garbled row reduction, random seed checking, and large circuit pre-processing. In addition to these original ideas, practical concerns involving large circuits and parallelization will be addressed.

4.1 Baseline Yao’s Garbled Circuit

Given a circuit that consists of 2-fan-in boolean gates, the generator constructs a garbled version as follows: for each wire w , the generator picks a random permutation bit $\pi_w \in \{0, 1\}$ and two random keys $w_0, w_1 \in \{0, 1\}^{k-1}$. Let $W_0 = w_0 || \pi_w$ and $W_1 = w_1 || (\pi_w \oplus 1)$, which are associated with bit value 0 and 1 of wire w , respectively. Next, for gate $g \in \{f | f : \{0, 1\} \times \{0, 1\} \mapsto \{0, 1\}\}$ that has input wire x with (X_0, X_1, π_x) , input wire y with (Y_0, Y_1, π_y) , and output wire z with (Z_0, Z_1, π_z) , the garbled truth table for g has four entries:

$$GTT_g \begin{cases} \text{Enc}(X_{0 \oplus \pi_x} || Y_{0 \oplus \pi_y}, Z_{g(0 \oplus \pi_x, 0 \oplus \pi_y)}) \\ \text{Enc}(X_{0 \oplus \pi_x} || Y_{1 \oplus \pi_y}, Z_{g(0 \oplus \pi_x, 1 \oplus \pi_y)}) \\ \text{Enc}(X_{1 \oplus \pi_x} || Y_{0 \oplus \pi_y}, Z_{g(1 \oplus \pi_x, 0 \oplus \pi_y)}) \\ \text{Enc}(X_{1 \oplus \pi_x} || Y_{1 \oplus \pi_y}, Z_{g(1 \oplus \pi_x, 1 \oplus \pi_y)}) \end{cases}$$

$\text{Enc}(K, m)$ denotes the encryption of message m under key K . Here the encryption key is a concatenation of two labels, and each label is a random key concatenated with its permutation bit. Intuitively, π_x and π_y permute the entries in GTT_g so that for $i_x, i_y \in \{0, 1\}$, the $(2i_x + i_y)$ -th entry represents the input pair $(i_x \oplus \pi_x, i_y \oplus \pi_y)$ for gate g , in which case the label associated with the output value $g(i_x \oplus \pi_x, i_y \oplus \pi_y)$ could be retrieved. More specifically, to evaluate the garbled gate GTT_g , suppose $X || b_x$ and $Y || b_y$ are the retrieved labels for input wire x and wire y , respectively, the evaluator will use $X || b_x || Y || b_y$ to decrypt the $(2b_x + b_y)$ -th entry in GTT_g and retrieve label $Z || b_z$, which is then used to evaluate the gates at the next level. The introduction of the permutation bit helps to identify the correct entry in GTT_g , and thus, only one, rather than all, of the four entries will be decrypted.

4.2 Free-XOR

Kolesnikov and Schneider [20] proposed the free-XOR technique that aims for removing the communication cost and decreasing the computation cost for XOR gates.

The idea is that the generator first randomly picks a global key R , where $R = r || 1$ and $r \in \{0, 1\}^{k-1}$. This global key has to be hidden from the evaluator. Then for each wire w , instead of picking both W_0 and W_1 at random, only one is randomly chosen from $\{0, 1\}^k$, and the other is determined by $W_b = W_{1 \oplus b} \oplus R$. Note that π_w remains the rightmost bit of W_0 . For an XOR gate having input wire x with $(X_0, X_0 \oplus R, \pi_x)$, input wire y with $(Y_0, Y_0 \oplus R, \pi_y)$, and output wire z , the generator lets $Z_0 = X_0 \oplus Y_0$ and $Z_1 = Z_0 \oplus R$. Observe that

$$\begin{aligned} X_0 \oplus Y_1 &= X_1 \oplus Y_0 = X_0 \oplus Y_0 \oplus R = Z_0 \oplus R = Z_1 \\ X_1 \oplus Y_1 &= X_0 \oplus R \oplus Y_0 \oplus R = X_0 \oplus Y_0 = Z_0. \end{aligned}$$

This means that while evaluating an XOR gate, XORing the labels for the two input wires will directly retrieve the label for the output wire. So no garbled truth table is needed, and the cost of evaluating an XOR gate is reduced from a decryption operation to a bitwise XOR.

This technique is only secure when the encryption scheme satisfies certain security properties. The solution provided by the authors is

$$\text{Enc}(X || Y, K) = H(X || Y) \oplus Z,$$

where $H : \{0, 1\}^{2k} \mapsto \{0, 1\}^k$ is a random oracle. Recently, Choi et al. [6] have further shown that it is sufficient to instantiate $H(\cdot)$ with a weaker cryptographic primitive, *2-circular correlation robust functions*. Our system instantiates this primitive with $H(X || Y) = \text{SHA-256}(X || Y)$. However, when AES-NI instructions are available, our system instantiates it with $H^k(X || Y) = \text{AES-256}(X || Y, k)$, where k is the gate index.

4.3 Garbled Row Reduction

The GRR (Garbled Row Reduction) technique suggested by Pinkas et al. [30] is used to reduce the communication overhead for non-XOR gates. In particular, it reduces the size of the garbled truth table for 2-fan-in gates by 25%.

Recall that in the baseline Yao’s garbled circuit, both the 0-key and 1-key for each wire are randomly chosen. After the free-XOR technique is integrated, the 0-key and 1-key for an XOR gate’s output wire depend on input key and R , but the 0-key for a non-XOR gate’s output wire is still free. The GRR technique is to make a smart choice for this degree of freedom, and thus, reduce one entry in the garbled truth table to be communicated over network.

In particular, the generator picks (Z_0, Z_1, π_z) by letting $Z_{g(0 \oplus \pi_x, 0 \oplus \pi_y)} = H(X_{0 \oplus \pi_x} || Y_{0 \oplus \pi_y})$, that is, either Z_0 or Z_1

is assigned to the encryption mask for the 0-th entry of the GTT_g , and the other one is computed by the equation $Z_b = Z_{1 \oplus b} \oplus R$. Therefore, when the evaluator gets $(X_{0 \oplus \pi_x}, Y_{0 \oplus \pi_y})$, both $X_{0 \oplus \pi_x}$ and $Y_{0 \oplus \pi_y}$ have rightmost bit 0, indicating that the 0-th entry needs to be decrypted. However, with GRR technique, she is able to retrieve $Z_{g(0 \oplus \pi_x, 0 \oplus \pi_y)}$ by running $H(\cdot)$ without inquiring GTT_g .

Pinkas et al. claimed that this technique is compatible with the free-XOR technique [30]. For rigorousness purposes, we carefully went through the details and came up with a security proof for our protocol that confirms this compatibility. The proof will be included in the full version of this paper.

4.4 Random Seed Checking

Recall that the cut-and-choose approach requires the generator to construct multiple copies of the garbled circuit, and more than half of these garbled circuits will be fully revealed, including the randomness used to construct the circuit. Goyal, Mohassel, and Smith [11] therefore pointed out an insight that the evaluator could examine the correctness of those check circuits by receiving a hash of the garbled circuit first, acquiring the random seed, and reconstructing the circuit and hash by herself.

This technique results in the communication overhead for check circuits independent of the circuit size. This technique has two phases that straddle the coin-flipping protocol. Before the coin flipping, the generator constructs multiple copies of the circuit as instructed by the cut-and-choose procedure. Then the generator sends to the evaluator the hash of each garbled circuit, rather than the circuit itself. After the coin flipping, when the evaluation circuits and the check circuits are determined, the generator sends to the evaluator the full description of the evaluation circuits and the random seed for the check circuits. The evaluator then computes the evaluation circuits and tests the check circuits by reconstructing the circuit and comparing its hash with the one received earlier. As a result, even for large circuits, the communication cost for each check circuit is simply a hash value plus the random seed. Our system provides that 60% of the garbled circuits are check circuits. Thus, this optimization significantly reduces communication overhead.

4.5 Working with Large Circuits

A circuit for a reasonably complicated function can easily consist of billions of gates. For example, a 4095-bit edit distance circuit has 5.9 billion gates. When circuits grow to such a size, the task of achieving high performance secure computation becomes challenging.

An $(I + 2C)$ -time solution Our solution for handling large circuits is based on Huang et al.'s work [13], which is the only prior work capable of handling large circuits (of up to 1.2 billion non-XOR gates) in the semi-honest setting. Intuitively, the generator could work with the evaluator in a pipeline manner so that small chunks of gates are being processed at a time. The generator could start to work on the next chunk while the evaluator is still processing the current one. However, this technique does not work directly with the random seed checking technique described above in Section 4.4 because the generator has to finish circuit construction and hash calculation before the coin flipping, but the evaluator could start the evaluation only after the coin flipping. As a result, the generator needs a way to construct the circuit first, wait for the coin flipping, and send the evaluation circuits to the evaluator without keeping them in memory the whole time. We therefore propose that the generator constructs the evaluation circuits all over again after the coin flipping, with the same random seed used before and the same keys for input wires gotten from OT.

We stress that when fully parallelized, the second construction of an evaluation circuit does not incur overhead to the overall execution time. Although we suggest to construct an evaluation circuit twice, the fact is that according to the random seed checking, a check circuit is already being constructed twice—once before the coin flipping by the generator for hash computation and once after by the evaluator for correctness verification. As a result, when each generator-evaluator pair is working on a single copy of the garbled circuit, the constructing time for a evaluation circuit totally overlaps with that for a check circuit. We therefore achieve the overall computation time $I + 2C$ mentioned earlier, where the first C is for the generator to calculate the circuit hash, and the other C is either for the evaluator to reconstruct a check circuit or for both parties to work on an evaluation circuit in a pipeline manner as suggested by Huang et al. [13].

Achieving an $(I + C)$ -time solution We observe that there is a way to achieve $I + C$ computation time, which exactly matches the running time of Yao in the semi-honest setting. This idea, however, is not compatible with the random-seed technique, and therefore represents a trade-off between communication and computation. Recall that the generator has to finish circuit construction and hash evaluation before beginning coin flipping, whereas the evaluator can start evaluating only after receiving the coin flipping results. The idea is to run the coin flipping in the way that only the evaluator gets the result and does not reveal it to the generator until the circuit construction is completed. Since the generator is oblivious to the coin flipping result, she sends every garbled circuit to the evaluator, who could then either

evaluate or check the received circuit. In order for the evaluator to get the generator’s input keys for evaluation circuits and the random seed for the check circuits, they run an OT, where the evaluator uses the coin flipping result as input and the generator provides either the random seed (for the check circuit) or his input keys (for the evaluation circuit). After the generator completes circuit construction and reveals the circuit hash, the evaluator compares the hash with her own calculation, if the hashes match, she proceeds with the rest of the original protocol. Note that this approach comes at the cost of sacrificing the random seed checking technique and its 60% savings in communication.

Working Set Optimization Another problem encountered while dealing with large circuits is the *working set minimization problem*. Note that the *circuit value problem* is log-space complete for P. It is suspected that $L \neq P$, that is, there exist some circuits that can be evaluated in polynomial time but require more than logarithmic space. This open problem captures the difficulty of handling large circuits during both the construction and evaluation, where at any moment there is a set of wires, called the *working set*, that are available and will be referenced in the future. For some circuits, the working set is inherently super-logarithmic. A naive approach is to keep the most recent D wires in the working set, where D is the upper bound of the input-output distance of all gates. However, there may be wires which are used as inputs to gates throughout the entire circuit, and so this technique could easily result in adding almost the whole circuit to the working set, which is especially problematic when there are hundreds of copies of a circuit of billions of gates. While reordering the circuit or adding identity gates to minimize D would mitigate this problem, doing so while maintaining the topological order of the circuit is known to be an NP-complete problem, the *graph bandwidth problem* [9].

Our solution to this difficulty is to pre-process the circuit so that each gate comes with a usage count. Our system has a compiler that converts a program in high-level language into a boolean circuit. Since the compiler is already using global optimization in order to reduce the circuit size, it is easy for the global optimizer to analyze the circuit and calculate the usage count for each gate. With this information, it is easy for the generator and evaluator to decrement the counter for each gate whenever it is being referenced and to toss away the gate whenever its counter becomes zero. In other words, we keep track of merely useful information and heuristically minimize the size of the working set, which is small compared with the original circuit size as shown in Table 1.

	AES	Dot ₄ ⁶⁴	RSA-32	EDT-255
circuit size	49,912	460,018	1,750,787	15,540,196
wrk set size	323	711	235	2,829

Table 1: The size of the working set for various circuits (sizes include input gates)

5 Boolean Circuit Compiler

Although the Fairplay circuit compiler can generate circuits, it requires a very large amount of computational resources to generate even relatively small circuits. Even on a machine with 48 gigabytes of RAM, Fairplay terminates with an out-of-memory error after spending 20 minutes attempting to compile an AES circuit. This makes Fairplay impractical for even relatively small circuits, and infeasible for some of the circuits tested in this project. One goal of this project was to have a general purpose system for secure computation, and so writing application specific programs to generate circuits, a technique used by others [13], was not an option.

To address this problem, we have implemented a new compiler that generates a more efficient output format than Fairplay, and which requires far lower computational resources to compile circuits. We were able to generate the AES circuit in only a few seconds on a typical desktop computer with only 8GB of RAM, and were able to generate and test much larger non-trivial circuits. We used the well-known *flex* and *bison* tools to generate our compiler, and implemented an optimizer as a separate tool. We also use the results from [30] to reduce 3 arity gates to 2 arity gates.

As a design decision, we created an imperative, untyped language with static scoping. We allow code, variables, and input/output statements to exist in the global scope; this allows very simple programs to be written without too much extra syntax. Functions may be declared, but may not be recursive. Variables do not need to be declared before being used in an unconditional assignment; variables assigned within a function’s body that are not declared in the global scope are considered to be local. Arrays are a language feature, but array indices must be constants or must be determined at compile time. If run-time determined indices are required for a function, a loop that selects the correct index may be used; this is necessary for oblivious evaluation. Variables may be arbitrarily concatenated, and bits or groups of bits may be selected from any variable and bits or ranges of bits may be assigned to; as with arrays, the index of a bit must be determined at compile time, or else a loop must be used. Note that loop variables may be used as such an index, since loops are always completely unrolled, and therefore the loop index can always be resolved at compile

time. Additional language features are planned as future work.

We use some techniques from the Fairplay compiler in our own compiler. In particular we use the single assignment algorithm from Fairplay, which is required to deal with assignments that occur inside of *if* statements. Otherwise, our compiler has several distinguishing characteristics that make it more resource efficient than Fairplay. The front end of our compiler attempts to generate circuits as quickly as possible, using as little memory as possible and performing only rudimentary optimizations before emitting its output. This can be done with very modest computational resources, and the intermediate output can easily be translated into a circuit for evaluation. The main optimizations are performed by the back end of the compiler, which identifies gates that can be removed without affecting the output of the circuit as a whole.

Unlike the Fairplay compiler, we avoided the use of hash tables in our compiler, using more memory-efficient storage. Our system can use one of three storage strategies: memory-mapped files, flat files without any mapping, and Berkeley DB. In our tests, we found that memory mapped files always resulted in the highest performance, but that Berkeley DB is only sometimes better than direct access without any mapping.

In the following sections, we describe these contributions in more detail, and provide experimental results.

5.1 Circuit Optimizations

The front-end of our compiler tends to generate inefficient circuits, with large numbers of unnecessary gates. As an example, for some operations the compiler generates large numbers of identity gates i.e. gates whose outputs follow one of their inputs. It is therefore essential to optimize the circuits emitted by the front end, particularly to meet our system's overall goal of practicality.

Our compiler uses several stages of optimization, most of which are global. As a first step, a local optimization removes redundant gates, i.e. gates that have the same truth table and input wires. This first step operates on a fixed-size chunk of the circuit, but we have found that there are diminishing improvements as the size of this window is increased. We also remove constant gates, identity gates, and inverters, which are generated by the compiler and which may be inadvertently generated during the optimization process. Finally, we remove gates that do not influence the output, which can be thought of as dead code elimination. The effectiveness of each optimization on different circuits is shown in Figure 1. The circuit that was least optimizable was the edit distance circuit, being reduced to only 82% of its size from the front end, whereas the RSA signing and the dot prod-

uct circuits were the most optimizable, being reduced to roughly half of the gates emitted by the front end.

Gate Removal The front-end of the compiler emits gates in topological order, and similar to Fairplay, our compiler assigns explicit identifiers to each emitted gate. To remove gates efficiently, we store a table that maps the identifiers of gates that were removed to the previously emitted gates, and for each gate that is scanned the inputs are rewritten according to this table. The table itself is then emitted, so that the identifiers of non-removed gates can be corrected. This mapping process can be done in linear time and space using an appropriate key-value store.

Removing Redundant Gates Some of the gates generated by the front end of our compiler have the same truth table and input wires as previously generated gates; such gates are redundant and can be removed. This removal process has the highest memory requirement of any other optimization step, since a description of every non-redundant gate must be stored. However, we found during our experiments that this optimization can be performed on discrete chunks of the circuit with results that are very close to performing the optimization on the full circuit, and that there are diminishing improvements in effectiveness as the size of the chunks is increased. Therefore, we perform this optimization using chunks, and can use hash tables to improve the speed of this step.

Removing Identity Gates and Inverters The front end may generate identity gates or inverters, which are not necessary. This may happen inadvertently, such as when a variable is incremented by a constant, or as part of the generation of a particular logic expression. While removing identity gates is straightforward, the removal of inverters requires more work, as gates which have inverted input wires must have their truth tables rewritten. There is a cascading effect in this process; the removal of some identity gates or inverters may transform later gates into identity gates or inverters. This step also removes gates with constant outputs, such as an XOR gate with two identical inputs. Constant propagation and folding occur as a side effect of this optimization.

Removing Unused Gates Finally, some gates in the circuit may not affect the output value at all. For this step, we scan the circuit backwards, and store a table of live gates; we then re-emit the live gates in the circuit and skip the dead gates. Immediately following this step, the circuit is prepared for the garbled circuit generator, which includes generating a usage count for each gate.

Effectiveness of Optimizations

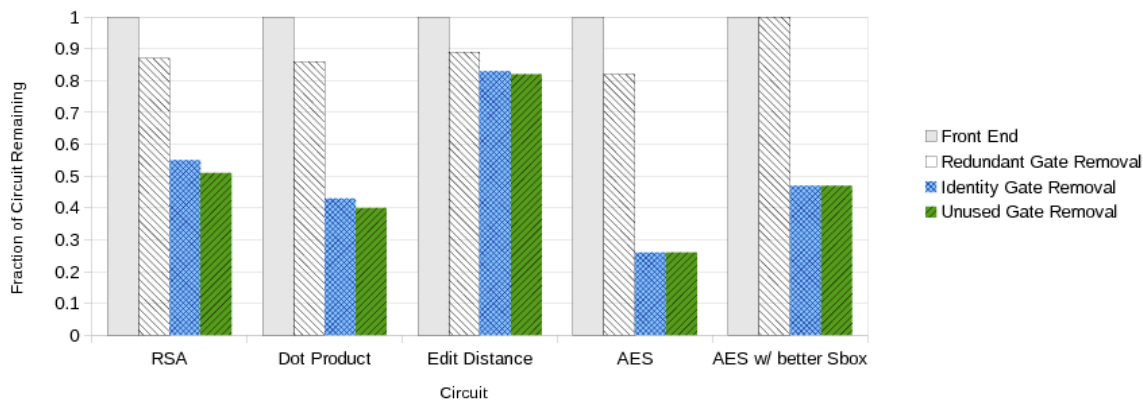


Figure 1: Average fraction of circuits remaining after each optimization is applied in sequence. We see that the *relative change* in circuit sizes after each optimization is dependent on the circuit itself, with some circuits being optimized more than others.

Circuit	DB (s)	mmap (s)	flat (s)
7200RPM Spinning Disk (ext4-fs)			
AES	4.3 ±0.5%	1.05 ± 1%	3.48 ±0.3%
RSA-32	103 ±0.3%	24.6 ±0.2%	78.4 ±0.3%
Dot ₄ ⁶⁴	32.56 ±0.1%	7.1 ±0.3%	28.37 ±0.1%
EDT-255	975 ±0.1%	240 ± 1%	700 ±0.9%
Solid-State Drive			
AES	3.62 ±0.3%	0.86 ± 1%	3.17 ±0.6%
RSA-32	96.5 ±0.2%	21.6 ±0.4%	68.3 ±0.3%
Dot ₄ ⁶⁴	30.5 ±0.5%	6.27 ± 1%	25.9 ±0.2%
EDT-255	907 ±0.1%	200 ±0.4%	590 ± 1%
Amazon EC2			
AES	5.56 ± 4%	1.12 ± 0%	7.11 ±0.3%
RSA-32	208 ±0.4%	45.7 ± 3%	240 ±0.1%
Dot ₄ ⁶⁴	46.3 ±0.1%	9.2 ±0.2%	60.7 ±0.2%
EDT-255	2500 ± 1%	405 ±0.2%	2050 ±0.2%
Circuit Sizes			
AES	RSA-32	Dot ₄ ⁶⁴	EDT-255
49,912	1,750,787	460,018	15,540,196

Table 2: Compile times for different storage systems for small circuits (sizes include input gates), using different storage media. Results are averaged over 30 experiments, with 95% confidence intervals. On EC2, a high-memory quadruple extra large instance was used.

Key-Value Stores Unfortunately, even though our compiler is more resource efficient than Fairplay, it still requires space that is linear in the size of the circuit. For very large circuits, circuits with billions of gates or more, this may exceed the amount of RAM that is available. Our compiler can make use of a computer’s hard drive to store intermediate representations of circuits and information about how to remove gates from the circuit. We used memory-mapped I/O to reduce the impact this has on performance; however, our use of *mmap* and *ftruncate* is not portable, and so our system also supports using an unmapped file or Berkeley DB. Our tests revealed that, as expected, memory-mapped I/O achieves the highest performance, but that Berkeley DB is sometimes better than unmapped files on high-latency filesystems. A summary of the performance of each method on a variety of storage systems is shown in Table 2.

Using the hard drive in this manner, we were able to compile our largest circuits. The performance impact of writing to disk should not be understated; a several-billion-gate edit distance 4095x4095 circuit required more than 3 days to compile on an Amazon EC2 high-memory image, with 68 GB of RAM, one third of which was spent waiting on I/O. Note, however, that this is a one-time cost; a compiled circuit can be used in unlimited evaluations of a secure computation protocol.

5.2 Compiler Testing Methodology

We tested the performance of our compiler using five circuits. The first was AES, to compare our compiler with the Fairplay system. We also used AES with the compact S-Box description given by Boyar and Parnia [3], which results in a smaller AES circuit. We used an RSA

RSA Size	Circuit Size	Compile Time (s)	Gates/s	Edit-Dist Size	Circuit Size	Compile Time (s)	Gates/s
16	208,499	2.6 ± 7%	80,000	31x31	144,277	1.70 ±0.7%	84,900
32	1,750,787	21.6 ±0.4%	81,100	63x63	717,233	8.56 ±0.7%	83,800
64	14,341,667	189 ±0.3%	75,900	127x127	3,389,812	41.7 ±0.5%	81,300
128	116,083,983	1810 ±0.3%	64,100	255x255	15,540,196	200 ±0.4%	77,700

Table 3: Time required to compile and optimize RSA and edit distance circuits on a workstation with an Intel Xeon 5506 CPU, 8GB of RAM and a 160GB SSD, using the textbook modular exponentiation algorithm. Note that the throughput for edit distance is higher even for comparably sized circuits; this is because the front end generates a more efficient circuit without any optimization. Compile times are averaged over 30 experiments, with 95% confidence intervals reported.

signing circuit with various toy key sizes, up to 128 bits, to test our compiler’s handling of large circuits; RSA circuits have cubic size complexity, allowing us to generate very large circuits with small inputs. We also used an edit distance circuit, which was the largest test case used by Huang et al. [13]; unlike the other test circuits, there is no multiplication routine in the inner loop of this function. Finally we used a dot product with error, a basic sampling function for the LWE problem, which is similar to RSA in creating large circuits, but also demonstrates our system’s ability to handle large input sizes.

After compiling these circuits, we tested the correctness by first performing a direct, offline evaluation of the circuit, and comparing the output to a non-circuit implementation. We then compared the output of an online evaluation to the offline evaluation. Additionally, for the AES circuit, we compared the output of the circuit generated by our compiler to the output of a circuit generated using Fairplay. We tested all three key-value stores on a variety of file systems, including a fast SSD, a spinning disk, and an Amazon EC2 instance store, checking for correctness as described above in each case.

5.3 Summary of Compiler Performance

Our compiler is able to emit and optimize large circuits in relatively short periods of time, less than an hour for circuits with tens of millions of gates on an inexpensive workstation. In Figure 1 we summarize the effectiveness of the various optimization stages on different circuits; in circuits that involve multiplication in finite fields or modulo an integer, the identity gate removal step is the most important, removing more than half of the gates emitted by the front-end. The edit distance circuit is the best-case for our front end, as less than 1/5 of the gates that are emitted can be removed by the optimizer. The throughput of our compiler is dependent on the circuit being compiled, with circuits which are more efficiently generated by the front-end being compiled faster; in Table 3 we compare the generation of RSA circuits to edit distance circuits.

6 Experimental Results

In this section, we give a detailed description of our system, upon which we have implemented various real world secure computation applications. The experimental environment is the Ranger cluster in the Texas Advanced Computing Center. Ranger is a blade-based system, where each node is a SunBlade x6240 blade running a Linux kernel and has four AMD Opteron quad-core 64-bit processors, as an SMP unit. Each node in the Ranger system has 2.3 GHz core frequency and 32 GB of memory, and the point-to-point bandwidth is 1 GB/sec. Although Ranger is a high-end machine, we use only a small fraction of its power for our system, only 512 out of 62,976 cores. Note that we use the PBC (Pairing-Based Cryptography) library [25] to implement the underlying cryptographic protocols such as oblivious transfers, witness-indistinguishable proofs, and so forth. However, moving to more modern libraries such as RELIC [31] is likely to give even better results, especially to those circuits with large input and output size.

System Setup In our system, both the generator and the evaluator run an equal number of processes, including a root process and many slave processes. A root process is responsible for coordinating its own slave processes and the other root process, while the slave processes work together on repeated and independent tasks. There are three pieces of code in our system: the generator, the evaluator, and the IP exchanger. Both the generator’s and evaluator’s program are implemented with Message Passing Interface (MPI) library. The reason for the IP exchanger is that it is common to run jobs on a cluster with dynamic working node assignment. However, when the nodes are dynamically assigned, the generator running on one cluster and the evaluator running on another might have a hard time locating each other. Therefore, a fixed location IP exchanger helps the match-up process as described in Figure 2. Our system provides two modes—the user mode and the simulation mode. The former works as mentioned above, and the latter simply

spawns an even number of processes, half for the generator and the other half for the evaluator. The network match-up process is omitted in the latter mode to simplify the testing of this system.

To achieve a security level of 2^{-80} , meaning that a malicious player cannot successfully cheat with probability better than 2^{-80} , requires at least 250 copies of the garbled circuit [32]. For simplicity, we used 256 copies in our experiments, that is, security parameters $k = 80$ and $s = 256$. Each experiment was run 30 times (unless stated otherwise), and in the following sections we report the average runtime of our experiments.

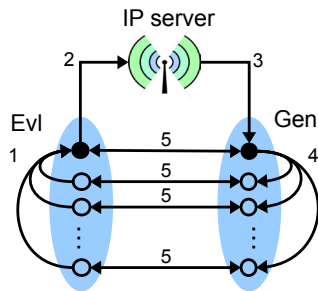


Figure 2: Both the generator and evaluator consist of a root process (solid dot) and a number of slave processes (hollow dots). The match-up works as follows: the slave evaluator processes send their IPs to the root evaluator process (Step 1), who then forwards them to the IP exchanger (Step 2). Next, the root generator process comes to acquire these IPs (Step 3) and dispatch them to its slaves (Step 4), who then proceed to pair up with one of the slave evaluator processes (Step 5) and start the main protocol. The arrows show the message flow.

Timing methodology When there is more than one process on each side, care must be taken in measuring the timings of the system. The timings reported in this section are the time required by the root process at each stage of the system. This was chosen because the root process will always be the *longest* running process, as it must wait for each slave process to run to completion. Moreover, in addition to doing all the work that the slaves do, the root processes also perform the input consistency check and the coin tossing protocol.

Impacts of the Performance Optimization Techniques

We have presented several performance optimization techniques in Section 4 with theoretical analyses, and here we demonstrate their empirical effectiveness in Table 4. As we have anticipated, the Random Seed Checking reduces the communication cost for the garbled circuits by 60%, and the Garbled Row Reduction further reduces by another 25%. In the RS and GRR columns,

the small deviation from the theoretical fraction 40% and 30%, respectively, is due to certain implementation needs. Our compiler is designed to reduce the number of non-XOR gates. In these four circuits, the ratio of non-XOR gates is less than 43%. So after further applying the Free-XOR technique, the final communication is less than 13% of that in the baseline approach.

	non-XOR (%)	Baseline (MB)	RS (%)	GRR (%)	FX (%)
AES	30.81	509	39.97	30.03	9.09
Dot ₄ ⁶⁴	29.55	4,707	39.86	29.91	8.88
RSA-32	34.44	17,928	39.84	29.88	10.29
EDT-255	41.36	159,129	39.84	29.87	12.36

Table 4: The impact of various optimization techniques: The Baseline shows the communication cost for 256 copies of the original Yao garbled circuit when $k = 80$; RS shows the remaining fraction after Random Seed technique is applied; GRR shows when Garbled Row Reduction is further applied; and FX shows when the previous two techniques and the Free-XOR are applied. (The communication costs here only include those in the generation and evaluation stages.)

Performance Gain by AES-NI On a machine with 2.53 GHz Intel Core i5 processor and 4GB 1067 MHz DDR3 memory, it takes 784 clock cycles to run a single SHA-256 (with OpenSSL 1.0.0g), while it needs only 225 cycles for AES-256 (with AES-NI). To measure the benefits of AES-NI, we use two instantiations to construct various circuits, listed in Table 5, and observe a consistent 20% saving in circuit construction.³

	size (gate)	AES-NI (sec)	SHA-256 (sec)	Ratio (%)
AES	49,912	0.12 ± 1%	0.15 ± 1%	78.04
Dot ₄ ⁶⁴	460,018	1.11 ± 0.4%	1.41 ± 0.5%	78.58
RSA-32	1,750,787	4.53 ± 0.5%	5.9 ± 0.8%	76.78
EDT-255	15,540,196	42.0 ± 0.5%	57.6 ± 1%	72.92

Table 5: Circuit generation time (for a single copy) with different instantiations (AES-NI vs SHA-256) of the 2-circular correlation robust function.

AES We used AES as a benchmark to compare our compiler to the Fairplay compiler, and as a test circuit

³The reason that saving 500+ cycles does not lead to more improvements is that this encryption operation is merely one of the contributing factors to generating a garbled gate. Other factors, for example, include GNU hash_map table insertion (~1,200 cycles) and erase (~600 cycles).

for our system. We tested the full AES circuit, as specified in FIPS-197 [8]. In the semi-honest model, it is possible to reduce the number of gates in an AES circuit by computing the key schedule offline; e.g. this is one of the optimizations employed by Huang et al. [13]. In the malicious model, however, such an optimization is not possible; the party holding the key could attempt to reduce the security level of the cipher by computing a malicious key schedule. So in our experiments we compute the entire function, including the key schedule, online.

In this experiment, two parties collaboratively compute the function $f : (x, y) \mapsto (\perp, \text{AES}_x(y))$, i.e., the circuit generator holds the encryption key x , while the evaluator has the message y to be encrypted. At the end, the generator will not receive any output, whereas the evaluator will receive the ciphertext $\text{AES}_x(y)$.

Type	Fairplay	Ours-A	Pinkas et al.	Ours-B
non-XOR	15,316	15,300	11,286	9,100
XOR	35,084	34,228	22,594	21,628

Table 6: The components of the AES circuits from different sources. Ours-A comes from the textbook AES algorithm, and Ours-B uses an optimized S-box circuit from [3]. (Sizes do not include input or output wires)

First of all, we demonstrate the performance of our compiler in Table 6. We have shown in Section 5 that our compiler is capable of large circuit generation. We also found in our experiments that our compiler produces smaller AES circuit than Fairplay. Given the same high-level description of AES encryption (textbook AES), our compiler produces a circuit with a smaller gate count and even fewer non-XOR gates. When applying the compact S-Box description proposed by Boyar and Parelta [3] to the high-level description as input to our compiler, a smaller AES circuit than the hand-optimized one from Pinkas et al. is generated with less effort.

In Table 7, both the computational and communication costs for each main stage are listed under the traditional setting, where there is only one process on each side. These main stages include oblivious transfer, garbled circuit construction, the generator’s input consistency check, and the circuit evaluation. Each row includes both the computation and communication time used. Note that network conditions could vary from setting to setting. Our experiments run in a local area network, and the data can only give a rough idea on how fast the system could be in an ideal environment. However, the precise amount of data being exchanged is reported.

We notice in Table 7 that the evaluator spends an unreasonable amount of time on communication with respect to the amount of data to be transmitted in both the oblivious transfer and circuit construction stages.

		Gen (sec)		Eval (sec)	Comm (KB)
OT	comp	45.8±0.09%		34.0±0.2%	5,516
	comm	0.1±	1%	11.9±0.6%	
Gen.	comp	35.6± 0.5%		–	3
	comm	–		35.6±0.5%	
Inp. Chk	comp	–		1.75±0.2%	266
	comm	–		–	
Evl.	comp	14.9±	0.6%	32.4±0.4%	28,781
	comm	18.2±	1%	3.2±0.8%	
Total	comp	96.3± 0.3%		68.0±0.2%	34,566
	comm	18.3±	1%	50.8±0.4%	

Table 7: The 95% two-sided confidence intervals of the computation and communication time for each stage in the experiment $(x, y) \mapsto (\perp, \text{AES}_x(y))$.

This is because the evaluator spends that time waiting for the generator to finish computation-intensive tasks. The same reasoning explains why in the circuit evaluation stage the generator spends more time in communication than the evaluator. This waiting results from the fact that both parties need to run the protocol in a synchronized manner. A generator-evaluator pair cannot start next communication round while any other pair has not finished the current one. This synchronization is crucial since our protocol’s security is guaranteed only when each communication round is performed sequentially. While the parallelization of the program introduces high performance execution, it does not and should not change this essential property. A stronger notion of security such as universal security will be required if asynchronous communication is allowed. By using TCP sockets in “blocking” mode, we enforce this communication round synchronization.

Note that the low communication during the circuit construction stage is due to the random seed checking technique. Also, the fact that the generator spends more time in the evaluation stage than she traditionally does comes from the second construction for evaluation circuits. Recall that only the evaluation circuits need to be sent to the evaluator. Since only 40% of the garbled circuits (102 out of 256) are evaluation-circuits, the ratio of the generator’s computation time in the generation and evaluation stage is $35.63:14:92 \simeq 5:2$.

We were unfortunately unable to find a cluster of hundreds of nodes that all support AES-NI. Our experimental results, therefore, do not show the full potential of all the optimization techniques we have proposed. However, recall that for certain circuits the running time in the semi-honest setting is roughly half of that in the

node #	4		16		64		256	
	Gen	Evl	Gen	Evl	Gen	Evl	Gen	Evl
OT	12.56±0.1%	8.41±0.1%	4.06±0.1%	2.13±0.2%	1.96±0.1%	0.58±0.2%	0.64±0.1%	0.19±0.2%
Gen.	8.18±0.4%	–	1.92±0.7%	–	0.49±0.4%	–	0.14± 1%	–
Inp. Chk	–	0.42± 4%	–	0.10± 10%	–	–	–	–
Evl.	3.3± 4%	7.08± 1%	0.80± 10%	1.58± 4%	0.23± 17%	0.37± 7%	0.12±0.5%	0.05±0.6%
Inter-com	4± 5%	13.2±0.3%	0.93± 10%	4.08±0.8%	0.31± 20%	1.98± 1%	0.11± 40%	0.72±0.2%
Intra-com	0.17± 30%	0.23± 20%	0.18± 8%	0.25± 6%	0.45± 20%	0.48± 15%	0.34± 30%	0.34± 30%
Total time	28.3±0.3%	29.4±0.3%	7.90±0.5%	8.17±0.4%	3.45± 2%	3.44± 2%	1.4± 10%	1.3± 9%

Table 8: The average and error interval of the times (seconds) running AES circuit. The number of nodes represents the degree of parallelism on each side. “–” means that the time is smaller than 0.05 seconds. Inter-com refers to the communication between the two parties, and intra-com refers to communication between nodes for a single party.

malicious setting. We estimate a 20% improvement in the performance of garbled circuit generation when the AES-NI instruction set becomes ubiquitous, based on the preliminary results presented above in Table 5.

Table 8 shows that the Yao protocol really benefits from the circuit-level parallelization. Starting from Table 7, where each side only has one process, all the way to when each side has 256 processes, as the degree of parallelism is multiplied by four, the total time reduces into a quarter. Note that the communication costs between the generator and evaluator remain the same, as shown in Table 7. It may seem odd that the communication costs are *reduced* as the number of processes increase. The real interpretation of this data is that as the number of processes increases, the “waiting time” decreases.

Notice that as the number of processes increases, the ratio of the time the generator spends in the construction and evaluation stage decreases from 5:2 to 1:1. The reason is that the number of garbled circuit each process handles is getting smaller and smaller. Eventually, we reach the limit of the benefits that the circuit-level parallelism could possibly bring. In this case, each process is dealing with merely a single copy of the garbled circuit, and the time spent in both the generation and evaluation stages is the time to construct a garbled circuit.

To the best of our knowledge, completing an execution of secure AES in the malicious model within 1.4 seconds is the best result that has ever been reported. The next best result from Nielsen et al. [28] is 1.6 seconds, and it is an amortized result (85 seconds for 54 blocks of AES encryption in parallel) in the random oracle model. This is only a crude comparison, however; our experimental setup uses a cluster computer while Nielsen et al. used only two desktops. A better comparison would be possible given a parallel implementation of Nielsen et al.’s system, and we are interested in seeing how much of an improvement such an implementation could achieve.

Large Circuits In this experiment, we run the 4095-bit edit distance circuit, that is, $(x, y) \mapsto (\perp, \text{EDT}(x, y))$, where $x, y \in \{0, 1\}^{4095}$. In particular, we use the $I + C$ approach, where the computation time could be roughly a half of that of the $I + 2C$ approach with the price of not getting to use the random-seed technique. Recall that in the $I + C$ approach, the generator and the evaluator conduct the cut-and-choose in a way that the generator does not know the check circuits until she finishes transferring all the garbled circuits. Next, both the parties run the circuit generation and evaluation in a pipeline manner, where one party is generating and giving away garbled gates on one end, and the other party is evaluating and checking the received gates at the other end at the same time. The results are shown in Table 9.

	Gen (sec)	Eval (sec)	Comm (Byte)
OT	19.73±0.5% 1.1± 6%	5.26±0.4% 15.6±0.6%	1.7×10^8
Cut-& Choose	1.1±0.8% –	– 1.5± 2%	6.5×10^7
Gen./Evl.	24,400± 1% 4,900± 1%	14,600± 3% 14,700± 2%	1.8×10^{13}
Inp. Chk	0.6± 20% 0.4± 40%	– 0.60± 20%	8.5×10^6
Total	24,400± 1% 4,900± 1%	14,600± 3% 14,700± 2%	1.8×10^{13}

Table 9: The result of $(x, y) \mapsto (\perp, \text{EDT-4095}(x, y))$. Each party is comprised of 256 cores in a cluster. This table comes from 6 invocations of the system. Similarly, the upper row in each stage is the computation time, while the lower is the communication time.

This circuit generated by our compiler has 5.9 billion gates, and 2.4 billion of those are non-XOR. It is worth

mentioning that, without the random-seed technique, the communication cost shown in Table 9 can also be estimated by $256 \times 2.4 \times 10^9 \times 3 \times 10 = 1.8 \times 10^{13}$, since 256 copies of the garbled circuits need to be transferred, each copy has 2.4 billion non-free gates, each non-free gate has three entries, and each entry has $k = 80$ bits.

In addition to showing that our system is capable of handling the largest circuits ever reported, we also have shown a speed in the malicious setting that is comparable to those in the semi-honest setting. In particular, we were able to complete a single execution of 4095-bit edit distance circuit in less than 8.2 hours with a rate of 82,000 (non-XOR) gates per second. Note that Huang et al.'s system is the only one, to the best of our knowledge, that is capable of handling such large circuits [13]; they reported a rate of over 96,000 (non-XOR) gates per second for an edit-distance circuit in the semi-honest setting.

7 Conclusion

We have presented a general purpose secure two party computation system which offers security against malicious adversaries and which can efficiently evaluate circuits with hundreds of millions and even billions of gates on affordable hardware. Our compiler can generate large circuits using fewer computational resources than similar compilers, and offers improved flexibility to users of the system. Our evaluator can take advantage of parallel computing resources, which are becoming increasingly common and affordable. As future work, we plan further improvements to our compiler and language, as well as experiments on systems other than Ranger.

The source code for this system can be downloaded from the authors' website (<http://crypto.cs.virginia.edu/>), along with example functions, including those describe in this paper.

8 Acknowledgements

We would like to thank Benny Pinkas, Thomas Schneider, Nigel Smart and Stephen Williams for providing us with a copy of their optimized AES circuit. We would also like to thank Gabriel Robins for his advice on minimizing circuits in VLSI systems. We are particularly grateful to Ian Goldberg for his very helpful comments.

This work is supported by Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under contract FA8750-11-2-0211. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US government.

References

- [1] BENDLIN, R., DAMGÅRD, I., ORLANDI, C., AND ZAKARIAS, S. Semi-homomorphic encryption and multiparty computation. In *Proceedings of the 30th Annual international conference on Theory and applications of cryptographic techniques: advances in cryptology* (Berlin, Heidelberg, 2011), EUROCRYPT'11, Springer-Verlag, pp. 169–188.
- [2] BOGETOFT, P., CHRISTENSEN, D. L., DAMGÅRD, I., GEISLER, M., JAKOBSEN, T. P., KRØIGAARD, M., NIELSEN, J. D., NIELSEN, J. B., NIELSEN, K., PAGTER, J., SCHWARTZBACH, M. I., AND TOFT, T. Secure Multiparty Computation Goes Live. In *Financial Cryptography* (2009), pp. 325–343.
- [3] BOYAR, J., AND PERALTA, R. A new combinational logic minimization technique with applications to cryptology. In *Proceedings of the 9th international conference on Experimental Algorithms* (Berlin, Heidelberg, 2010), SEA'10, Springer-Verlag, pp. 178–189.
- [4] BRICKELL, J., AND SHMATIKOV, V. Privacy-preserving graph algorithms in the semi-honest model. In *Proceedings of the 11th international conference on Theory and Application of Cryptology and Information Security* (Berlin, Heidelberg, 2005), ASIACRYPT'05, Springer-Verlag, pp. 236–252.
- [5] CANETTI, R., LINDELL, Y., OSTROVSKY, R., AND SAHAI, A. Universally composable two-party and multi-party secure computation. In *Proceedings of the thirty-fourth annual ACM symposium on Theory of computing* (New York, NY, USA, 2002), STOC '02, ACM, pp. 494–503.
- [6] CHOI, S. G., KATZ, J., KUMARESAN, R., AND ZHOU, H.-S. On the security of the "free-xor" technique. In *Proceedings of the 9th international conference on Theory of Cryptography* (Berlin, Heidelberg, 2012), TCC'12, Springer-Verlag, pp. 39–53.
- [7] DAMGARD, I., PASTRO, V., SMART, N., AND ZAKARIAS, S. Multiparty Computation from Somewhat Homomorphic Encryption. In *Proceedings of the 32th Annual International Cryptology Conference on Advances in Cryptology* (2012), CRYPTO '12. <http://eprint.iacr.org/2011/535>.
- [8] FIPS. *Advanced Encryption Standard (AES)*, 2001.
- [9] GAREY, M., GRAHAM, R., JOHNSON, D., AND KNUTH, D. Complexity results for bandwidth minimization. *SIAM Journal on Applied Mathematics* 34, 3 (1978), 477–495.
- [10] GENTRY, C., HALEVI, S., AND SMART, N. P. Homomorphic Evaluation of the AES Circuit. In *Proceedings of the 32th Annual International Cryptology Conference on Advances in Cryptology* (2012), CRYPTO '12. <http://eprint.iacr.org/2012/099>.
- [11] GOYAL, V., MOHASSEL, P., AND SMITH, A. Efficient two party and multi party computation against covert adversaries. In *Proceedings of the theory and applications of cryptographic techniques 27th annual international conference on Advances in cryptology* (Berlin, Heidelberg, 2008), EUROCRYPT'08, Springer-Verlag, pp. 289–306.
- [12] HENECKA, W., K ÖGL, S., SADEGHI, A.-R., SCHNEIDER, T., AND WEHRENBURG, I. Tasty: tool for automating secure two-party computations. In *Proceedings of the 17th ACM conference on Computer and communications security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 451–462.
- [13] HUANG, Y., EVANS, D., KATZ, J., AND MALKA, L. Faster secure two-party computation using garbled circuits. In *Proceedings of the 20th USENIX conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 35–35.
- [14] HUANG, Y., MALKA, L., EVANS, D., AND KATZ, J. Efficient Privacy-Preserving Biometric Identification. In *NDSS'11* (2011).

- [15] ISHAI, Y., KILIAN, J., NISSIM, K., AND PETRANK, E. Extending Oblivious Transfers Efficiently. In *CRYPTO'03*, vol. 2729 of *LNCs*. Springer Berlin / Heidelberg, 2003, pp. 145–161.
- [16] ISHAI, Y., PRABHAKARAN, M., AND SAHAI, A. Founding cryptography on oblivious transfer — efficiently. In *Proceedings of the 28th Annual conference on Cryptology: Advances in Cryptology* (Berlin, Heidelberg, 2008), *CRYPTO 2008*, Springer-Verlag, pp. 572–591.
- [17] JHA, S., KRUGER, L., AND SHMATIKOV, V. Towards practical privacy for genomic computation. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2008), *SP '08*, IEEE Computer Society, pp. 216–230.
- [18] KIRAZ, M. *Secure and Fair Two-Party Computation*. PhD thesis, Technische Universiteit Eindhoven, 2008.
- [19] KIRAZ, M., AND SCHOENMAKERS, B. A Protocol Issue for The Malicious Case of Yao's Garbled Circuit Construction. In *27th Symposium on Information Theory in the Benelux* (2006).
- [20] KOLESNIKOV, V., AND SCHNEIDER, T. Improved garbled circuit: Free xor gates and applications. In *Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part II* (Berlin, Heidelberg, 2008), *ICALP '08*, Springer-Verlag, pp. 486–498.
- [21] LINDELL, Y., OXMAN, E., AND PINKAS, B. The IPS Compiler: Optimizations, Variants and Concrete Efficiency. In *CRYPTO'11* (2011), pp. 259–276.
- [22] LINDELL, Y., AND PINKAS, B. Privacy preserving data mining. In *Proceedings of the 20th Annual International Cryptology Conference on Advances in Cryptology* (London, UK, UK, 2000), *CRYPTO '00*, Springer-Verlag, pp. 36–54.
- [23] LINDELL, Y., AND PINKAS, B. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Proceedings of the 26th annual international conference on Advances in Cryptology* (Berlin, Heidelberg, 2007), *EUROCRYPT '07*, Springer-Verlag, pp. 52–78.
- [24] LINDELL, Y., AND PINKAS, B. Secure two-party computation via cut-and-choose oblivious transfer. In *Proceedings of the 8th conference on Theory of cryptography* (Berlin, Heidelberg, 2011), *TCC'11*, Springer-Verlag, pp. 329–346.
- [25] LYNN, B. Pairing-Based Cryptography Library, 2006. <http://crypto.stanford.edu/abc/>.
- [26] MALKA, L. Vmccrypt: modular software architecture for scalable secure computation. In *Proceedings of the 18th ACM conference on Computer and communications security* (New York, NY, USA, 2011), *CCS '11*, ACM, pp. 715–724.
- [27] MOHASSEL, P., AND FRANKLIN, M. Efficiency tradeoffs for malicious two-party computation. In *Proceedings of the 9th international conference on Theory and Practice of Public-Key Cryptography* (Berlin, Heidelberg, 2006), *PKC'06*, Springer-Verlag, pp. 458–473.
- [28] NIELSEN, J. B., NORDHOLT, P. S., ORLANDI, C., AND BURRA, S. S. A New Approach to Practical Active-Secure Two-Party Computation. In *Proceedings of the 32th Annual International Cryptology Conference on Advances in Cryptology* (2012), *CRYPTO '12*. <http://eprint.iacr.org/2011/091>.
- [29] OSADCHY, M., PINKAS, B., JARROUS, A., AND MOSKOVICH, B. Scifi - a system for secure face identification. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2010), *SP '10*, IEEE Computer Society, pp. 239–254.
- [30] PINKAS, B., SCHNEIDER, T., SMART, N. P., AND WILLIAMS, S. C. Secure two-party computation is practical. In *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology* (Berlin, Heidelberg, 2009), *ASIACRYPT '09*, Springer-Verlag, pp. 250–267.
- [31] RELIC. <http://code.google.com/p/relic-toolkit/>.
- [32] SHELAT, A., AND SHEN, C.-H. Two-output secure computation with malicious adversaries. In *Proceedings of the 30th Annual international conference on Theory and applications of cryptographic techniques: advances in cryptology* (Berlin, Heidelberg, 2011), *EUROCRYPT'11*, Springer-Verlag, pp. 386–405.
- [33] YAO, A. C. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science* (Washington, DC, USA, 1982), *SFCS '82*, IEEE Computer Society, pp. 160–164.

Progressive authentication: deciding when to authenticate on mobile phones

Oriana Riva[†] Chuan Qin^{§*} Karin Strauss[†] Dimitrios Lymberopoulos[†]
[†]Microsoft Research, Redmond [§]University of South Carolina

Abstract

Mobile users are often faced with a trade-off between security and convenience. Either users do not use any security lock and risk compromising their data, or they use security locks but then have to inconveniently authenticate every time they use the device. Rather than exploring a new authentication scheme, we address the problem of deciding *when* to surface authentication and for *which applications*. We believe reducing the number of times a user is requested to authenticate lowers the barrier of entry for users who currently do not use any security. Progressive authentication, the approach we propose, combines multiple signals (biometric, continuity, possession) to determine a level of confidence in a user's authenticity. Based on this confidence level and the degree of protection the user has configured for his applications, the system determines whether access to them requires authentication. We built a prototype running on modern phones to demonstrate progressive authentication and used it in a lab study with nine users. Compared to the state-of-the-art, the system is able to reduce the number of required authentications by 42% and still provide acceptable security guarantees, thus representing an attractive solution for users who do not use any security mechanism on their devices.

1 Introduction

Security on mobile phones is often perceived as a barrier to usability. Users weaken the security of their phones for the convenience of interacting with their applications without having to type a password every time. As a result, according to a recent study [25], more than 30% of mobile phone users do not use a PIN on their phones. On the other hand, the amount of high-value content stored on phones is rapidly increasing, with mobile payment

and money transfer applications as well as enterprise data becoming available on mobile devices [22].

One approach for increasing security of mobile phones is to replace PINs and passwords with a more suitable authentication scheme. Yet, alternative schemes have their own flaws. Token-based approaches [6, 31, 35], for instance, are in general harder to attack than passwords, but they go against the desire of users to carry fewer devices. Biometric identification has gained interest in the mobile community [11], but not without performance, acceptability, and cost issues [27].

In this work, we look at the problem of mobile authentication from a different angle. Rather than exploring a new authentication scheme, we study the problem of *when* to surface authentication and for *which applications*. Unlike desktops and laptops, which users tend to use for a long, continuous period of time, users access mobile phones periodically or in response to a particular event (*e.g.*, incoming email notification). This lack of continuous interaction creates the need to authenticate with the device almost every time users wish to use it. Even though the interaction between users and mobile devices might not be continuous, the physical contact between the user and the mobile device can be. For instance, if a user places his phone in his pocket after a phone call, even though the user stops interacting with it, the authenticated owner is still “in contact” with the device. When the user pulls the phone out of his pocket, authentication should not be necessary. On the other hand, if the phone lost contact with the authenticated user (*e.g.*, left on a table), then authentication should be required. As a result, if the phone is able to accurately infer the physical interaction between the authenticated user and the device (*e.g.*, through its embedded and surrounding sensors), it can extend the validity of a user authentication event, reducing the frequency of such events.

This approach not only significantly lowers the authentication overhead on the user, but also makes the authentication effort proportional to the value of the content

*Work done while interning at Microsoft Research.

being accessed. If the system has strong confidence in the user's authenticity, the user will be able to access any content without explicitly authenticating. If the system has low confidence in his authenticity, he will only be able to access low-value content (*e.g.*, a weather app) and will be required to explicitly authenticate to view high-value content (*e.g.*, email, banking). By doing so, the system provides low overhead with security guarantees that can be acceptable for a large user population, particularly users who do not use any security lock on their phones – our primary target.

We propose mobile systems that *progressively authenticate* (and de-authenticate) users by constantly collecting cues about the user, such that at any point in time the user is authenticated with a certain level of confidence. Several authentication signals are used. The system can exploit the increasing availability of sensors on mobile devices to establish users' identity (*e.g.*, through voice recognition). Once the user is authenticated, the system can exploit continuity to extend the validity of a successful authentication by leveraging accelerometers and touch sensors to determine when the phone "has left" the user after a successful login. Finally, as users' personal devices are increasingly networked, it can use proximity and motion sensors to detect whether the phone is next to another device of the same user where he is currently authenticated and active.

Automatically inferring a user's authenticity from continuous sensor streams is challenging because of inherent noise in sensing data as well as energy and performance constraints. If sensor streams are naively used, the system could suffer a high number of false rejections (not recognizing the owner) and/or unauthorized accesses (recognizing others as the owner). Our solution consists of using machine learning techniques to robustly combine and cross-check weak sensor signals. Sensors and related processing create energy and performance issues. First, processing the sensor data and running the inference stack on a mobile device can be challenging. We show how to architect the system to mitigate these problems by offloading processing to the cloud or disabling computation-expensive signals. Second, continuously recording data from the sensors can be a major power hog in a mobile device. We rely on existing architectures for low power continuous sensing such as LittleRock [28, 29]. We expect in the near future all mobile devices to be equipped with such low power sensing subsystems, as it is already made possible by the latest generation of mobile processors with embedded ARM Cortex M4 based sensing subsystems [37].

In summary, the contributions of this work are three-fold. First, we introduce the progressive authentication model, which relies on the continuous combination of multiple authentication signals collected through widely

available sensor information. This approach makes the authentication overhead lower and proportional to the value of the content being accessed. Second, we present a Windows Phone implementation of progressive authentication, which explores the implications of applying the model to a concrete phone platform. Third, we show through experiments and a deployment of the system with 9 users how progressive authentication (within our study setup) achieves the goal of reducing the number of explicit authentications (by 42%) with acceptable security guarantees (no unauthorized accesses and only 8% of cases in which the user's authenticity is estimated higher than it should be), power consumption (325 mW) and delay (987 msec). We believe this represents an attractive solution for users who currently do not use any security lock on their phones.

The rest of the paper is organized as follows. The next section motivates our work through a user study on access control on mobile phones and sets the goals of our work. Section 3 presents the key principles of progressive authentication, and Section 4 and Section 5 demonstrate these principles through the prototype system we designed and implemented. We then evaluate our system: in Section 6 we describe how we collected sensor traces and trained the inference model, and in Section 7 we present the experimental results. Finally, we discuss related work and conclude.

2 Motivations and assumptions

Most smart phones and tablets support some locking mechanism (*e.g.*, PIN) that, if used, prevents access to all the devices applications, with the exception of a few pre-defined functions such as answering incoming calls, making emergency calls, and taking photos. Unlike desktops and laptops, users interact with their phones for short-term tasks or in response to specific events (*e.g.*, incoming SMS notification). This results in users having to authenticate almost every time they wish to use their phone.

2.1 User study

We investigated how well this model meets the access control needs of mobile users through a user study [13]. We recruited 20 participants (9M/11F, age range: 23-54) who owned both smart phones and tablets, using Microsoft's recruiting service to reach a diverse population in our region. They were primarily iPhone (9) or Android (8) phone users. 11 used a PIN on their phone and 9 did not use any. Although this study is much broader, some of its findings motivate progressive authentication.

Beyond all-or-nothing: Today's mobile devices force users to either authenticate before accessing any applica-

tion or to entirely disable device locking. According to our study, this all-or-nothing approach poorly fits users' needs. Across our participants, those who used security locks on their devices wanted about half of their applications to be unlocked and always accessible. Not coincidentally, participants who did *not* use security locks wanted to have about half of their applications locked.

Multi-level security: Even when offered the option to categorize their applications into two security levels (*i.e.*, one unprotected level and one protected by their preferred security scheme), roughly half of our participants expressed the need for at least a third category, mainly motivated by security and convenience trade-off reasons. Most participants chose to have a final configuration with one unlocked level, one weakly-protected level for private, but easily accessible applications, and one strongly-protected level for confidential content such as banking.

Convenience: Participants were asked to rank the security of different authentication approaches (PINs, passwords, security questions, and biometric authentication). More than half of the participants ranked passwords and PINs as the most secure authentication schemes, but when asked to indicate their favorite authentication mechanism, 85% chose biometric systems. Convenience was clearly the dominant factor in their decision.

These lessons highlight the value of correctly trading off security and convenience on a per-application basis. This is the cornerstone of progressive authentication.

2.2 Assumptions and threat model

Progressive authentication builds on two main assumptions. First, we assume a *single-user model*, which is in line with current smart phones. However, such as in related systems [20, 23], adding multiple profiles (*e.g.*, family members, guests) to progressive authentication would be straightforward. Second, we assume the availability of *low-cost sensors* in the device and the environment for detecting a user's presence and identity. Indeed, the sensors used in our implementation are widely available in modern devices and office environments. As more sensors become pervasive (*e.g.*, 3D depth cameras), they can be easily folded into the system.

The main security goal of progressive authentication is to protect important applications from unauthorized use, while providing a probabilistic deterrent to use of less sensitive applications by others. In a noise-free environment where sensor signals are reliable and error-free, the same security guarantees can be provided regardless of the physical environment and the people present. In reality, these guarantees depend on several external conditions (presence of people with similar aspect, background noise, light conditions, etc.), such that the sys-

tem relies on the probabilistic guarantees of a machine learning model trained to account for these variables.

Progressive authentication seeks to preserve its goal in the presence of adversaries under the following conditions. Our model inserts barriers against unauthorized use when the phone is left unattended. Attacks can be carried out by both strangers and "known non-owner" attackers such as children and relatives. Attackers can operate both in public (*e.g.*, a conference room, a restaurant) and private spaces (*e.g.*, an office, at home). Attackers may *not know* which signals progressive authentication utilizes (*e.g.*, not knowing that voice is a factor and speaking into the system) or they may *know* which signals are in use and therefore try to avoid them (*e.g.*, remaining silent). Attacks in which the adversary obtains privileged information (*e.g.*, passwords or biometrics) are orthogonal to our guarantees, *i.e.*, progressive authentication does not make these authentication mechanisms more secure, so they are not considered in our security analysis. Finally, we assume the phone operating system and the devices the user owns are trusted, and that attacks cannot be launched on wireless links used by the system.

In the presence of the legitimate user, progressive authentication allows him to implicitly authorize other users (by simply handing them the phone), as we assume the above attacks can be prevented by the owner himself. For instance, if the user authenticates with the phone by entering a PIN and, soon after that, hands it to another person, the phone will remain authenticated.

Stronger attacker models are possible, but addressing them would require more sophisticated sensors and inference algorithms, thus increasing the burden on the user (cost of sensors, phone form factor, training effort) or limiting the applicability of our approach in environments not equipped with these sensors.

2.3 Goals

Based on the assumptions and threat model described above, our system has the following goals:

Finer control over convenience versus security trade-offs: Today's models are all-or-nothing: they require users to make a single security choice for all applications. Users who do not use authentication typically make this choice for convenience. Our goal is to give these users more flexibility such that they can keep the convenience of not having to authenticate for low-value content, but to improve security for high-value content. Progressive authentication enables per-application choices.

Moderating the surfacing of authentication: Too many authentication requests can be annoying to users. In addition to per-application control, our goal is to reduce the number of times users are inconvenienced by having

to authenticate. We do this by augmenting devices with inference models that use sensor streams to probabilistically determine if the user should continue to be authenticated, avoiding the surfacing of authentication requests.

Low training overhead for users: Our focus is on convenience, so the above inference models should be easy to train and deploy. Some scenarios, such as potential attacks, are difficult to train after the phone ships, so this type of training needs to be done in advance, in a user-agnostic manner. Personalized models (*e.g.*, voice models), however, cannot be trained in advance because they depend on input from the user. In such cases, users should not be burdened with complex training routines. Our solution partitions the model into a high-level, user-agnostic model that can be trained before shipping, and low-level, personalized models that can be trained via normal user interaction with the device (*e.g.*, by recording the user's voice during phone calls).

Overhead management: Mobile phones are afflicted with battery limitations, yet their computing capabilities are limited. It should be possible to let the system transition between high-performance and energy-saving modes. We achieve this goal by offloading computation to the cloud, or moving all the computation locally and disabling certain sensor streams, at the cost of accuracy.

3 Progressive authentication model

Progressive authentication establishes the authenticity of the user by combining multiple authentication signals (*multi-modal*) and leveraging *multi-device* authentication. The goal is to keep the user authenticated while in possession of the device (*i.e., continuity* since a last successful authentication is detected) or de-authenticate the user once the user lets go of it (*i.e.*, a discontinuity is detected). The confidence level in the user's authenticity is then compared to one authentication threshold for a single-level approach, or to multiple authentication thresholds for *multi-level* authentication.

Multi-modal. Possible signal types used for multi-modal authentication are the following:

Biometric signals: user appearance and sound (face and voice recognition). Fingerprinting and other hard biometric identification can also be used, but we focus on low-sensitivity signals. High-sensitivity signals may result in privacy concerns (*e.g.*, if stored on untrusted servers).

Behavioral signals: deviation of a user's current behavior from his past recurrent behavior may result in lower confidence. For example, if the phone is used at an unusual time and in a location that the user has never visited, then authentication may be surfaced. Location signals are a subset of behavioral signals.

Possession signals: nearby objects that belong to the user, such as a laptop or a PC, may increase the confidence level of the phone. This may be detected using Bluetooth signal strength or RFIDs, for example.

Secrets: PINs and passwords are still compatible with progressive authentication and actually represent some of the strongest signals it uses. They are requested from the user when the system is unable to determine the user's authenticity with high confidence.

In combining these signals several challenges must be considered. First, most signals are produced using unreliable and discrete sensor measurements. Second, certain signals may require combining readings from sources with different sampling frequencies and communication delays. As a result, most signals are not individually sufficient to determine user authenticity and when combined they may be inconsistent as well. Finally, signals vary in strength. Some signals can provide a stronger indication of authenticity than others because, for example, some may be easier to fake and some are more discriminating than others (*e.g.*, a PIN vs. the user's voice which may be recorded in a phone call). For all these reasons, signals need to be combined and cross-checked. However, drawing the correlations across these signals manually is a cumbersome job, prone to errors and inconsistencies. As we discuss in the next section, we use machine learning tools to derive a robust inference model from the signals we collect.

Continuous. Continuity is one of the cornerstones of progressive authentication. It comes from the observation that users are likely to use their phones shortly after a previous use. For example, after the user reads emails, he locks the phone to save energy. He keeps holding the phone and talking to someone else. When he tries to use the phone five minutes later, the phone is locked even if he did not let go of it. If the user has been touching the phone since the last successful authentication, the authentication level should be maintained unless "negative" signals are being received (*e.g.*, mismatching biometric signals). By "touching", we currently mean actively holding or storing the phone in a pocket. A phone's placement with respect to the user can be determined by accelerometers, touch screens, light, temperature and humidity sensors, most of which are embedded in modern phones.

Multi-device. Progressive authentication takes advantage of widespread device connectivity to gather information about the user from other devices he owns. If a user is logged in and active in another nearby device, this information represents a strong signal of the user's presence. For example, if a user enters his office, places the phone on his desk, logs into his PC and wants to use the phone, the phone already has some cues about his

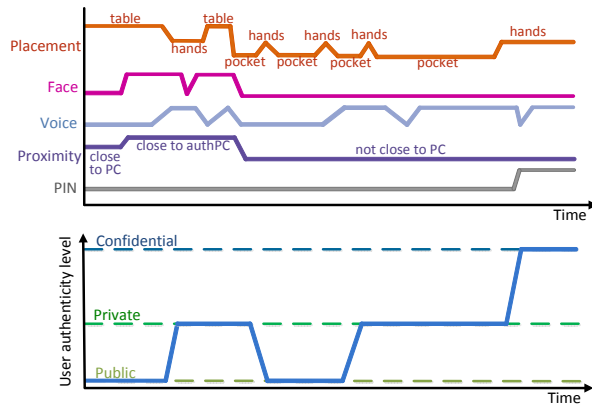


Figure 1: Progressive authentication in action. Signals are combined to infer the level of confidence in user authenticity. Based on this level, access (without PIN) to public, private or confidential content is granted.

authenticity. Devices need to be registered once (*e.g.*, at purchase time) so the overhead for the user is limited.

Multi-level. Users storing high-value content on their devices want to protect this harder than less valuable information. Progressive authentication associates user authenticity with a confidence level. This enables the system to depart from the all-or-nothing paradigm and allows the user to associate different protection levels to different data and applications. For example, a weather or a game application does not reveal any sensitive data, so it can be made accessible to anybody. An email or a social networking application contains sensitive information, but perhaps not as sensitive as a financial application. Thus, the confidence level required for accessing email is lower than that for accessing the banking application. In a complete system, users may configure their applications into an appropriate security level, perhaps at installation time, may specify levels by application categories (*e.g.*, games, email, banking, etc.), or may even accept a default configuration set by their corporate’s IT department. We expect users to deal with no more than three or four levels. The preferred configuration that emerged from our user study was three levels. Our system prototype adopts the same.

Putting it all together. Figure 1 illustrates the progressive authentication process. Multiple signals are aggregated into a single framework, which determines the level of confidence in user authenticity. The example considers continuity (phone placement), biometric (face and voice) and multi-device (proximity to a PC where the user is logged on/off) signals as well PIN events. Based on the confidence level, the user is allowed to access predefined data and applications in three sensitivity levels: public, which requires a very low confidence; private, which requires medium confidence; and

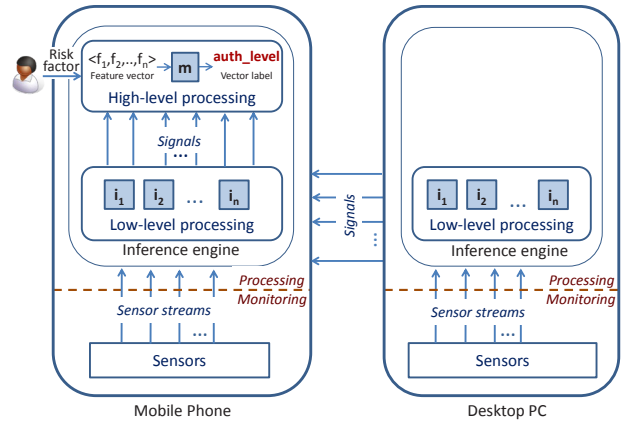


Figure 2: Progressive authentication architecture. The two-level inference engine processes incoming sensor data to estimate the level of a user’s authenticity.

confidential, which requires very high confidence. Note that when the user’s authenticity level is too low for accessing high security applications, the system requires a PIN, which raises the score to the confidential level. Previously achieved levels are maintained by continuity as long as signals are sufficient.

4 Architecture

This section describes the architecture of progressive authentication. Note that the system is designed for phones and mobile devices with rich sensing capabilities, but could also be applied to other types of computers. The specific system we consider is a 2-device configuration including a mobile phone and another user-owned device such as a desktop PC. Progressive authentication is used to control authentication on the phone and the PC simply acts as another sensor.

As Figure 2 shows, the software running on each device consists of two levels. At the monitoring level, sensor streams are continuously collected and fed as inputs to the inference engine, the core of our system. The inference engine processes sensor information in two stages. A collection of inference modules (i_i boxes inside low-level processing) processes raw sensor data to extract the set of primitive signals described in the previous section, such as placement of the mobile device as an indication of continuity, presence of the user through biometric inference, and proximity to other known devices as an indication of device possession and safe location. A confidence level is associated to each signal. These inferred signals are not sufficiently robust in isolation to authenticate or de-authenticate the user, hence the next step (high-level processing) combines them to compute the overall confidence in the user’s authenticity.

We rely on well-established machine learning techniques, particularly support vector machine (SVM) models, to do this properly. In using such models, *feature extraction* is the most critical step. Features should provide as accurate and discriminative information as possible about the user's authenticity and whether the phone has left or not the user since the last successful authentication (Table 1 in the next section lists all features we have considered in our prototype). Once the high-level processing extracts these features from the most recent signals produced by low-level processing, they are combined in a *feature vector*. This vector is passed as input to the machine learning model (*m* box inside high-level processing), which in turn associates a *label* to it, indicating the estimated confidence in the user's authenticity at that particular time. This label is then mapped to one of the system protection levels, which we currently limit to three. Named after the most restrictive type of application to which they allow access, they are: *public level*, in which only applications classified as public are accessible; *private level*, in which public and private applications are accessible; and *confidential level*, in which all applications are available.

The machine learning model is trained offline using a set of *labeled feature vectors* (Section 6 describes how they are obtained). The model is trained to minimize a *loss function*, which represents the relative seriousness of the kinds of mistakes the system can make. More specifically, it represents the loss incurred when the system decides that a user has authenticity level higher than it should be and possibly automatically authenticates him (*false authentication*), or when the system decides that he has authenticity level lower than it should be and requires explicit authentication (*false rejection*). The system exposes a so-called *risk factor* to users such that they can themselves adjust how aggressively they want to trade security for convenience. The higher the risk factor, the higher the convenience, but also the higher the security risk.

Unlike low-level processing, which is continually active as sensor data are fed into the system, high-level processing is activated only when the phone detects touch-screen activity (*i.e.*, the user is about to use an application). This model is sufficient to provide the illusion of continuous authentication without unnecessarily draining the battery of the phone and wasting communication resources. The high level keeps being invoked periodically, as long as the phone's touch screen detects activity, allowing the system to promptly adjust its authentication decision to sudden changes in ambient conditions, which may indicate threat situations.

A main challenge for this system is resource management. Sensor monitoring and processing need to minimally impact the phone's operation and battery lifetime.

For this reason, we designed the architecture in a modular manner. Depending on its resource constraints, the phone can decide to offload some processing to the cloud or another device so as to reduce the computing overhead, or it can instead decide to do more processing locally in order to reduce communication. In addition, all low-level processing modules are independent and the high-level processing can work with all or some of them, *i.e.*, some modules can be disabled to save resources, in which case the corresponding features are computed based on the latest available signals. In the evaluation section, we show the impact of these resource-saving decisions on the overall system's accuracy. The system currently supports a few static configurations. Systems such as MAUI [8] and AlfredO [10] may be used to support dynamic module reconfiguration.

For this kind of authentication to be viable, it is important to keep the machine learning models as user-agnostic as possible. The main inference model can be trained independently using a broad population (before the phone ships). Some of the low-level inference algorithms, specifically face and voice recognition, are user specific. However, their training does not incur a high overhead on the user because it can be done in the background, during user interaction with the device. A user's appearance can be recorded while the user types or reads from the PC. 1 minute of image recording is sufficient for the face recognition model to work accurately. Voice can be recorded during phone calls. In particular, Speaker Sense [21], the algorithm we used in our implementation, was designed to specifically support such type of training. The voice model was trained using 2 minutes of audio recording.

5 Implementation

We built a progressive authentication prototype on Windows Phone 7.1 OS and used it in a 2-device configuration with a desktop PC running Windows. The bottom part of Figure 3 shows the detailed architecture of the system running on the phone and on the desktop PC.

Sensors. On the phone, we use accelerometers, light, temperature/humidity sensor, touch screen, login events, microphone, and Bluetooth receiver. All these sensors are widely available on commercial phones (*e.g.*, Android) today. As the platform we chose does not yet support light and temperature/humidity sensors, we augmented it by using the .NET Gadgeteer [9] sensor kit, a platform for quick prototyping of small electronic gadgets and embedded hardware devices. Figure 5 shows the final prototype. On the PC, we use external webcams, sensors for activity detection (mouse motion, key presses, login/logout events), and Bluetooth adapters. Bluetooth is used for proximity detection between phone

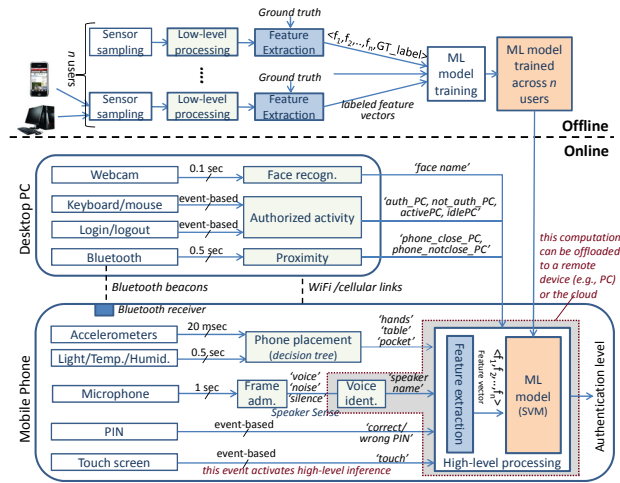


Figure 3: Detailed architecture of a 2-device progressive authentication implementation. Offline, a machine learning model is trained using traces from several users. Online, the model is loaded into the phone and invoked using the features extracted from the most-recent sensor signals. Signals' confidence values are not shown.

and PC. The system employs Bluetooth, WiFi or cellular links for communication. The channel can be encrypted. **Low-level processing.** It currently produces these signals: proximity to known devices (detected using Bluetooth beacons), presence of activity on phone or PC (detected using touch screen and activity detection sensors, respectively), speaker identification, face recognition, and phone placement.

Voice recognition relies on Speaker Sense [21], which is based on a Gaussian Mixture Model (GMM) classifier [30]. To make the recognition process more efficient, in a first stage, called Frame Admission, sound recorded by the phone's microphone is processed to identify voiced speech frames and discard silence frames or unvoiced speech frames. Voiced speech frames are admitted to the second stage, called Voice Identification, where speaker recognition occurs and produces an identifier with associated confidence.

Face recognition is based on a proprietary algorithm. Pictures are collected using a PC's webcam and fed into the face recognition model. This returns an identifier of the recognized person along with a confidence.

High-level processing. The primitive authentication signals and associated confidence levels generated by the low-level processing are combined at this stage to extract the features used by the machine learning model. Figure 4 gives a detailed example of the entire process, where accelerometer data are aggregated every 200 msec and processed by the placement decision tree to extract placement signals. Continuity features are extracted and

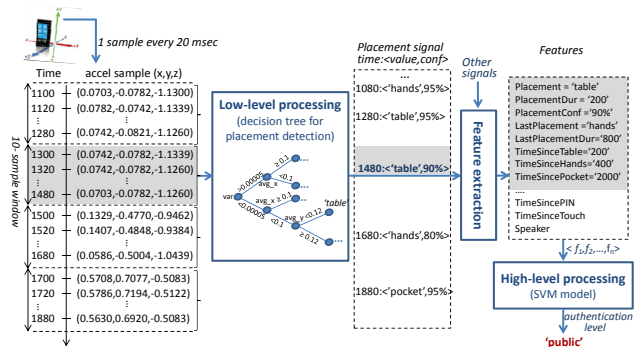


Figure 4: Processing chain from accelerometer data (sampled every 20 msec) to placement signals and to continuity features which are fed as input to the machine learning model to estimate the user's authentication level.

combined in a feature vector with all other features (biometric, possession, continuity and secret-derived features). Feature vectors are fed as input to the machine learning (ML) model and a label indicating the authenticity of the user is produced.

Table 1 lists all features currently used. These features cover all categories of signals described in Section 3 with the exception of behavioral signals. For these, we plan to first collect longer-term traces of users interacting with our prototype, and then extract behavioral patterns for each user. The current implementation relies on SVM models, but we experimented also with Decision Tree and Linear Regression models (see Section 7).

Finally, the high-level processing and some of the low-level processing can be too computationally intensive for mobile phones, so we support a *minimum-power configuration* where Voice Identification and high-level processing (gray box in Figure 3) are offloaded to the cloud (Windows Azure) or a remote device.

User interface. Figure 5 shows our current user interface. We do not claim this to be the most appropriate interface. For instance, users may want to configure the level of details returned in the feedback. However, we used it during our user study [13] and it was well received by our participants. When the phone is picked up, the user can see which applications are accessible and which are locked (not shown in the figure). When trying to access a locked application, the user is prompted to enter a PIN and receives a visual explanation of why access is denied. Icons representing the authentication signals are lit up (the corresponding signal is present) or grayed out (the corresponding signal is absent). This allows users to understand why a PIN is required.

Table 1: Machine learning features used in the high-level processing.

Category	Features	Description
Cont.	Placement, PlacementDuration, PlacementConf	Current placement of the phone, how long it has lasted, and associated confidence
Cont.	LastPlacement, LastPlacementDuration	Last placement of the phone, and how long it lasted
Cont.	TimeSinceTable, TimeSinceHands, TimeSincePocket	Time elapsed since the last time the phone was on the table, in the user's hands, or pocket
Cont./Secrets	TimeSincePIN, TimeSinceTouch	Time since last login event and time since the phone's screen was last touched
Biom.	Speaker, SpeakerConf	Whether a human voice was identified (owner, other, no-voice) and associated confidence
Biom.	TimeSinceOwnerVoice, TimeSinceNonOwnerVoice	Time since (any) voice was identified
Biom.	TimeSinceSound	Time since any sound (either voice or noise) was detected
Poss./Biom.	ProxAuthDev, ProxAuthDevConf	Proximity of phone to a device where the user is logged in and active, and confidence
Poss./Biom.	TimeSinceProx	Time elapsed since the proximity status last changed



Figure 5: System prototype running on a WP Samsung Focus augmented with Gadgeteer sensors (white box).

6 Data collection and offline training

We collected traces from 9 users (4F, 5M) to evaluate our system prototype. The participants were researchers (4), engineers (3) and admins (2) working in our group. During the lab study, they used the system for a little longer than an hour. The study consisted of two parts. The first part focused on collecting data for evaluating the accuracy of the low-level processing models. It lasted for roughly 30 minutes and it was guided by a researcher. The data collected was used to train voice identification and face recognition models, which were plugged into the system for use in the second part.

The second part of the study was conducted in two user study rooms that had a participant and an observer side, which allowed us to collect information about the activities in which the participant was engaged without disturbing. This information was used to generate the *ground truth* necessary for training the high-level processing model (more details below). Each study session was also video-recorded such that it could be used later for the same purpose. Participants were asked to perform a series of tasks by following a script. The script consisted of ordinary tasks in an office setting, involving

a phone and a desktop PC. The user was asked to check his phone from time to time, work on the PC, answer incoming phone calls, chat with a colleague, and walk to a “meeting room” with his phone to meet a colleague. The script also included security attacks. Overall, the second part of the study lasted 40 minutes. The data collection ran continuously and collected sensor measurements from phone and PC, as well as types and time of applications invoked on the phone.

Each participant interacted with a total of five phone applications categorized as public (1 application), private (3) or confidential (1). This distribution reflects the results of our preliminary user study where across their most frequently used applications (*i.e.*, more than 10 times a day) participants wanted 22% of them to be always available (public) and the rest protected by some security mechanism (private or confidential). We then assumed a similar frequency of invocation across these 5 applications: across all participants, the average ratio at which public, private, and confidential applications were invoked was 19%:57%:24%, respectively.

6.1 Attack scenarios

In the script, we inserted 3 attack sessions for a total of 26 attack attempts (12 to private applications and 14 to confidential applications) covering the threat scenarios described in Section 2.2. In two sessions the user leaves the office and the phone on the desk, and soon after that an attacker (*i.e.*, a user unknown to the system) enters the room and tries to use the phone. These attacks take place in a private place (*i.e.*, office). We simulate both an attacker that knows which signals the system is using and one that does not know. In the first case, the attacker's strategies are staying out of the camera's range, being silent and entering the office soon after the user leaves and locks his PC. In the second case, the attacker is in the camera's range and speaks three times for about 10 sec-

onds. In each of the two sessions we have both types of attacks. The third type of attack occurs in a public place. The participant forgets his phone in a meeting room. The participant is asked to interact with the phone just until leaving the meeting room. The attacker enters the room soon after the user leaves, picks the phone up and tries to access the phone applications. These scenarios cover a small range of situations, but in the office setup we selected for the study they simulate all types of attacker strategies we considered in our threat model.

6.2 Training and testing the models

Low-level processing models were trained and tested using the data collected in the first part of the study. The inferred model for placement detection is the same across all users, while the face and voice recognition models are user specific (more details in Section 7.5). To train the high-level inference model, which is user-agnostic, we extracted feature vectors from each user's trace, and used WEKA [14], a popular suite of machine learning software, to train three models: decision tree, support vector machine with a non-linear model, and linear regression. As the top part of Figure 3 shows, the model is trained offline, across users. 8 users' traces are used for training and the remaining user's trace for testing. A feature vector is generated for each line in the user trace and then labeled with a ground truth label – the state the model is trained to recognize under different conditions. The ground truth labels are extracted by the study's observer based on a set of predetermined definitions. These definitions are used to provide an objective, quantifiable, and implementation-independent way of defining the ground truth labels:

Public Label: The legitimate owner is not present OR Other people are in contact with the phone OR The legitimate owner is present, but not in contact with the phone and other people are present.

Private Label: The legitimate owner has been in contact with the phone since the last private-level authentication OR The legitimate owner is present and is not in contact with the phone and no one else is present.

Confidential Label: The legitimate owner has been in contact with the phone since the last confidential-level authentication.

Across all participants, the distribution of ground truth labels was such that 55.0% of the labels were of type public (2379 labels), 42.6% were of type private (1843 labels), and the rest were of type confidential (2.4% or 105 labels). The distribution of the ground truth labels should not be confused with the distribution of application invocations and the type of application invoked. For instance, despite the participants invoked the confidential application as frequently as the public application or one

of the three private applications, only 2.4% of the labels were confidential. This means that only in few cases the signals the system could collect were sufficient to automatically authenticate the user at the confidential level. Hence, confidential applications required a PIN most of the time.

In the testing phase, the model was invoked using a user's trace. For each touch event which had been generated less than 0.5 seconds earlier, a feature vector was generated and fed as input to the model. To assess the model's accuracy, the output of the model was then compared against the ground truth label. We also tried training the high-level model on a per-user basis, but the accuracy of the resulting model was lower than that of the generalized model, so we did not pursue this strategy (more details in Section 7).

7 Experimental evaluation

We verify that our prototype meets the following goals: (1) Lowers authentication overhead on mobile phones; (2) Allows users to trade off stronger protection and more convenience; (3) Achieves reasonable accuracy in estimating the level of user authenticity; and (4) Provides acceptable execution time and power consumption.

7.1 Authentication overhead

The main goal of progressive authentication is to reduce the authentication overhead on mobile phones and become a viable (more secure) solution for users who currently do not use security locks on their devices. We measure how many times the participants executing the tasks of our script had to type a PIN and how many times they would have had to type a PIN without progressive authentication. We also count the number of unauthorized authentications (UAs) that occurred during the attack scenarios – cases of false authentication in which a non-legitimate user tried to unlock the phone and succeeded. For progressive authentication, we use an SVM model. As baseline schemes, we assume a system that locks the user's phone after 1 minute of inactivity (PhoneWithPIN) and one that never locks the phone (PhoneWithoutPIN). We choose a 1-minute timeout for the baseline case because the script was designed to allow for frequent application invocations in a limited time window. In real-life such invocations would be spread over longer time. Table 2 shows that, on average, compared to a state-of-the-art phone with PIN, progressive authentication reduces the number of times a user is requested to enter a PIN by 42% and provides the same security guarantees (*i.e.*, 0 UAs). Our claim is that such a system would be acceptable by users who currently do not use PINs on their phones. If for some of these users

Table 2: Reduction in the authentication overhead with progressive authentication (using an SVM model without loss function) compared to the two baseline cases available on today’s mobile phones. The table reports the number of times the user was required to enter a PIN (Num of PINs) and how many unauthorized authentications (Perc of UAs) occurred. Average and standard deviation are reported.

Avg [Stdev]	PhoneWithoutPIN	PhoneWithPIN	ProgAuth
Num of PINs	0.0 [0.0]	19.2 [0.6]	11.2 [0.4]
Perc of UAs	100% [0.0]	0.0% [0.0]	0.0% [0.0]

this reduction in overhead is not sufficient, they can decrease it even more by tuning the system’s risk factor, which we evaluate next.

7.2 Convenience and security trade-offs

As with password-based systems, for which users can set easy or hard passwords, with progressive authentication users can set a high or low risk factor (R). If R is high, the inference model is optimized for convenience – it reduces the number of false rejections (FRs), but it can possibly increase the number of false authentications (FAs). If R is low, the inference model is optimized for security. Recall that FAs are cases in which the system overestimates the level of the user authenticity and grants the user automatic access instead of requesting a PIN. Unauthorized accesses (UAs), reported in the previous test, represent a subset of the total number of FAs, as UAs only refer to cases in which non-legitimate users got access to the system. FAs refer to any user, legitimate or non-legitimate ones. FRs are cases in which the system underestimates the level of user authenticity and unnecessarily requires a password.

We configure 4 additional SVM models with increasing risk factors and compare their rates of FAs and FRs for private and confidential applications against a baseline without loss function (this is the same model used in Section 7.1, which uses $R = 1$). Table 3 shows that as R increases the percentage of FAs for private applications (FA Priv) increases from 4.9% to 16.1% and there are no FAs for confidential applications (FA Conf). Conversely, FRs decrease. With $R = 20$, the system reduces FR Priv to only 34.4% (*i.e.*, the user is required to enter a PIN for private applications 1 out of 3 times), but it still requires PINs for confidential applications most of the time (96.8%). This happens because the loss function used for optimizing the model always penalizes more strongly FAs for confidential applications (*i.e.*, if the penalty for

Table 3: Comparison of 5 SVM models, each with a different risk factor (R). One default model does not use any loss function ($R = 1$), while the other 4 are optimized either for convenience ($R = 5$ and $R = 20$) or for security ($R = 0.05$ and $R = 0.2$). The table reports percentage of false authentications and false rejections for private (FA Priv and FR Priv) and confidential (FA Conf and FR Conf) applications.

Risk factor	%FA Priv	%FA Conf	%FR Priv	%FR Conf
0.05	3.3	0.0	57.7	100.0
0.2	3.6	0.0	55.8	100.0
1	4.9	0.0	53.5	98.4
5	5.8	0.0	39.9	96.8
20	16.1	0.0	34.4	96.8

accessing private applications is P , that for confidential applications is P^2). This makes it very hard for the system to allow access to confidential applications without a PIN.

7.3 High-level processing accuracy

We have so far used an SVM model. We now evaluate the accuracy of high-level processing in more detail and compare SVM against two other popular modeling techniques: a decision tree with maximum depth of five and linear regression. All models are trained and cross-validated across users (*i.e.*, successively trained on every eight users and tested on the ninth user). None of the models use a loss function. Unlike the previous tests where we measured the accuracy of the model in estimating the level of user authenticity only for the first touch event leading to an application invocation, here we evaluate its accuracy in estimating the ground truth labels for all touch events extracted from the trace. This means that this analysis not only reports whether access to an application correctly required or did not require a PIN, but also whether while the application was in use the system was able to maintain the correct level of authentication or to de-authenticate the user as expected.

Precision and recall. We start by reporting the precision and recall for each model. Precision is defined as the fraction of correct predictions across all testing samples that resulted in the same prediction, while recall is defined as the fraction of correct predictions across all testing samples with the same ground truth label. As Figure 6(b) shows, SVM and decision tree outperform linear regression, which instead presents many incorrect predictions when the ground truth is private or confidential. This could lead to a disproportionate number of both FRs and FAs, so we discard it. We choose to use SVM

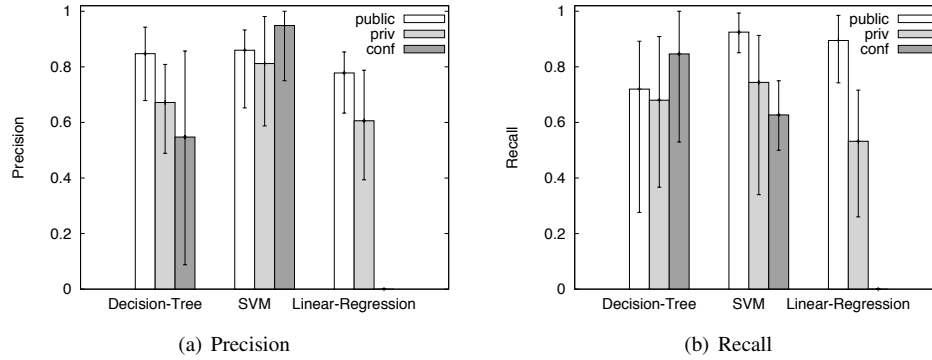


Figure 6: Precision and recall of different machine learning models when trained with 8 user traces and tested with the remaining one. The average and standard deviation is across the 9 users. No loss function is used.

Table 4: Confusion matrix for the SVM model.

	Rec. as Public	Rec. as Priv.	Rec. as Conf.
Public	92.50%	7.44 %	0.06%
Private	25.02%	74.42%	0.56%
Confidential	37.31%	0.00 %	62.69%

as the default inference model in our system because it is the most secure: it is able to recognize 92.5% of the public labels (high recall) which implies very few false authentications, and shows precision higher than 81% for all labels (Figure 6(a)). These graphs consider inferences done when both the authentic user and the attacker used the system. Since we saw in Table 2 that no unauthorized accesses occurred (*i.e.*, cases of false authentications when an attacker tried to access the system), all false authentications of SVM happened for authentic users and were due to incorrectly binning a user’s access as private/confidential instead of public. Decision tree is a more aggressive model which presents fewer FRs for confidential (higher recall for confidential), but generates more FAs (lower recall for public).

Table 4 further characterizes the results for SVM. It provides a complete breakdown of test outputs, including which states were incorrectly inferred when an incorrect inference was made (*i.e.*, how the difference between 100% and the height of the bar in Figure6(b) is broken down). We observe that most of the errors are false rejections for the confidential state: when the ground truth is confidential (last row in the table), the model labels it as public 37% of the time (first column) and it never labels it as private (second column). When inferring private states, it labels private states as public 25% of the time and almost never as confidential. The higher accuracy in inferring private states is an artifact of our traces and

the reliability of the sensor signals our implementation uses: the number of private ground truth labels (42.6%) was much higher than the confidential ones (2.4%), thus making the model better trained for inferring private labels.

From these results two observations follow. First, we do expect users to place commonly used applications in the private application bin more often than in the confidential one, hence requiring higher convenience for private and higher security for confidential. Second, one could argue that for the confidential level, the system should simply adopt password-based authentication and avoid making any inference. Our model does not exclude this option, but in an implementation with a larger number of more reliable signals, the system would be more likely to automatically detect the confidential level thus making the machine learning approach still useful at this level. Overall, these results show high potential in using machine learning models such as SVM in this context. False authentications are less than 8% and restricted to authentic users, and the system is able to infer the private/confidential states about 70% of the time.

Feature Importance. Next, we evaluate which features more strongly contributed in training the inference model. Table 5 shows the relative importance of features for SVM (in WEKA, this parameter is called GainRatioAttributeEvaluator: “it evaluates the worth of an attribute by measuring the gain ratio with respect to the class”). Features are shown in rank order (second column), along with their respective gain ratio (third column). The feature rank shows the importance of the corresponding authentication signals: possession (ProxAuthDev and ProxAuthDevConf), continuity (LastPlacementDuration), secrets (TimeSincePIN), and biometric (TimeSinceOwnerVoice). All types of signals envisioned for progressive authentication contributed to the inference model. Particularly, these results confirm the im-

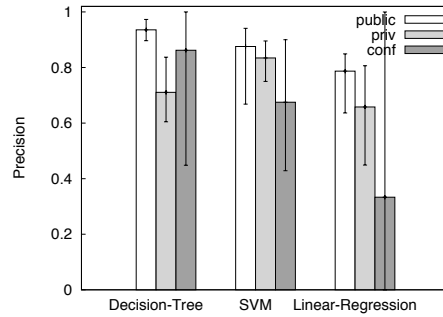
Table 5: Relative feature importance for SVM

Feature rank	Feature name	Gain ratio
1	ProxAuthDev	0.16105
2	LastPlacementDuration	0.09785
3	TimeSincePIN	0.04879
4	ProxAuthDevConf	0.04584
5	TimeSinceOwnerVoice	0.04554
6	TimeSinceProx	0.03919
7	TimeSinceTouch	0.02802
8	TimeSinceSound	0.02618
9	LastPlacement	0.02529
10	TimeSinceTable	0.02264
11	Placement	0.01849
12	TimeSinceHands	0.0174
13	TimeSinceNonOwnerVoice	0.01505
14	TimeSincePocket	0.01456
15	Speaker	0.00983
16	SpeakerConf	0.00907
17	PlacementDuration	0.00884
18	PlacementConf	0.00000

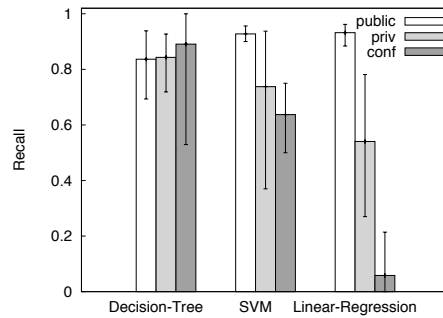
portance of recency and continuity of significant events that indicate user authenticity, such as being close to a known PC, having been in contact with the phone recently, having a PIN entered, or “hearing” the owner’s voice. However, the fact that in this specific implementation of progressive authentication ProxAuthDev is the top ranked feature does not mean that progressive authentication works only in scenarios where the user is nearby a PC. Instead, not surprisingly, the features ranked first are those derived from the most reliable sensor signals. For instance, ProxAuthDev is derived from BT-based proximity, activity detection, and PIN events.

7.4 Model personalization

We have so far evaluated high-level processing models trained across users because these are the models we expect to ship with the phones. However, it is possible to collect additional data from specific users to further tailor the model to their behavior. To assess the benefits of personalization, we evaluate the models when trained with data of a single user and tested through leave-one-out cross-validation. Figure 7 reports average precision and recall of these “personalized” models. Compared to the models trained across users (see Figure 6), SVM’s recall remains the same while the precision is slightly worse. Decision tree and linear regression show slightly better performance, but these modest improvements do not justify the overhead of training personal models. Perhaps with much longer traces for individual users we could see



(a) Precision



(b) Recall

Figure 7: Precision and recall of different models when tested through leave-one-out cross-validation. Average and standard deviation across users are shown. No loss function is used.

higher improvements, but otherwise these results confirm that building accurate generalized models that can be shipped with the phone is feasible.

7.5 Low-level processing accuracy

The low-level processing models include placement detection, face recognition and voice identification. Figure 8 reports the average accuracy (across 9 users) and variance for each recognition model. The accuracy is computed as the ratio of correct recognitions out of the total number of model invocations.

In the current prototype, **placement** can be in one of three possible states: “hands”, “table”, or “pocket”. Although they do not cover all possible states, we started with three states that are fairly common and sufficient to cover the scenarios in the script. To evaluate the placement detector accuracy, each participant performed a series of actions that put the phone in each of these states multiple times. In the meantime, sensor measurements were recorded: accelerometer samples were taken every 20 milliseconds, temperature and humidity every second, and light every 100 milliseconds. The placement recog-

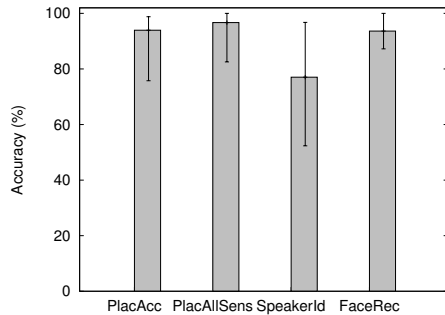


Figure 8: Average accuracy of placement, voice and face recognition models. Two versions of the placement model are tested: PlacAllSens relies on external Gadgeteer sensors and PlacAcc does not.

dition model was invoked every half second. We measured the accuracy of two decision tree models, one that uses only accelerometer data (PlacAcc), and another that is augmented with temperature/humidity and light sensors (PlacAllSens).

Figure 8 shows that the model accuracy varies from 83% to 100% when all sensors are used, and from 76% to 99% if only accelerometers are used. Without the added Gadgeteer sensors, the model still provides high accuracy. As Table 6 shows, the most common error consists of confusing a pocket placement with a table placement. Due to the prototype bulky form factor, the light sensor was sometimes left outside of the pocket, biasing the system towards a “table” state. The “hands” state was rarely confused with other states because of the high reliability of the temperature/humidity sensor (used to detect a human hand).

For **face recognition**, a webcam mounted on a desktop PC’s screen collected pictures of the participants. First, users were asked to look at the center of the screen (4 pictures) and then to look at 8 different points around the perimeter of the screen (2 pictures for each point). We trained a user-specific model using these 20 photos. For testing, a video was recorded while users read text on the screen and watched a ball moving in a figure of eight pattern across the entire screen. Roughly 1,000 frames were extracted from the video and used for testing. The whole recording was performed indoors and in the same lighting conditions (since it was from a desktop webcam). Figure 8 shows a 94% accuracy and a small variance across users. In scenarios with changing light conditions it is likely that the face recognition accuracy would decrease, but for office environments where lighting conditions are more stable this was not a problem. Nevertheless, when face recognition did not work we observed a small impact on the overall accuracy of our system because the face recognition signals are cross-

Table 6: Confusion matrix for the PlacAllSens model.

	Recogn. as Hand	Recogn. as Table	Recogn. as Pocket
Hand	98.78%	1.22%	0.003%
Table	0.01%	98.43%	1.56%
Pocket	0.12%	5.82%	94.06%

Table 7: Confusion matrix for voice identification.

	Rec. as Owner’s voice	Rec. as Other’s voice or Unknown
Owner’s voice	77.0%	23.0%
Other’s voice	0.4%	99.6%

checked against other presence signals collected using activity detection sensors (*e.g.*, login/logout events, typing on keyboard or moving a mouse) which can be estimated with much higher reliability.

Finally, we evaluated the accuracy of the **speaker identification** model [21]. In the study, the participants’ voice was recorded for roughly 10 minutes using the phone’s microphone at 16kHz, 16bit mono format. All samples were collected in an office environment where the user’s voice was the dominant signal and there was modest background noise such as people talking in the hall, keyboard typing, or air conditioning. The phone was placed within 1 to 2 meters from the participant. The participant was asked to speak or read some text (4 of them only read text, 1 only spoke, and 4 alternated between reading and speaking). This model requires multiple voices for training, so we created a single model with the samples we collected. The model was trained using 2 minutes from each of the users and the voice of all 9 participants was tested against the model (*i.e.*, a user’s voice may be recognized as somebody’s else voice or as unknown). Compared to the other models in Figure 8, speaker identification varied the most across users. There were 3 participants for whom the average accuracy was roughly 59.3%, while for all the other users the model achieved at least 83.7% accuracy. However, Table 7 shows that the system rarely recognized another voice as the owner’s voice (0.4%) – false positives were rare. Most of the errors were due to the system not recognizing the user (23%) – false negatives. In the actual study (the second part), these errors were unfortunately amplified, perhaps due to the variety of conditions in which the phone was used (the speaker was at a variable distance from the user, sometimes the phone was in a pocket while the user spoke, different background noise, etc.).

Table 8: Power consumption and execution time on a Samsung Focus WP 7.1 for 4 different power configurations.

Conf	Sensing (mW)	Comput (mW)	Comm (mW)	TotalPower (mW)	ExTime (sec)
LocalMin	<1	41	0	42	0.20
Local	≈160	447	44	651	0.23
LocalRemote	≈160	71	94	325	0.99
Remote	≈160	49	98	307	1.50-2.81

7.6 Latency and power consumption

We evaluate latency and power overhead using a 2-device configuration including a WP 7.1 Samsung Focus using WiFi for communication and a Windows PC (2.66GHz Intel Xeon W3520 CPU with 6GB of RAM). We consider four device configurations:

- *LocalMin*: low-level and high-level processing runs on the phone, however power-consuming tasks (BT-based proximity detection and voice identification/recognition) are disabled.
- *Local*: all low-level and high-level processing tasks depicted in Figure 3 run on the phone.
- *LocalRemote*: computation-light low-level processing runs on the phone while voice identification and high-level processing run on the PC (*i.e.*, gray-colored modules in Figure 3 are offloaded).
- *Remote*: both low-level and high-level processing runs on the PC. The phone only runs the sensors and sends raw measurements to the PC.

We measured the power drained from the phone battery by connecting it to a Monsoon Power Monitor [26], specifically designed for Windows phones. The Monsoon meter supplies a stable voltage to the phone and samples the power consumption at a rate of 5KHz. During the measurements, the phone had the display and WiFi on, which corresponds to an idle power consumption of 896 mW. Table 8 shows power consumption and execution time results for all four configurations. LocalMin is the best configuration from both a power consumption and latency point of view: it consumes only 42 mW and has an average delay of 200 msec. However, as this configuration disables features derived from proximity and voice recognition, its model may provide lower accuracy compared to the other 3 configurations, which allow for the complete set of features. Specifically, LocalMin’s SVM model is able to reduce the number of PINs by 53%, but it presents 70% UAs to private applications. On the other hand, the model still provides

0% UAs to confidential applications. Although less accurate, this model still provides advantages compared to an unprotected system, thus being an option for users who currently do not use a PIN on their phone.

Among the other 3 configurations, the best compromise from a power-latency point of view is LocalRemote – this is also the default configuration of our system. Its delay is less than 1 second and it consumes about 325 mW, which may be acceptable for modern phones. The reason for such a reduction in power consumption compared to Local is that this configuration offloads voice identification to the PC thus significantly reducing the power drained by the phone’s CPU. Basically, the phone uploads voice recordings only if voice is detected (*i.e.*, it does not upload raw data if no voice is present). Local represents the fastest full configuration with an average execution time of 225 msec per inference, including feature extraction (≈ 90 msec) and SVM invocation (≈ 115 msec). This configuration may be convenient for a phone at home or in an office, with full power or currently connected to a power charger.

We have shown a range of options. Executing on the client is faster, using the server for remote execution yields lower power costs. Temporarily disabling computation-intensive features can also significantly lower power consumption at the cost of accuracy. By switching between these configurations and possibly even temporarily disabling progressive authentication, we can deliver a system with lower authentication overhead and acceptable power and latency requirements. In general, users who currently do not use PINs on their phones can have a much more protected system without the need to worry about power consumption.

8 Related work

Other researchers have explored work related to progressive authentication in the following areas: multi-level authentication systems, context-based and automatic authentication, and mobile device authentication in general.

8.1 Multi-level authentication

Multi-level authentication has been considered before. As in progressive authentication, data and applications are categorized in different levels of authorization, variously called “hats” [34], “usage profiles” [19], “spheres” [32], “security levels” [4], or “sensitive files” [36]. With the exception of TreasurePhone [32] and MULE [36], most of this work has been conceptual, with no actual implementation. TreasurePhone divides applications into multiple access spheres and switches from one sphere to another using the user’s location, a personal token, or physical “actions” (*e.g.*, lock-

ing the home door would switch from the “Home” to the “Closed” sphere). However, these sphere switching criteria have flaws. First, location is rather unreliable and inaccurate, and when used in isolation, it is difficult to choose the appropriate sphere (*e.g.*, being alone at home is different than being at home during a party). Second, the concept of personal tokens requires users to carry more devices. Third, monitoring physical “actions” assumes that the device can sense changes in the physical infrastructure, something that is not yet viable. Conversely, progressive authentication enables automatic switching among the multiple levels of authentication by relying on higher-accuracy, simpler and more widely available multi-modal sensory information. MULE proposes to encrypt sensitive files stored in laptops based on their location: if the laptop is not at work or at home, these files are encrypted. Location information is provided by a trusted location device that is contacted by the laptop in the process of regenerating decryption keys. Progressive authentication protects applications, not files, and it uses multiple authentication factors, unlike MULE, which uses location exclusively.

8.2 Automatic authentication

Other forms of automatic authentication use a single authentication factor such as proximity [6, 7, 18], behavioral patterns [33], and biometrics, such as typing patterns [1, 24], hand motion and button presses [3]. Most of these techniques are limited to desktop computers, laptops or specific devices (*e.g.*, televisions [3]). The closest to our work is Implicit Authentication [33], which records a user’s routine tasks such as going to work or calling friends, and builds a profile for each user. Whenever deviations from the profile are detected, the user is required to explicitly authenticate. Progressive authentication differs from this work in that it uses more sensory information to enable real-time, finer granularity modeling of the device’s authentication state. On the other hand, any of those proximity, behavioral and biometric patterns could be plugged into our system. Transient authentication [6, 7] requires the user to wear a small token and authenticate with it from time to time. This token is used as a proximity cue to automate laptop authentication. This approach requires the user to carry and authenticate with an extra token, but its proximity-based approach is relevant to our work in that it also leverages nearby user-owned devices (*i.e.*, the tokens) as authentication signals.

8.3 Mobile device authentication

The design of more intuitive and less cumbersome authentication schemes has been a popular research

topic. Current approaches can be roughly classified into knowledge-based, multi-factor, and biometric authentication techniques. All three are orthogonal to progressive authentication. Our goal is not to provide a new “explicit” authentication mechanism, but instead to increase the usability of current mechanisms by reducing the frequency at which the user must authenticate. When explicit authentication is required, any of these techniques can be used.

Knowledge-based approaches assume that a secret (*e.g.*, a PIN) is shared between the user and the device, and must be provided every time the device is used. Due to the limited size of phone screens and on-screen keyboards, this can be a tedious process [5], especially when it is repeated multiple times per day. In multi-factor authentication, more than one type of evidence is required. For instance, two-factor authentication [2, 31, 35] requires a PIN and secured element such as a credit card or USB dongle. This practice presents major usability issues, as the need to carry a token such as SecurID [31] goes against the user’s desire to carry fewer devices. Biometric schemes [5, 16, 27] leverage biometrics [17] or their combinations [12, 15], such as face recognition and fingerprints, to authenticate the user with high accuracy. Even though very secure, biometric identification comes with acceptability, cost and privacy concerns [27], and is especially cumbersome on small devices.

9 Conclusions

We presented a novel approach to progressively authenticate (and de-authenticate) users on mobile phones. Our key insight is to combine multiple authentication signals to determine the user’s level of authenticity, and surface authentication only when this level is too low for the content being requested. We have built a system prototype that uses machine learning models to implement this approach. We used the system in a lab study with nine users and showed how we could reduce the number of explicit authentications by 42%. We believe our results should make this approach attractive to many mobile users who do not use security locks today. Overall, progressive authentication offers a new point in the design of mobile authentication and provides users with more options in balancing the security and convenience of their devices.

10 Acknowledgments

We thank the authors of Speaker Sense for making it available to us. Special thanks to Eiji Hayashi for many insightful discussions on the architecture of progressive authentication and for designing its user interface. Thanks to Asela Gunawardana for helping us with the

machine learning models used by the system, and to Tom Blank, Jeff Herron, Colin Miller and Nicolas Villar for helping us instrumenting the WP with additional sensors. We also thank David Molnar and Bryan Parno for their comments on an earlier draft of this paper, and the anonymous reviewers for their insightful feedback. Finally, we are especially grateful to all users who participated in our user studies.

References

- [1] BERGADANO, F., GUNETTI, D., AND PICARDI, C. User authentication through keystroke dynamics. *ACM Trans. Inf. Syst. Secur.* 5 (November 2002), 367–397.
- [2] C.G.HOCKING, S.M.FURNELL, N.L.CLARKE, AND P.L.REYNOLDS. A distributed and cooperative user authentication framework. In *Proc. of IAS '10* (August 2010), pp. 304–310.
- [3] CHANG, K.-H., HIGHTOWER, J., AND KVETON, B. Inferring identity using accelerometers in television remote controls. In *Proc. of Pervasive '09* (2009), pp. 151–167.
- [4] CLARKE, N., KARATZOUNI, S., AND FURNELL, S. Towards a Flexible, Multi-Level Security Framework for Mobile Devices. In *Proc. of the 10th Security Conference* (May 4–6 2011).
- [5] CLARKE, N. L., AND FURNELL, S. M. Authentication of users on mobile telephones - A survey of attitudes and practices. *Computers and Security* 24, 7 (Oct. 2005), 519–527.
- [6] CORNER, M. D., AND NOBLE, B. Protecting applications with transient authentication. In *Proc. of MobiSys '03* (2003), USENIX.
- [7] CORNER, M. D., AND NOBLE, B. D. Zero-interaction authentication. In *Proc. of MobiCom '02* (2002), ACM, pp. 1–11.
- [8] CUERVO, E., BALASUBRAMANIAN, A., CHO, D.-K., WOLMAN, A., SAROJU, S., CHANDRA, R., AND BAHL, P. MAUI: making smartphones last longer with code offload. In *Proc. of MobiSys '10* (2010), ACM, pp. 49–62.
- [9] Gadgeteer. <http://netmf.com/gadgeteer/>.
- [10] GIURGIU, I., RIVA, O., JURIC, D., KRIVULEV, I., AND ALONSO, G. Calling the cloud: Enabling mobile phones as interfaces to cloud applications. In *Proc. of Middleware '09* (November 30 - December 4 2009), Springer.
- [11] How Apple and Google will kill the password. http://www.computerworld.com/s/article/9206998/How_Apple_and_Google_will_kill_the_password..
- [12] GREENSTADT, R., AND BEAL, J. Cognitive security for personal devices. In *Proc. of the 1st ACM workshop on AISec* (2008), ACM, pp. 27–30.
- [13] HAYASHI, E., RIVA, O., BRUSH, A., STRAUSS, K., AND SCHECHTER, S. Goldilocks and the Two Mobile Devices: Going Beyond All-Or-Nothing Access to a Device's Applications. In *Proc. of SOUPS '12* (July 11-13 2012), ACM.
- [14] HOLMES, G., DONKIN, A., AND WITTEN, I. Weka: A machine learning workbench. In *Proc. of the 2nd Australia and New Zealand Conference on Intelligent Information Systems* (December 1994), pp. 357–361.
- [15] HONG, L., AND JAIN, A. Integrating faces and fingerprints for personal identification. *IEEE Trans. Pattern Anal. Mach. Intell.* 20 (December 1998), 1295–1307.
- [16] JAIN, A., BOLLE, R., AND PANKANTI, S. *Biometrics: Personal Identification in a Networked Society*. Kluwer Academic Publ., 1999.
- [17] JAIN, A., HONG, L., AND PANKANTI, S. Biometric identification. *Commun. ACM* 43 (February 2000), 90–98.
- [18] KALAMANDEEN, A., SCANNELL, A., DE LARA, E., SHETH, A., AND LAMARCA, A. Ensemble: cooperative proximity-based authentication. In *Proc. of MobiSys '10* (2010), pp. 331–344.
- [19] KARLSON, A. K., BRUSH, A. B., AND SCHECHTER, S. Can I borrow your phone?: Understanding concerns when sharing mobile phones. In *Proc. of CHI '09* (2009), ACM, pp. 1647–1650.
- [20] LIU, Y., RAHMATI, A., HUANG, Y., JANG, H., ZHONG, L., ZHANG, Y., AND ZHANG, S. xShare: supporting impromptu sharing of mobile phones. In *Proc. of MobiSys '09* (2009), ACM, pp. 15–28.
- [21] LU, H., BRUSH, A. J. B., PRIYANTHA, B., KARLSON, A. K., AND LIU, J. SpeakerSense: Energy Efficient Unobtrusive Speaker Identification on Mobile Phones. In *Proc. of Pervasive 2011* (June 12-15 2011), pp. 188–205.
- [22] Mobile wallet offered to UK shoppers. <http://www.bbc.co.uk/news/technology-13457071>.
- [23] NI, X., YANG, Z., BAI, X., CHAMPION, A. C., AND XUAN, D. Diffuser: Differentiated user access control on smartphone. In *Proc. of MASS '09* (12-15 October 2009), IEEE, pp. 1012–1017.
- [24] NISENSEN, M., YARIV, I., EL-YANIV, R., AND MEIR, R. Towards behaviorometric security systems: Learning to identify a typist. In *Proc. of PKDD '03* (2003), Springer, pp. 363–374.
- [25] 24% of mobile users bank from a phone. Yet most don't have security measures in place. <http://www.bullguard.com/news/latest-press-releases/press-release-archive/2011-06-21.aspx>.
- [26] Monsoon Power Monitor. <http://www.msoon.com/LabEquipment/PowerMonitor/>.
- [27] PRABHAKAR, S., PANKANTI, S., AND JAIN, A. K. Biometric recognition: Security and privacy concerns. *IEEE Security and Privacy* 1 (2003), 33–42.
- [28] PRIYANTHA, B., LYMBERPOULOS, D., AND LIU, J. LittleRock: Enabling Energy Efficient Continuous Sensing on Mobile Phones. Tech. Rep. MSR-TR-2010-14, Microsoft Research, February 18 2010.
- [29] PRIYANTHA, B., LYMBERPOULOS, D., AND LIU, J. LittleRock: Enabling Energy-Efficient Continuous Sensing on Mobile Phones. *IEEE Pervasive Computing* 10 (2011), 12–15.
- [30] REYNOLDS, D. A. An overview of automatic speaker recognition technology. In *Proc. of ICASSP '02* (2002), vol. 4, pp. IV-4072–IV-4075.
- [31] RSA SecurID. <http://www.rsa.com/node.aspx?id=1156>.
- [32] SEIFERT, J., DE LUCA, A., CONRADI, B., AND HUSSMANN, H. TreasurePhone: Context-Sensitive User Data Protection on Mobile Phones. In *Proc. of Pervasive '10*. 2010, pp. 130–137.
- [33] SHI, E., NIU, Y., JAKOBSSON, M., AND CHOW, R. Implicit authentication through learning user behavior. In *Proc. of ISC '10* (October 2010), pp. 99–113.
- [34] STAJANO, F. One user, many hats; and, sometimes, no hat – towards a secure yet usable PDA. In *In Proc. of Security Protocols Workshop* (2004).
- [35] STAJANO, F. Pico: No more passwords! In *Proc. of Security Protocols Workshop* (March 28–30 2011).
- [36] STUDER, A., AND PERRIG, A. Mobile user location-specific encryption (MULE): using your office as your password. In *Proc. of WiSec '10* (2010), ACM, pp. 151–162.
- [37] TEXAS INSTRUMENTS. OMAPTM 5 mobile applications platform, 13 July 2011. Product Bulletin.

Origin-Bound Certificates: A Fresh Approach to Strong Client Authentication for the Web

Michael Dietz
Rice University
mdietz@rice.edu

Alexei Czeskis
University of Washington
alexei@czeskis.com

Dirk Balfanz*
Google Inc.
balfanz@google.com

Dan S. Wallach
Rice University
dwallach@rice.edu

Abstract

Client authentication on the web has remained in the internet-equivalent of the stone ages for the last two decades. Instead of adopting modern public-key-based authentication mechanisms, we seem to be stuck with passwords and cookies.

In this paper, we propose to break this stalemate by presenting a fresh approach to public-key-based client authentication on the web. We describe a simple TLS extension that allows clients to establish strong authenticated channels with servers and to *bind* existing authentication tokens like HTTP cookies to such channels. This allows much of the existing infrastructure of the web to remain unchanged, while at the same time strengthening client authentication considerably against a wide range of attacks.

We implemented our system in Google Chrome and Google's web serving infrastructure, and provide a performance evaluation of this implementation.

1 Introduction

In the summer of 2011, several reports surfaced of attempted man-in-the-middle attacks against Google users who were primarily located in Iran. The Dutch certification authority DigiNotar had apparently issued certificates for *google.com* and other websites to entities not affiliated with the rightful owners of the domains in question¹. Those entities were then able to pose as Google and other web entities and to eavesdrop on the communication between users' web browsers and the websites they were visiting. One of the pieces of data such eavesdroppers could have conceivably recorded were *authentication cookies*, meaning that the man-in-the-middle

could have had full control over user accounts, even after the man-in-the-middle attack itself was over.

This attack should have never been possible: authenticating a client to a server while defeating man-in-the-middle attacks is theoretically a solved problem. Simply put, client and server can use an authenticated key agreement protocol to establish a secure permanent "channel." Once this channel is set up, a man-in-the-middle cannot "pry it open", even with stolen server certificates.

Unfortunately, this is *not* how authentication works on the web. We neither use sophisticated key agreement protocols, nor do we establish authenticated "channels." Instead, we send secrets directly from clients to servers with practically every request. We do this across all layers of the network stack. For example, to authenticate users, passwords are sent from clients to servers; SAML or OpenID assertions are sent from clients to servers in order to extend such user authentication from one website to another; and HTTP cookies are sent with every HTTP request after the initial user authentication in order to authenticate that HTTP request.

We call this pattern *bearer tokens*: the bearer of a token is granted access, regardless of the channel over which the token is presented, or who presented it².

Unfortunately, bearer tokens are susceptible to certain classes of attacks. Specifically, an adversary that manages to steal a bearer token from a legitimate user can impersonate that user to web services that require it. For different kinds of bearer tokens these attacks come in different flavors: passwords are usually obtained through phishing or keylogging, while cookie theft happens through man-in-the-browser malware (*e.g.*, Zeus [16]), cross site scripting attacks, or adversaries that manage to sniff the network or insert themselves into the network between the client and server [1, 7]).

The academic community, of course, has known of authentication mechanisms that avoid the weaknesses of

¹The opinions expressed here are those of the authors and do not necessarily reflect the position of Google.

²It later turned out that the certificates had, in fact, been created fraudulently by attackers that had compromised DigiNotar.

²Bearer tokens, originally called "sparse capabilities" [25], were widely used in distributed systems, well before the web.

bearer tokens since before the dawn of the web. These mechanisms usually employ some form of public-key cryptography rather than a shared secret between client and server. Authentication protocols based on public-key cryptography have the benefit of not exposing secrets to the eavesdropper which could be used to impersonate the client to the server. Furthermore, when public/private key pairs are involved, the private key can be moved out of reach of thieving malware on the client, perhaps using a hardware Trusted Platform Module (TPM). While in theory this problem seems solved, in *practice* we have seen attempts to rid the web of bearer tokens gain near-zero traction [10] or fail outright [13].

In this paper, we present a fresh approach to using public-key mechanisms for strong authentication on the web. Faced with an immense global infrastructure of existing software, practices and network equipment, as well as users' expectations of how to interact with the web, we acknowledge that we cannot simply "reboot" the web with better (or simply different) authentication mechanisms. Instead, after engaging with various stakeholders in standards bodies, browser vendors, operators of large website, and the security, privacy and usability communities, we have developed a layered solution to the problem, each layer consisting of minor adjustments to existing mechanisms across the network stack.

The key contributions of this work are:

- We present a slight modification to TLS client authentication, which we call *TLS-OBC*. This new primitive is simple and powerful, allowing us to create strong TLS channels.
- We demonstrate how higher-layer protocols like HTTP, federation protocols, or even application-level user login can be hardened by "binding" tokens at those layers to the authenticated TLS channel.
- We describe our efforts in gaining community support for an IETF draft [2], as well as support from major browser vendors; both Google's Chrome and Mozilla's Firefox have begun to incorporate and test support for TLS-OBC.
- We present a detailed report on our client-side implementation in the open-source Chromium browser, and our server-side implementation inside the serving infrastructure of a large website.
- We give some insight into the process that led to the proposal as presented here, contrasting it with existing work and explaining real-world constraints, ranging from privacy expectations that need to be weighed against security requirements, to deployment issues in large datacenters.

Summary. The main idea of this work is easily explained: browsers use self-signed client certificates

within TLS client authentication. These certificates are generated by the browser on-the-fly, as needed, and contain no user-identifying information. They merely serve as a foundation upon which to establish an authenticated *channel* that can be re-established later.

The browser generates a different certificate for every website to which it connects, thus defeating any cross-site user tracking. We therefore call these certificates *origin-bound certificates* (OBCs). This design choice also allows us to completely decouple certificate generation and use from the user interface; TLS-OBC client authentication allows the existing web user experience to remain the same, despite the changes under the hood.

Since the browser will consistently use the same client certificate when establishing a TLS connection with an origin, the website can "bind" authentication tokens (*e.g.*, HTTP cookies) to the OBC, thereby creating an authenticated channel. This is done by simply recording which client certificate should be used at the TLS layer when submitting the token (*i.e.*, cookie) back to the server. It is at *this* layer (in the cookie, not in the TLS certificate) that we establish user identity, just as it is usually done on the web today.

TLS-OBC's channel-binding mechanism prevents stolen tokens (*e.g.*, cookies) from being used over other TLS channels, thereby making them useless to token thieves, solving a large problem in today's web.

2 Threat Model

We consider a fairly broadly-scoped (and what we believe to be a real-world) threat model. Specifically, we assume that attackers are occasionally able to "pry open" TLS sessions and extract the enclosed sensitive data by exploiting a bug in the TLS system [22], mounting a man in the middle (MITM) attack through stolen server TLS certificates [1], or utilizing man-in-the-browser malware [16]. These attacks not only reveal potentially private data, but in today's web will actually allow attackers to impersonate and completely compromise user accounts by capturing and replaying users' authentication credentials (which, as we noted earlier, are usually in the form of bearer tokens). These attacks are neither theoretical nor purely academic; they *are* being employed by adversaries in the wild [24].

In this paper we focus on the TLS and HTTP layers of the protocol stack, and on protecting the authentication tokens at those layers—mostly HTTP cookies (but also identity assertions in federation protocols)—by binding them to the underlying authenticated TLS-OBC channel. We have a parallel effort under way to protect the application-layer user logins, but that is mostly outside the scope of this paper. To model this distinction, we consider two classes of attacker. The first class

is an attacker that has managed to insert themselves as a MITM *during* the initial authentication step (when the user trades his username/password credentials for a cookie), or an attacker that steals user passwords through a database compromise or phishing attack. The second class of attacker is one that has inserted itself as a MITM *after* the initial user authentication step where credentials are traded for an authentication token. The first class of attacker is strictly stronger than the second class of attacker as a MITM that can directly access a user's credentials can trade them in for an authentication token at his leisure. While the second class of attacker, a MITM that can only steal the authentication token, has a smaller window of opportunity (the duration for which the cookie is valid) for access to the user's private information.

For the purposes of this paper, we choose to focus on the second class of attacker. In short, we assume that the user has already traded their username/password credentials to acquire an authentication token that will persist across subsequent connections. Our threat model allows for attackers to exploit MITM or eavesdropping attacks during any TLS handshake or session subsequent to the initial TLS connection to a given endpoint—including attacks that cause a user to re-authenticate as discussed in Section 4.3. Attacks that target user credentials during the initial TLS connection, rather than authentication tokens during subsequent TLS connections, are dealt with in a forthcoming report.

3 TLS-OBC

We propose a slightly modified version of traditional TLS client certificates, called Origin-Bound Certificates (OBCs), that will enable a number of useful applications (as discussed in Section 4).

3.1 Overview

Fundamentally, an Origin-Bound Certificate is a self-signed certificate that browsers use to perform TLS Client Authentication. Unlike normal certificates, and their use in TLS Client Authentication (see Section 8.1), OBCs do not require any interaction with the user. This property stems from the observation that since the browser generates and stores only one certificate per origin, it's always clear to the browser which certificate it must use; no user input is necessary to make the decision.

On-Demand Certificate Creation If the browser does not have an existing OBC for the origin it's connecting to, a new OBC will be created on-the-fly. This newly generated origin-bound certificate contains *no* user iden-

tifying information (*e.g.*, name or email). Instead, the OBC is used only to prove, cryptographically, that subsequent TLS sessions to a given server originate from the same client, thus building a continuous TLS *channel*³, even across different TLS sessions.

User Experience As noted earlier, there is no user interface for creating or using Origin-Bound Certificates. This is similar to the UI for HTTP cookies; there is typically no UI when a cookie is set nor when it is sent back to the server. Origin-Bound Certificates are similar to cookies in other ways as well:

- Clients use a different certificate for each origin. Unless the origins collaborate, one origin cannot discover which certificate is used for another.
- Different browser profiles use different Origin-Bound Certificates for the same origin.
- In incognito or private browsing mode, the Origin-Bound Certificates used during the browsing session get destroyed when the user closes the incognito or private browsing session.
- In the same way that browsers provide a UI to inspect and clean out cookies, there should be a UI that allows users to reset their Origin-Bound Certificates.

3.2 The Origin-Bound Certificates TLS Extension

OBCs do not alter the semantics of the TLS handshake and are sent in exactly the same manner as traditional client certificates. However, because they are generated on-the-fly and have no associated UI component, we must differentiate TLS-OBC from TLS client-auth and treat it as a distinct TLS extension. Figure 1 shows, at a high level, how this extension fits in with the normal TLS handshake protocol; the specifics of the extension are explained below.

The first step in the client-server decision to use OBCs occurs when the client advertises acceptance of the TLS-OBC extension in its initial `ClientHello` message. If the server chooses to accept the use of OBCs, it echoes the TLS-OBC extension identifier in its `ServerHello` message. At this point, the client and server are considered to have negotiated to use origin-bound client certificates for the remainder of the TLS session.

After OBCs have been negotiated, the server sends a `CertificateRequest` message to the client that specifies the origin-bound certificate types that it will accept (ECDSA, RSA, or both). Upon a client's receipt of the `CertificateRequest`, if the client has already generated an OBC associated with the server endpoint,

³We use the same notion as TAOS [27] does, of a cryptographically strong link between two nodes.

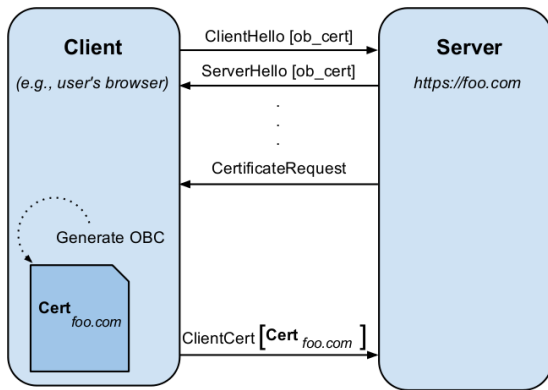


Figure 1: TLS-OBC extension handshake flow.

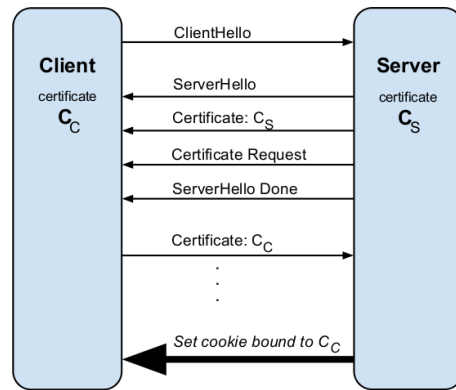


Figure 2: Process of setting an OBC bound cookie

the existing OBC is returned to the server in the client’s ClientCertificate message. If this is the first connection to the server endpoint or if no acceptable existing OBC can be found, an origin-bound certificate must be generated by the client then delivered to the server in the client’s ClientCertificate message.

During the OBC generation process, the client creates a self-signed client certificate with common and distinguished names set to “anonymous.invalid” and an X509 extension that specifies the origin for which the OBC was generated.

4 Securing Web Authentication Mechanisms with TLS-OBC

We now show how origin-bound certificates can be used to strengthen other parts of the network stack: In Section 4.1 we explain how HTTP cookies can be bound to TLS channels using TLS-OBC. In Section 4.2 we show how federation protocols (such as OpenID or OpenID Connect) can be hardened against attackers, and in Section 4.3 we turn briefly to application-level user authentication protocols.

4.1 Channel-binding cookies

OBCs can be used to strengthen cookie-based authentication by “binding” cookies to OBCs. When issuing cookies for an HTTP session, servers can associate the client’s origin-bound certificate with the session (either by unforgeably encoding information about the certificate in the cookie value, or by associating the certificate with the cookie’s session through some other means). That way, if and when a cookie gets stolen from a client, it cannot be used to authenticate a user when communicated over a TLS connection initiated by a different client – the cookie thief would also have to steal the private key associated with the client’s origin-bound certificate

– a task considerably harder to achieve (especially in the presence of Trusted Platform Modules or other Secure Elements that can protect private key material).

Service Cookie Hardening One way of unforgeably encoding an OBC into a cookie is as follows. If a traditional cookie is set with value v , a channel bound cookie may take the form of:

$$\langle v, \text{HMAC}_k(v + f) \rangle$$

where v is the value, f is a fingerprint of the client OBC, k is a secret key (known only to the server), and $\text{HMAC}_k(v + f)$ is a keyed message authentication code computed over v concatenated to f with key k . This information is all that is required to create and verify a channel bound cookie. The general procedure for setting a hardened cookie is illustrated in Figure 2. Care must be taken not to allow downgrade attacks: if both v and $\langle v, \text{HMAC}_k(v + f) \rangle$ are considered valid cookies, a man-in-the-middle might be able to strip the signature and simply present v to the server. Therefore, the protected cookie *always* has to take the form of $\langle v, \text{HMAC}_k(v + f) \rangle$, even if the client doesn’t support TLS-OBC.

Cookie Hardening for TLS Terminators The technique for hardening cookies, as discussed above, assumes that the cookie-issuing service knows the OBC of the connecting client. While this is a fair assumption to make for most standalone services, it is not true for many large-scale services running in datacenters. In fact, for optimization and security reasons, some web services have TLS “terminators”. That is, all TLS requests to and from an application are first passed through the TLS terminator node to be “unwrapped” on their way in and are “wrapped” on their way out.

There are two potential approaches to cookie hardening with TLS terminators. First, TLS terminators could extract a client’s OBC and pass it, along with other information about the HTTP request (such as cookies sent by

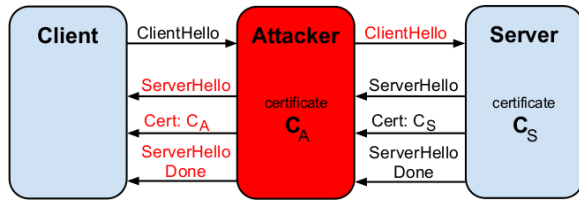


Figure 3: MITM attack during a TLS handshake

the client) to the backend service. The backend service can then create and verify channel-bound cookies using the general procedure in the previous section.

The second approach involves using the TLS terminator to channel-bind the cookies of legacy services that cannot or will not be modified to deal with OBC information sent to them by the TLS terminator. Using this approach, TLS terminators must receive a list of cookie names to harden for each service to which they cater. When receiving an outbound HTTP response with a Set-Cookie header for a protected cookie, the TLS terminator must compute the hardened value using the OBC fingerprint, rewrite the cookie value in the Set-Cookie header, and only then wrap the request in a TLS stream. Similarly, the TLS terminator must inspect incoming requests for Cookie headers bearing a protected cookie, validate them, and rewrite them to only have the raw value. Any inbound request with a channel-bound cookie that fails verification must be dropped by the TLS verifier.

Channel-Bound Cookies Protect Against MITM

As mentioned earlier, TLS MITM attacks happen and some can go undetected (see Figure 3 for a depiction of a conventional MITM attack). Channel-bound cookies can be used to bring protection against MITM attacks to web users.

Recall that our threat model assumes that at some time in the past, the user's client was able to successfully authenticate with the server. At that point, the server would have set a cookie on the client and would have bound that cookie to the client's legitimate origin-bound certificate. This process is shown in Figure 2. Observe that on a subsequent visit, the client will send its cookie (bound to the client's OBC). However, the MITM lacks the ability to forge the client's OBC and must substitute a new OBC in its handshake with the server. Therefore, when the MITM forwards the user's cookie on to the server, the server will recognize that the cookie was bound to a different OBC and will drop the request. This process is shown in Figure 4. The careful reader will observe that a MITM attacker may strip the request of any bearer tokens completely and force the user to provide his username/password once more or fabricate a new cookie and

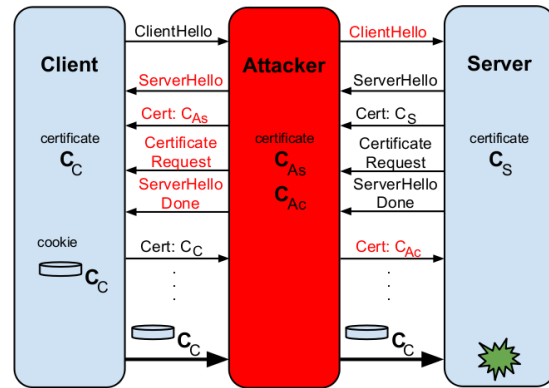


Figure 4: Using OBCs and bound cookies to protect against MITM. The server recognizes a mismatch between the OBC to which the cookie is bound and the cert of the client (attacker) with who it is communicating.

log the user in as another identity. We cover this more in Section 4.3 and in an upcoming report.

4.2 Hardening Federation Protocols

Channel-binding cookies with OBCs allows a single entity to protect the authentication information of its users, but modern web users have a plethora of other login credentials and session tokens that make up their digital identity. Federation protocols like OpenID [20], OpenID Connect [23], and BrowserID [14] have been proposed as a way to manage this explosion of user identity state. At a high level, these federation protocols allow the user to maintain a single account with an identity provider (IdP). This IdP can then generate an *identity assertion* that demonstrates to relying parties that the user controls the identity established with the identity provider. While these federation techniques reduce the number of credentials a user is responsible for remembering, they make the remaining credentials much more valuable. It is therefore critical to protect the authentication credentials for the identity provider as well as the mechanism used to establish the identity assertion between identity provider and relying party. Towards that end, we explore using TLS-OBC and channel-binding to harden a generic federation system against attack.

PostKey API The first step towards hardening a federation protocol is to provide a way for an identity provider and *relying party* to communicate in a secure, MITM resistant manner. We introduce a new browser API called the *PostKey* API to facilitate this secure communication. This new API is conceptually very similar to the *PostMessage* [11] communication mechanism that allows distinct windows within the browser to send messages to each other using inter-process communication

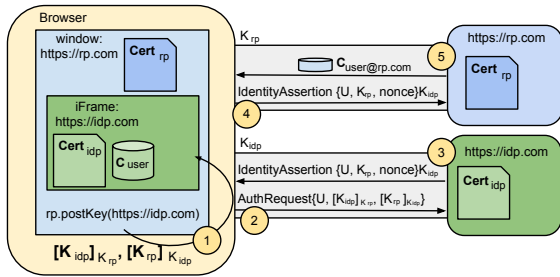


Figure 5: Simplified federation protocol authorization flow using PostKey and OBCs.

rather than the network. The goal of *PostKey* extends beyond a simple communication mechanism to encompass the secure establishment of a “proof key” that communicates the public key of an OBC to a different origin within the browser by exposing a new browser window function:

```
otherWindow.postKey(message, targetOrigin)
```

This *postKey* call works like the existing *postMessage* call but additional *cert* and *crossCert* parameters are added to the event received by the recipient window’s message handler. The *cert* parameter contains a certificate that is signed by the receiver’s origin-bound key and includes: the sender’s origin, the sender’s OBC public key, the receiver’s origin, and an X509 extension that includes a random nonce. The *crossCert* has the sender and receiver’s roles reversed (*i.e.*, it contains the receiver’s key, signed by the sender’s key) and includes the same random nonce as in *cert*.

These certificates form what is called a cross certification, where the recipient of the certification can establish that the sender’s public key is K_S because K_S has been signed, by the browser, with the receiver’s private key K_R . Additionally, the caller’s public key cross-certifies the receiver’s public key to establish that both keys belong to the same browser.

It’s important to note that the sender does not get to choose the keys used in this cross certification process. Instead, the browser selects the OBCs associated with the origins of the sender and receiver and automatically performs the cross certification using the keys associated with the found OBCs.

Putting it all together The combination of the *PostKey* API and origin-bound certificates can be used to improve upon several federation protocols.

Figure 5 shows the steps required to federate a user’s identity in a generic federation protocol that had been modified to work with the *PostKey* API and OBCs. In step 1 the *relying party* issues a *PostKey* javascript request to the IdP’s iFrame and the IdP receives a cross

certification from the web browser. In step 2, an Authorization Request is issued to the IdP. Since the request is sent over the TLS channel authenticated with K_{IDP} the server associates the incoming request with the user U associated with K_{IDP} . The authorization request contains the cross certification that asserts that K_{RP} and K_{IDP} belong to the same user’s browser so upon user consent, the IdP can respond (in step 3) with a single use Identity Assertion that asserts that K_{RP} is also associated with user U . The IdP’s iFrame then passes the Identity Assertion to the RP’s frame where, in step 4, the Identity Assertion is forwarded to the relying party’s server. The relying party verifies that the Identity Assertion was delivered over a channel authenticated with K_{RP} , has been properly signed by the IdP, and has not been used yet. If this verification succeeds the RP can now associate user U with key K_{RP} by setting a cookie in the user’s browser as shown in step 5.

4.3 Protecting user authentication

We’ve largely considered the initial user-authentication phase, when the user submits his credentials (*e.g.*, username/password) in return for an authenticated session, to be out of scope for this paper. However, we now briefly outline how TLS-OBC can be leveraged in order to secure this tricky phase of the authentication flow.

As a promising direction where TLS-OBC can make a significant impact, we explore the ideas put forth by a recent workshop paper by Czeskis *et al.* [8], where the authors frame authentication in terms of protected and unprotected login. They define unprotected login as an authentication during which all of the submitted credentials are user-supplied and are therefore vulnerable to phishing attacks. For example, these types of logins occur when users first sign in from a new device or after having cleared all browser state (*i.e.*, cleared cookies). The authors observe that to combat the threats to unprotected login, many websites are moving towards protected login, whereby user-supplied credentials are accompanied by supplementary, “unphishable” credentials such as cookies or other similar tokens. For example, websites may set long-lived cookies for users the first time they log in from a new device (an unprotected login), which will not be cleared when a user logs out or his session expires. On subsequent logins, the user’s credentials (*i.e.*, username/password) will be accompanied by the previously set cookie, allowing websites to have some confidence that the login is coming from a user that has already had some interaction with the website rather than a phisher. The authors argue that websites should move all possible authentications to protected login, minimize unprotected login, and then *alert* users when unprotected logins occur. The paper argues that this approach is meaningful

because phishers are not able to produce protected logins and will be forced to initiate unprotected logins instead. Given that unprotected logins should occur rarely for legitimate users, alerting users during an unprotected login will make it significantly harder for password thieves to phish for user credentials.

It's important to note that websites can't fully trust protected logins because they are vulnerable to MITM attacks. However, with TLS-OBC, websites can protect themselves by channel-binding the long-lived cookie that enables the protected login. Combining TLS-OBC with the protected login paradigm allows us to build systems which are resilient to more types of attacks. For example, when describing the attack in Figure 4, we mentioned that attackers could deliver the user cookie, but that would alert the server to the presence of a MITM. We also mentioned that attackers could drop the channel-bound cookie altogether and force the user to re-authenticate, but that this attack was out of scope. However, using TLS-OBC along with the protected/unprotected paradigm, if the attacker forced the user to re-authenticate, the server could force an unprotected login to be initiated and an alert would be sent to the user, notifying him of a possible attack in progress. Hence, channel-bound cookies along with TLS-OBC would protect the user against this type of attack as well.

The careful reader will observe that protecting first logins from new devices (an initial unprotected login) is difficult since the device and server have no pre-established trust. We are currently in the beginning stages of building a system to handle this case and leave further discussion as future work.

5 Implementation

In order to demonstrate the feasibility of TLS origin-bound certificates for channel-binding HTTP cookies, we implemented the extensions discussed in Section 3. The changes made while implementing origin-bound certificates span many disparate systems, but the major modifications were made to OpenSSL, Mozilla's Network Security Services (used in Firefox and Chrome), the Google TLS terminator, and the open-source Chromium browser.

5.1 TLS Extension Support

We added support for TLS origin-bound certificates to OpenSSL and Mozilla's Network Security Stack by implementing the new TLS-OBC extensions, following the appropriate guidelines [5]. We summarize each of these changes below.

NSS Client Modifications Mozilla's Network Se-

curity Stack (NSS) was modified to publish its acceptance of the TLS-OBC extension when issuing a `ClientHello` message to a TLS endpoint. Upon receipt of a `ServerHello` message that demonstrated that the communicating TLS endpoint also understands and accepts the TLS-OBC extension, a new X509 certificate is generated on-the-fly by the browser for use over the negotiated TLS channel. These NSS modifications required 108 modified or added lines across 6 files in the NSS source code.

OpenSSL Server Modifications The OpenSSL TLS server code was modified to publish its acceptance of the TLS-OBC extension in its `ServerHello` message. Furthermore, if during the TLS handshake the client and server agree to use origin bound certificates, the normal client certificate verification is disabled and the OBC verification process is used instead.

The new verification process attempts to establish that the certificate delivered by the client is an OBC rather than a traditional client authentication certificate. The check is performed by confirming that the certificate is self-signed and checking for the presence of the X509 OBC extension. With these two constraints satisfied, the certificate is attached to the TLS session for later use by higher levels of the software stack.

An upstream patch of these changes is pending and has preliminary support from members of the OpenSSL community. The proposed patch requires 316 lines of modification to the OpenSSL source code where most of the changes focus on the TLS handshake and client certificate verification submodules.

5.2 Browser Modifications

In addition to the NSS client modifications discussed above, Chromium's cookie storage infrastructure was adapted to handle the creation and storage of TLS origin-bound certificates. The modifications required to generate the OBCs resulted in a 712 line patch (across 8 files) to the Chromium source code. Storage of OBCs in the existing Chromium cookie infrastructure required an additional 1,164 lines added across 15 files. These changes have been upstreamed as an experimental feature of Chromium since version 16.

6 Performance Evaluation

We have conducted extensive testing of our modifications to TLS and have found them to perform well, even at a significant scale. We report on these results below.

6.1 Chromium TLS-OBC Performance

Experimental methodology In order to demonstrate that the performance impact of adding origin-bound certificates to TLS connections is minimal, we evaluated the performance of TLS-OBCs in the open-source Chromium browser using industry standard benchmarks. All experiments were performed with Chromium version 19.0.1040.0 running on an Ubuntu (version 10.04) Linux system with a 2.0GHz Core 2 Duo CPU and 4GB of RAM.

All tests were performed against the TLS secured version of a Google's home page. During the tests JavaScript was disabled in the browser to minimize the impact of the JavaScript engine on any observed results. Additionally, SPDY connection pooling was disabled, the browser cache was cleared, and all HTTP connections were reset between each measured test run in order to eliminate any saved state that would skew the experimental results. The Chromium benchmark results discussed in section 6.1.1 were gathered with the Chromium benchmarking extension [12] and the HTML5 Navigation Timing [19] JavaScript interface.

6.1.1 Effects on Chromium TLS Connection Setup

We first analyzed the slowdown resulting the TLS-OBC extension for all connections bound for our website's *HTTPS* endpoints. The two use-cases considered by these tests were the first visit, which requires the client-side generation of a fresh origin-bound certificate, and subsequent visits where a cached origin-bound certificate is used instead.

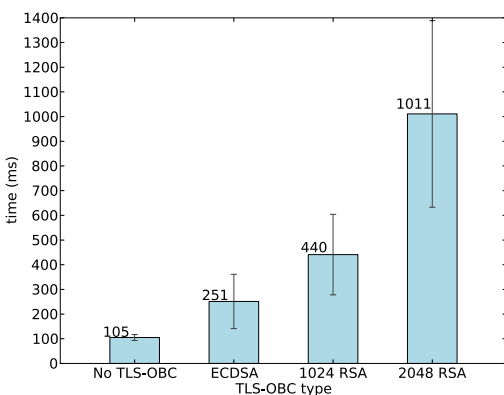


Figure 6: Observed Chromium network latency (ms) with TLS-OBC certificate generation.

The first test shown in Figure 6 shows the total network latency in establishing a connection to our web site and retrieving the homepage on the user's first visit. We

measured the total network latency from the Navigation Timing *fetchStart* event to the *responseEnd* event, encapsulating TLS handshake time as well as network communication latency.

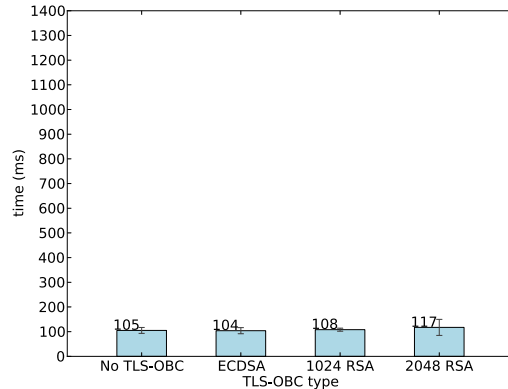


Figure 7: Observed Chromium network latency (ms), TLS-OBC certificate pre-generated.

The results shown in Figure 7 represent subsequent requests to our web site where there is a cache hit for a pre-generated origin-bound certificate. We observed no meaningful impact of the additional *CertificateRequest* and *Certificate* messages required in the TLS handshake on the overall network latency.

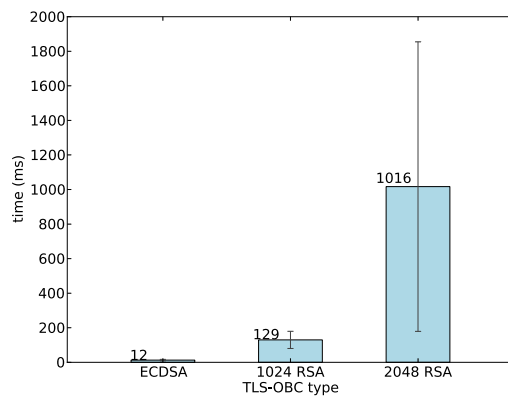


Figure 8: NSS certificate generate times (ms).

The differences between the latencies observed in Figures 6 and 7 imply that origin-bound certificate generation is the contributing factor in the slowdown observed when first visiting an origin that requires a new origin bound certificate. We measured the performance of the origin-bound certificate generation routine, as shown in Figure 8, and found that the certificate generation does seem to be the contributing factor in the higher latencies

seen when first connecting to an origin with an origin-bound certificate.

Client Performance Analysis These observations demonstrate that certificate generation is the main source of slowdown that a client using origin-bound certificates will experience. The selection of public key algorithm has a significant impact on the fresh connection case, and an insignificant impact on subsequent connections. This suggests that production TLS-OBC browsers should speculatively use spare CPU cycles to precompute public/private key pairs, although fresh connections will still need to sign origin-bound certificates, which cannot be done speculatively.

6.2 TLS Terminator Performance

We also measured the impact of TLS-OBC on Google’s high-performance TLS terminator used inside the data-center of our large-scale web service. To test our system, we use a corpus of HTTP requests that model real-world traffic and send that traffic through a TLS terminator to a backend that simulates real-world responses, *i.e.*, it varies both response delays (forcing the TLS terminator to keep state about the HTTP connection in memory for the duration of the backend’s “processing” of the request) as well as response sizes according to a real-world distribution. Mirroring real-world traffic patterns, about 80% of the HTTP requests are sent over resumed TLS sessions, while 20% of requests are sent through freshly-negotiated TLS sessions.

We subjected the TLS terminator to 5 minutes of 3000 requests-per-second TLS-only traffic and periodically measured memory and CPU utilization of the TLS terminator during that period.

We ran four different tests: One without origin-bound certificates, one with a 1024-bit RSA client key pair, one with a 2048-bit RSA client key pair, and one with a 163-bit client key pair on the *sect163k1* elliptic curve (used for ECDSA). We also measure the latency introduced by the TLS terminator for each request (total server-side latency minus backend “processing” time).

Figure 9 shows the impact on memory. Compared to the baseline (without client certificates) of about 1.85GB, the 2048-bit RSA client certs require about 12% more memory, whereas the 1024-bit RSA and ECDSA keys increase the memory consumption by less than 1%.

Figure 10 shows the impact on CPU utilization. Compared to the baseline (without client certificates) of saturating about 4.3 CPU cores, we observed the biggest increase in CPU utilization (of about 7%) in the case of the ECDSA client certificates.

Finally, Figure 11 through Figure 14 show latency histograms. While we see an increase in higher-latency responses when using client-side certificates, the majority

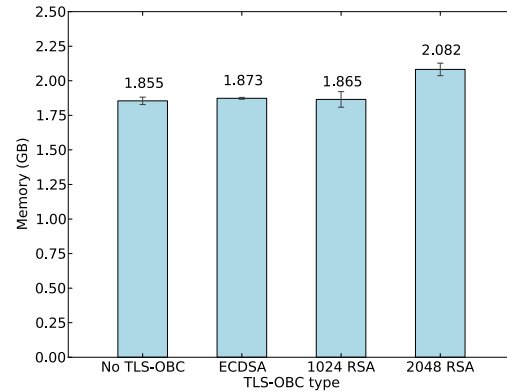


Figure 9: Server-side memory footprint of various client-side key sizes.

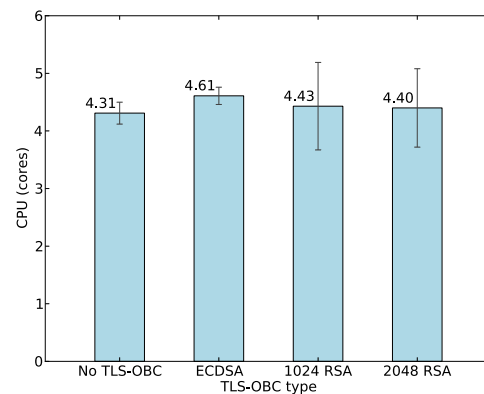


Figure 10: Server-side CPU utilization for various client-side key sizes.

of requests are serviced in under one millisecond in all four cases.

Server Performance Analysis If we cared purely about minimizing the memory and CPU load on our TLS terminator systems, our measurements clearly indicate that we should use 1024-bit RSA. As 1024-bit RSA and 163-bit ECDSA are offer equivalent security [4], however the ECDSA server costs might be worth the client-side benefits.

7 Discussion – Practical Realities

We now discuss a variety of interesting details, challenges, and tensions that we encountered while dealing with the actual nature of how applications are developed and maintained on the web.

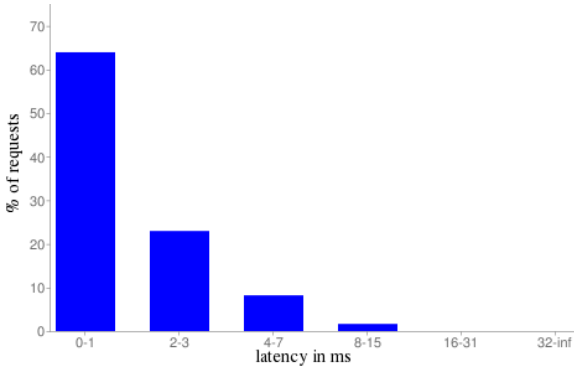


Figure 11: Latency without client certificates.

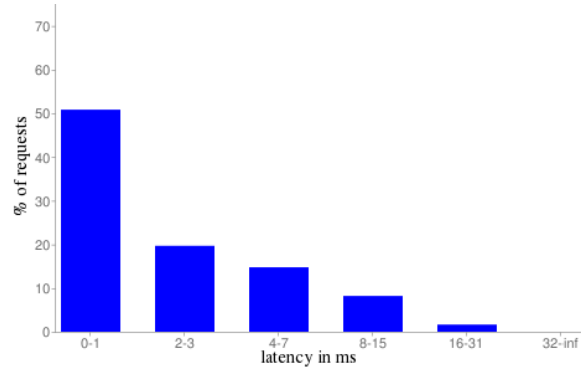


Figure 13: Latency with 2048-bit RSA certificate.

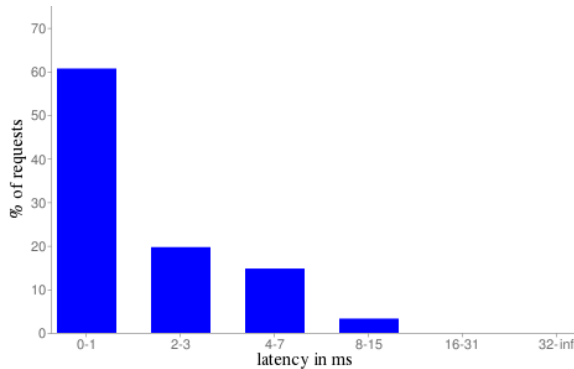


Figure 12: Latency with 1024-bit RSA certificate.

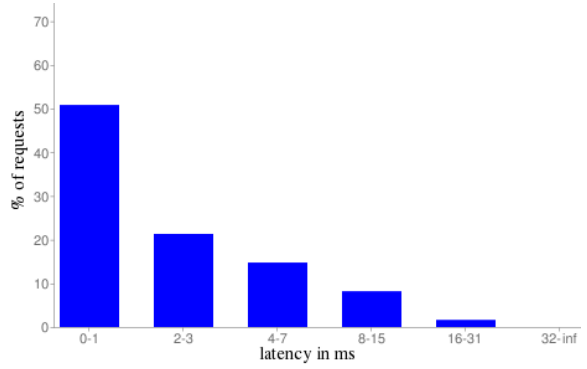


Figure 14: Latency with 163-bit ECDSA certificate.

7.1 Domain Cookies and TLS-OBC

In Section 4 we explained how cookies can be *channel-bound* using TLS-OBC, hardening them against theft. However, this works only as long as the cookie is not set across multiple origins. For example: when a cookie is set by origin *foo.example.com* for domain *example.com*, then clients will send the cookie with requests to (among others) *bar.example.com*. Presumably, however, the client will use a different client certificate when talking to *bar.example.com* than it used when talking to *foo.example.com*. Thus, the channel-binding will break.

Bortz *et al.* [6] make a convincing argument that domain cookies are a poor choice from a security point-of-view, and we agree that in the long run, domain cookies should be replaced with a mix of origin cookies and high-performance federation protocols.

In the meantime, however, we would like to address the issue of domain cookies. In particular, we would like to be able to channel-bind domain cookies just as we’re able to channel-bind origin cookies.

To that end, we are currently considering a “legacy mode” of TLS-OBC, in which the client uses whole domains (based on eTLDs), rather than web origins, as the

granularity for which it uses client-side certificates. Note that this coarser granularity of client certificate scopes does *not* increase the client’s exposure to credential theft. All the protocols presented in this paper maintain their security properties against men-in-the-middle, *etc.* The only difference between origin-scoped client certificates and (more broadly-scoped) domain-scoped client certificates is that in the latter case, related domains (*e.g.*, *foo.example.com* and *bar.example.com*) will be able to see the same OBC for a given browser.

It is also worth noting that even coarse-grained domain-bound client certificates alleviate many of the problems of domain cookies, if those cookies are channel-bound – including additional attacks from the Bortz *et al.* paper.

In balance, we feel that the added protection afforded to widely-used domain cookies outweighs the slight risk of “leaking” client identity across related domains, and are therefore planning to support the above-mentioned “legacy mode” of TLS-OBC.

7.2 Privacy

The TLS specification [9] indicates that both client and server certificates should be sent in the clear during the

handshake process. While OBCs do not bear any information that could be used to identify the user, a single OBC is meant to be reused when setting up subsequent connections to an origin. This certificate reuse enables an eavesdropper to track users by correlating the OBCs used to setup TLS sessions to a particular user and track a users browsing habits across multiple sessions.

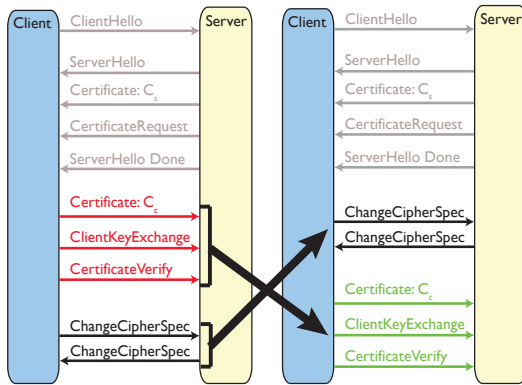


Figure 15: TLS encrypted client certificates

Towards rectifying this issue, we propose to combine TLS-OBC with an encrypted client certificate TLS extension. This extension modifies the ordering of TLS handshake messages so that the client certificate is sent over an encrypted channel rather than in the clear. Figure 15 shows the effect this extension has on TLS message ordering.

7.3 SPDY and TLS-OBC

The SPDY [26] protocol multiplexes several HTTP requests over the same TLS connection, thus achieving higher throughput and lower latency. SPDY has been implemented in Google Chrome for some time, and will be supported in Firefox 11. SPDY always runs over TLS.

One feature of SPDY is *IP pooling*, which allows HTTP sessions from the same client to different web origins to be carried over the same TLS connection if: the web origins in question resolve to the same IP address, *and* the server in the original TLS handshake presented a certificate for all the web origins in question.

For example, if *a.com* and *b.com* resolved to the same IP address, and the server at that IP address presented a valid certificate for *a.com* and *b.com* (presumably through wildcard subject alternative names), then a SPDY client would send requests to *a.com* and *b.com* through the same SPDY (and, hence, TLS) connection.

Remember that with TLS-OBC, the client uses a different client TLS certificate with *a.com* than with *b.com*. This presents a problem. The client needs to be able to present different client certificates for different origins.

In fact, this is not a problem unique to TLS-OBC, but applies to TLS client authentication in general: theoretically speaking, a client might want to use different non-OBC TLS certificates for different origins, even if those origins qualify for SPDY IP pooling.

One solution would be to disallow SPDY IP pooling whenever the client uses a TLS client certificate. Instead, the client would have to open a new SPDY connection to the host to which it wishes to present a client certificate. This solution works well when client certificates are rare: most of the time (when no client certificates are involved), users will benefit from the performance improvements of SPDY IP pooling. When TLS client certificates become ubiquitous, however (as we expect it to be the case through TLS-OBC), most of the time the client *would not* be able to take advantage of SPDY IP pooling if this remained the solution to the problem.

Therefore, SPDY needs to address the problem of client certificates and IP pooling. From version 3 onward, it does this by adding a new CREDENTIAL control frame type. The client sends a CREDENTIAL frame whenever it needs to present a new client certificate to the server (for example, when talking to a new web origin over an IP-pooled SPDY connection). A CREDENTIAL frame allows the client to prove ownership of a public-key certificate without a new TLS handshake by signing a TLS extractor value [21] with the private key corresponding to the public-key certificate.

7.4 Other Designs We Considered

Before settling on TLS-OBC, we considered, and rejected, a number of alternative designs. We share these rejected ideas below to further motivate the choice for TLS-OBC.

Application-Level Crypto API In this design, web client applications would be able to use a crypto API (similar to a PKCS#11 API, but accessible by JavaScript in the browser). JavaScript would be able to generate key pairs, have them certified (or leave the certificates self-signed), use the private key to sign arbitrary data, *etc.*, all without ever touching the private key material itself (again, similar to PKCS#11 or similar crypto APIs).

Every web origin would have separate crypto key containers, meaning that keys generated in one web origin would not be accessible by Javascript running in other web origins. It would be up to individual applications to sign relevant (and application-specific) authentication tokens used in HTTP requests (*e.g.*, special URL query parameters) with keys from that web origin. The application could further design its authentication tokens in such a way that they don't grant ambient authority to a user's account, but rather authorize specific actions on a user's

account (*e.g.*, to send an email whose contents hashes to a certain value, *etc.*).

Such a system would give some protection against a TLS MITM: being unable to mint authentication tokens itself, the attacker could only eavesdrop on a connection. Also, this approach doesn't require changes in the TLS or HTTP layers, and is therefore "standards committee neutral", except for the need for a standardized JavaScript crypto API, which presumably would be useful in other contexts (than authentication) as well.

Note, however, that TLS-OBC with channel-bound cookies provides strictly more protection, preventing men-in-the-middle from eavesdropping. This approach is also vulnerable to XSS attacks and requires applications to be re-written to use these application-level authentication tokens (instead of existing cookies).

We didn't consider the advantages mentioned above strong enough to outweigh the disadvantages of this approach.

Signed HTTP Requests We also explored designs where the client would sign HTTP requests at the HTTP layer. For example, imagine an HTTP request header "X-Request-Signature" that contained a signature of the HTTP request. The key used to sign requests would be client-generated, per-origin, *etc.*, just like for TLS-OBC. Unlike TLS-OBC, this would not require a change in TLS, or HTTP for that matter. This design, however, quickly morphed into a re-implementation of TLS at the HTTP layer. For example, protection against replay attacks leads to timestamps, counters, synchronization issues, and extra round trips. Another example is session renegotiation, questions of renegotiation protocols, and the resulting induced latency.

TLS solves all these issues for us: it protects against replay attacks, allow session renegotiation to be multiplexed with data packages, and many other issues that would have to be addressed at the HTTP layer. We felt that the TLS extension we're proposing was far less complex than the additions to the HTTP layer that would have been necessary to get to comparable security, hence our focus on TLS.

8 Related Work

Origin-bound certificates are closely related to traditional client certificates; we take this opportunity to explain why traditional client certificates don't work in today's web. We also briefly mention various similar efforts to remedy the security issues with authentication on the web, and explain why they stop short of a complete solution.

8.1 Traditional TLS Client Certificates

While TLS *server* authentication is widely used across the web, the *client* authentication aspect of TLS is used much less frequently. Just like TLS server authentication identifies a web server to a client (*i.e.*, browser), TLS client authentication uses public key cryptography to authenticate a client to a web server; this process is an optional part of the TLS handshake.

While effective in small, managed systems such as enterprise networks, the flaws of TLS client authentication begin to emerge as we examine them at web scale:

Bad User Experience One issue that prevents conventional TLS client authentication from becoming the standard for web authentication is the cumbersome, complicated, and onerous interface that a user must wade through in order to use a client certificate. Typically, when web servers request that browsers generate a TLS client certificate, browsers display a dialog where the user must choose the certificate cipher and key length. Even worse, when web servers request that the browser provide a certificate, the user is prompted to select the client certificate to use with the site they are attempting to visit. This "login action" happens during the TLS handshake, before the user can inspect any content of the website (which presumably would help her decide whether or not she wanted to authenticate to the site in the first place).

Layer Confusion Arguably, TLS client authentication puts user identity at the wrong layer in the network stack. An example that reveals this layer confusion is multi-login: Google has implemented a feature in which multiple accounts can be logged into the website at the same time (multiple user identities are encoded in the cookie). This makes it easy to quickly switch between accounts on the site, and even opens up the potential to show a "mashup" of several users' accounts on one page (*e.g.*, show calendars of all the logged-in accounts). With TLS client authentication, the user identity is established at the TLS layer, and is "inherited" from there by the HTTP and application layers. However, client certificates usually contain exactly one user identity, thus forcing the application layer to also only see this one use identity.

Privacy Once a user has obtained a certificate, any site on the web can request TLS client authentication with that certificate. The user can now choose to not be logged in at all, or use the same identity at the new site that they use with other sites on the web. That is a poor choice. Creating different certificates for different sites makes the user experience worse: Now the user is presented with a list of certificates every time they visit a website requiring TLS client authentication.

Portability Since certificates ideally are related to a private key that can't be extracted from the underlying platform, by definition, they can't be moved from one device to another. So any solution that involves TLS client authentication also has to address and solve the user credential portability problem. Potential solutions include re-obtaining certificates from the CA for different devices, extracting private keys (against best security practices) and copying them from one device to another, or cross-certifying certificates from different devices. So far we have not been able to come up with good user interfaces for any of these solutions.

Trusted Computing Base in Datacenters Large datacenters often terminate TLS connections at the datacenter boundary [3], perhaps even using specialized hardware for this relatively expensive part of the connection setup between client and server. If the TLS client certificate is what authenticates the user, then the source of that authentication is lost at the datacenter boundary.

This means that the TLS terminators become part of the trusted computing base – they simply report to the backends who the user is that was authenticated during the TLS handshake. A compromised TLS terminator would in this case essentially become “root” with respect to the applications running in the datacenter.

Contrast this with a cookie-based authentication system, in which the TLS terminator forwards the cookie that the browser sends to the app frontend. In such a system, the cookies are minted and authenticated by the app frontend, and the TLS terminator would not be able to fabricate arbitrary authentic cookies. Put another way, in a cookie-based authentication system a compromised TLS terminator can modify an incoming request before it is delivered to the backend service, but cannot forge a completely new request from an arbitrary user.

In summary, TLS client authentication presents a range of issues, ranging from privacy to usability to deployment problems that make it unsuitable as an authentication mechanism on the web.

8.2 Other Related Efforts

CardSpace Microsoft's CardSpace [13] authentication system attacked two of the problems mentioned so far: First, it replaced passwords with a public-key based protocol, thus eliminating one kind of bearer tokens. Second, it moved user identity from the TLS layer to the application layer.

It allowed users to manage multiple digital identities from a single user interface. CardSpace stored user identities in the form of identity “cards”. When visiting a website that implemented the CardSpace protocol, users could choose which card, and hence which identity, to

use to authenticate with that website. Instead of a username/password pair, a cookie, or a TLS client certificate, CardSpace would authenticate users by sending cryptographic tokens that encoded the user identity. There is no consensus on why CardSpace did not become an industry standard; however, we believe the same complexity that gave CardSpace a wide variety of features, also contributed to its demise by unnecessarily complicating the user interface, interaction, and development models.

CardSpace by itself was also agnostic to the use of bearer tokens in lower layers of the protocol stack once the user was logged in. In this paper we approach the problem from the opposite direction: we build a strong foundation at the TLS layer that allows us to harden other protocols (HTTP, application-specific login, *etc.*), so theoretically origin-bound certificates and CardSpace are more complementary than competing proposals – in particular one could imagine a “channel-bound” CardSpace token that results in a channel-bound cookie (see Section 4). However, we strive to learn from CardSpace's failure in the market and carefully designed our system to not alter the user experience (and burden developers) too much from what users (and developers) are already used to.

BrowserID Mozilla has recently developed a prototype of an authentication mechanism called BrowserID [14], which abstracts identity to the level of email addresses. BrowserID is aimed at the password bearer token, at least for websites that choose to become relying parties to email providers. For those, instead of using a password, users authenticate by providing a cryptographic proof of email ownership. Similarly to CardSpace, the browser maintains a cache of emails (identities) and generates the respective proofs (tokens) for the user. Unlike CardSpace, BrowserID is based on both a simpler model of identity (email addresses vs. a variety of claims) and a simpler implementation platform (JWTs vs. WS-Trust).

BrowserID is complementary to the ideas put forth in this paper. Since it mostly plays at the application layer, it is agnostic to the use of bearer tokens at lower layers (*e.g.*, HTTP cookies). It could easily be adjusted by binding BrowserID identity assertions to the underlying TLS channel if the browser supports origin-bound certificates.

TLS-SA As another approach, Opplinger *et al.* address the disconnect between user authentication and TLS channels in their proposed TLS Session Aware (TLS-SA) User Authentication scheme [17, 18]. TLS-SA is intended to solve the man-in-the-middle (MITM) problem by providing the server side of a TLS connection with the information necessary to determine if a user's credentials have been sent over a different TLS session than the session that the client thought the credentials were

being sent over. However, these protections apply only to the initial user credentials and not to the subsequent bearer tokens. To our knowledge TLS-SA has neither been implemented nor tested on a mass, web scale.

Hardening Cookies Some work has also focused on hardening the information stored in HTTP cookies. For example, Murdoch presented a method for toughening cookies by encoding values not only based on a secret server key, but also on a hash of the user's password [15]. This approach has the benefit of making it harder for attackers to fabricate fake cookies (even if the secret server key has been compromised), but does not protect the user if the cookie is ever stolen.

9 Conclusion

In this paper we presented TLS origin-bound certificates as a new approach to TLS client certificates. TLS-OBCs act as a foundational layer on which the notion of an authenticated channel for the web can be established.

We showed how TLS-OBCs can be used to harden existing HTTP layer authentication mechanisms like cookies, federated login protocols, and user authentication.

We implemented TLS-OBCs as an extension to the OpenSSL and NSS TLS implementations and deployed TLS-OBC to the Chromium open source browser as well as the TLS terminator of a major website.

Finally, we demonstrated that the performance overhead imparted by using TLS-OBC is small in terms of CPU and memory load on the TLS server and observed latency on the TLS client.

We see origin-bound certificates as a first step towards enabling more secure web protocols and applications.

10 Acknowledgements

A great number of individuals have contributed to the work presented in this paper. We would like to thank the team at Google, including Mayank Upadhyay, Adam Langley, Wan-Teh Chang, Matt Mueller, Ryan Hamilton, Diana Smetters, Adam Barth and Warren Zhang for helping us develop the ideas presented in this paper, and for implementing and testing them. Our thanks go out to Ben Adida, Mike Hanson and Brian Smith from Mozilla, as well as the members of the IETF TLS Working Group for sanity-checking and improving our proposals. We would also like to thank Tadayoshi Kohno for his support.

Finally, we would like to thank the anonymous reviewers of our manuscript for helping us make this a better paper.

References

- [1] H. Adkins. An update on attempted man-in-the-middle attacks. <http://googleonlinesecurity.blogspot.com/2011/08/update-on-attempted-man-in-middle.html>, Aug 2011.
- [2] D. Balfanz. TLS Origin-Bound Certificates. <http://tools.ietf.org/html/draft-balfanz-tls-obc-01>, Nov 2011.
- [3] J. Barr. AWS Elastic Load Balancing: Support for SSL Termination. <http://aws.typepad.com/aws/2010/10/elastic-load-balancer-support-for-ssl-termination.html>, Oct 2010.
- [4] S. Blake-Wilson, T. Dierks, and C. Hawk. ECC Cipher Suites for TLS. <http://tools.ietf.org/html/draft-ietf-tls-ecc-01>, March 2001.
- [5] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport layer security (tls) extensions. <http://tools.ietf.org/html/rfc4366>, Apr 2006.
- [6] A. Bortz, A. Barth, and A. Czeskis. Origin cookies: Session integrity for web applications. In *Web 2.0 Security & Privacy*, 2011.
- [7] E. Butler. Firesheep. <http://codebutler.com/firesheep>, 2010.
- [8] A. Czeskis and D. Balfanz. Protected Login. In *Proceedings of the Workshop on Usable Security (at the Financial Cryptography and Data Security Conference)*, March 2012.
- [9] T. Dierks and C. Allen. *The TLS Protocol, Version 1.0*. Internet Engineering Task Force, Jan. 1999. RFC-2246, <ftp://ftp.isi.edu/in-notes/rfc2246.txt>.
- [10] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2 – Client Certificates, 2008. <http://tools.ietf.org/html/rfc5246#section-7.4.6>.
- [11] I. Hickson. HTML5 Web Messaging. <http://dev.w3.org/html5/postmsg/>, Jan 2012.
- [12] J. Hurwich. Chrome benchmarking extension. <http://www.chromium.org/developers/design-documents/extensions/how-the-extension-system-works/chrome-benchmarking-extension>, Sept 2010.
- [13] Microsoft. Introducing windows cardspace, 2006. <http://msdn.microsoft.com/en-us/library/aa480189.aspx>.
- [14] Mozilla. BrowserID, 2012. <https://developer.mozilla.org/en/BrowserID>.
- [15] S. Murdoch. Hardened stateless session cookies. *Security Protocols XVI*, pages 93–101, 2011.
- [16] A. Mushaq. Man in the Browser: Inside the Zeus Trojan, 2010. http://threatpost.com/en_us/blogs/man-browser-inside-zeus-trojan-021910.
- [17] R. Oppliger, R. Hauser, and D. Basin. SSL/TLS session-aware user authentication—or how to effectively thwart the man-in-the-middle. *Computer Communications*, 29(12):2338–2246, 2006.
- [18] R. Opplinger, R. Hauser, and D. Basin. SSL/TLS session-aware user authentication revisited. *Computers & Security*, 27(3-4):64–70, 2008.
- [19] S. Park and D. L. Dill. Verification of cache coherence protocols by aggregation of distributed transactions. *Theory of Computing Systems*, 31(4):355–376, 1998.
- [20] D. Recordon and B. Fitzpatrick. OpenID authentication 1.1. http://openid.net/specs/openid-authentication-1_1.html, May 2008.
- [21] E. Rescorla. Keying Material Exporters for Transport Layer Security (TLS). <http://tools.ietf.org/html/rfc5705>, March 2010.
- [22] J. Rizzo and T. Duong. Beast. <http://vnhacker.blogspot.com/2011/09/beast.html>, Sept 2011.
- [23] N. Sakimura, D. Bradley, B. de Mederiso, M. Jones, and E. Jay. OpenID connect standard 1.0 - draft 07. <http://openid.net/specs/openid-connect-standard-1>
- [24] C. M. Shields and M. M. Toussain. Subterfuge: The MITM Framework. <http://subterfuge.googlecode.com/files/Subterfuge-WhitePaper.pdf>, 2012.
- [25] A. S. Tanenbaum, S. J. Mullender, and R. van Renesse. Using sparse capabilities in a distributed operating system. In *6th International Conference on Distributed Computing Systems*, pages 558–563, Cambridge, Massachusetts, May 1986.
- [26] The Chromium Project. SPDY, 2012. <http://www.chromium.org/spdy>.
- [27] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems (TOCS)*, 12(1):3–32, 1994.

Data Node Encrypted File System: Efficient Secure Deletion for Flash Memory

Joel Reardon, Srdjan Capkun, David Basin
Department of Computer Science, ETH Zurich

Abstract

We propose the Data Node Encrypted File System (DNEFS), which uses on-the-fly encryption and decryption of file system data nodes to efficiently and securely delete data on flash memory systems. DNEFS is a generic modification of existing flash file systems or controllers that enables secure data deletion while preserving the underlying systems' desirable properties: application-independence, fine-grained data access, wear-levelling, and efficiency.

We describe DNEFS both abstractly and in the context of the flash file system UBIFS. We propose UBIFSec, which integrates DNEFS into UBIFS. We implement UBIFSec by extending UBIFS's Linux implementation and we integrate UBIFSec in the Android operating system running on a Google Nexus One smartphone. We show that it is efficient and usable; Android OS and applications (including video and audio playback) run normally on top of UBIFSec. To the best of our knowledge, this work presents the first comprehensive and fully-implemented secure deletion solution that works within the specification of flash memory.

1 Introduction

Flash memory is used near universally in portable devices. However, the way modern systems use flash memory has a serious drawback—it does not guarantee deletion of stored data. To the user, data appears to be deleted from the file system, but in reality it remains accessible after deletion [39]. This problem is particularly relevant for modern smartphones, as they store private data, such as communications, browsing, and location history as well as sensitive business data. The storage of such data on portable devices necessitates guaranteed secure deletion.

Secure deletion is the operation of sanitizing data on a storage medium, so that access to the data is

no longer possible on that storage medium [9]. This is in contrast to standard deletion, where metadata simply indicates that the data's storage location is no longer needed and can be reused. The time between marking data as deleted and its actual (secure) deletion is called the *deletion latency*. We use the term *guaranteed secure deletion* to denote secure deletion with a fixed, (small) finite upper bound on the deletion latency for all data.

On magnetic storage media, secure data deletion is implemented by overwriting a file's content with non-sensitive information [29], or by modifying the file system to automatically overwrite any discarded sector [2]. However, flash memory cannot perform in-place updates of data (i.e., overwrites) [8]; it instead performs erasures on *erase blocks*, which have a larger granularity than read/write operations. A single erase block may store data for different files, so it can only be erased when all the data in the erase block is marked as deleted or when the live data is replicated elsewhere. Moreover, flash memory degrades with each erasure, so frequent erasures shorten the device's lifetime. Therefore, the simplistic solution of erasing any erase block that contains deleted data is too costly with regards to time and device wear [35].

In this work, we present the Data Node Encrypted File System (DNEFS), which securely and efficiently deletes data on flash memory; it requires only a few additional erasures that are evenly distributed over the erase blocks. DNEFS uses on-the-fly encryption and decryption of individual data nodes (the smallest unit of read/write for the file system) and relies on key generation and management to prevent access to deleted data. We design and implement an instance of our solution for the file system UBIFS [14] and call our modification UBIFSec.

UBIFSec has the following attractive properties. It provides a guaranteed upper bound on deletion

latency. It provides fine-grained deletion, also for truncated or overwritten parts of files. It runs efficiently and produces little wear on the flash memory. Finally, it is easy to integrate into UBIFS's existing Linux implementation, and requires no changes to the applications using UBIFS. We deploy UBIFSec on a Google Nexus One smartphone [11] running an Android OS. The system and applications (including video and audio playback) run normally on top of UBIFSec.

Even though DNEFS can be implemented on YAFFS (the file system used on the Android OS), this would have required significant changes to YAFFS. We test DNEFS within UBIFS, which is a supported part of the standard Linux kernel (since version 2.6.27) and which provides interfaces that enable easy integration of DNEFS.

We summarize our contributions as follows. We design DNEFS, a system that enables guaranteed secure data deletion for flash memory—operating within flash memory's specification [26]. We instantiate DNEFS as UBIFSec, analyze its security, and measure its additional battery consumption, throughput, computation time, and flash memory wear to show that it is practical for real-world use. We provide our modification freely to the community [37].

2 Background

Flash Memory. Flash memory is a non-volatile storage medium consisting of an array of electronic components that store information [1]. Flash memory has very small mass and volume, does not incur seek penalties for random access, and requires little energy to operate. As such, portable devices almost exclusively use flash memory.

Flash memory is divided into two levels of granularity. The first level is called *erase blocks*, which are on the order of 128 KiB [11] in size. Each erase block is divided into *pages*, which are on the order of 2 KiB in size. Erase blocks are the unit of erasure, and pages are the unit of read and write operations [8]. One cannot write data to a flash memory page unless that page has been previously *erased*; only the erasure operation performed on an erase block prepares the pages it contains for writing.

Erasing flash memory causes significant physical wear [22]. Each erasure risks turning an erase block into a bad block, which cannot store data. Flash erase blocks tolerate between 10^4 to 10^5 erasures before they become bad blocks. To promote a longer device lifetime, erasures should be evenly levelled over the erase blocks.

MTD Layer. On Linux, flash memory is accessed through the Memory Technology Device (MTD) layer [23]. MTD has the following interface: read and write a page, erase an erase block, check if an erase block is bad, and mark an erase block as bad. Erase blocks are referenced sequentially, and pages are referenced by the erase block number and offset.

Flash File Systems. Several flash memory file systems have been developed at the MTD layer [4, 40]. These file systems are log-structured: a class of file systems that (notably) do not perform in-place updates. A log-structured file system consists of an ordered list of changes from an initial empty state, where each change to the file system is appended to the log's end [34]. Therefore, standard log-structured file systems do not provide secure deletion because new data is only appended.

When a change invalidates an earlier change then the new, valid data is appended and the erase block containing the invalidated data now contains wasted space. Deleting a file, for example, appends a change that indicates the file is deleted. All the deleted file's data nodes remain on the storage medium but they are now invalid and wasting space. A garbage collection mechanism detects and recycles erase blocks with only invalid data; it also copies the remaining valid data to a new location so it may recycle erase blocks mostly filled with invalid data.

Flash Translation Layer. Flash memory is commonly accessed through a Flash Translation Layer (FTL) [1, 15], which is used in USB sticks, SD cards, and solid state drives. FTLs access the raw flash memory directly, but expose a typical hard drive interface that allows any regular file system to be used on the memory. FTLs can either be a hardware controller or implemented in software. An FTL translates logical block addresses to raw physical flash addresses, and internally implements a log-structured file system on the memory [6]. Therefore, like log-structured file systems, FTLs do not provide secure data deletion. In Section 5 we explain how to modify an FTL to use DNEFS to enable efficient secure deletion for any file system mounted on it.

UBI Layer. Unsorted Block Images (UBI) is an abstraction of MTD, where logical erase blocks are transparently mapped to physical erase blocks [10]. UBI's logical mapping implements wear-leveling and bad block detection, allowing UBI file systems to ignore these details. UBI also permits the atomic updating of a logical erase block—the new data is either entirely available or the old data remains.

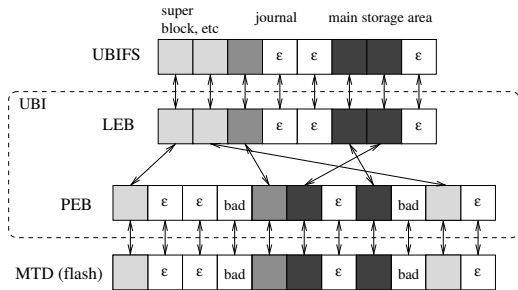


Figure 1: Erase block relationships among MTD, UBI, and UBIFS. Different block shades label different areas of the file system. Empty LEBs are labelled by ϵ and are not mapped to a corresponding PEB by UBI. Similarly, bad PEBs are labelled and not mapped onto by UBI.

UBI exposes the following interface: read and write to a Logical Erase Block (LEB), erase an LEB, and atomically update the contents of an LEB. UBI LEBs neither become bad due to wear, nor should their erasure counts be levelled.

Underlying this interface is an injective partial mapping from LEBs to physical erase blocks (PEBs), where PEBs correspond to erase blocks at the MTD layer. The lower half of Figure 1 illustrates this relationship. Wear monitoring is handled by tracking the erasures at the PEB level, and a transparent remapping of LEBs occurs when necessary. Remapping also occurs when bad blocks are detected. Despite remapping, an LEB’s number remains constant, regardless of its corresponding PEB.

Atomic updates occur by invoking UBI’s update function, passing as parameters the LEB number to update along with a buffer containing the desired contents. An unused and empty PEB is selected and the page-aligned data is then written to it. UBI then updates the LEB’s mapping to the new PEB, and the previous PEB is queued for erasure. This erasure can be done either automatically in the background or immediately with a blocking system call. If the atomic update fails at any time—e.g., because of a power loss—then the mapping is unchanged and the old PEB is not erased.

UBIFS. The UBI file system, UBIFS [14], is designed specifically for UBI, and Figure 1 illustrates UBIFS’s relationship to UBI and MTD. UBIFS divides file data into fixed-sized data nodes. Each data node has a header that stores the data’s inode number and its file offset. This inverse index is used by the garbage collector to determine if the nodes on an erase block are valid or can be discarded.

UBIFS first writes all data in a journal. When this

journal is full, it is committed to the main storage area by logically moving the journal to an empty location and growing the main storage area to encompass the old journal. An index is used to locate data nodes, and this index is also written to the storage medium. At its core, UBIFS is a log-structured file system; in-place updates are not performed. As such, UBIFS does not provide guaranteed secure data deletion.

Adversarial Model. In this work, we model a novel kind of attacker that we name the *peek-a-boo attacker*. This attacker is more powerful than the strong *coercive* attacker considered in other secure deletion works [27, 30]. A coercive attacker can, at any time, compromise both the storage medium containing the data along with any secret keys or passphrases required to access it. The peek-a-boo attacker extends the coercive attacker to also allow the attacker to obtain (“to peek into”) the content of the storage medium at some point(s) in time prior to compromising the storage medium.

Coercive attacks model legal subpoenas that require users to forfeit devices and reveal passwords. Since the time of the attack is arbitrary and therefore unpredictable, no extraordinary sanitization procedure can be performed prior to the compromise time. Since the attacker is given the user’s secret keys, it is insufficient to simply encrypt the storage media [17]. The peek-a-boo attacker models an attacker who additionally gets temporary read-access to the medium (e.g., a hidden virus that is forced to send suicide instructions upon being publicly exposed) and then subsequently performs a coercive attack. It is roughly analogous to forward secrecy in the sense that if a secure deletion scheme is resilient to a peek-a-boo attacker, it prevents recovery of deleted data even if an earlier snapshot of the data from the storage medium is available to the attacker.

Figure 2 shows a timeline of data storage and an adversarial attack. We divide time into discrete intervals called *purging epochs*. At the end of each purging epoch any data marked for deletion is securely deleted (purged). We assume that purging is an atomic operation. The lifetime of a piece of data is then defined as all the purging epochs from the one when it was written to the one when it was deleted. We say that data is *securely deleted* if a peek-a-boo attacker cannot recover the data when performing peek and boo attacks in any purging epochs outside the data’s lifetime.

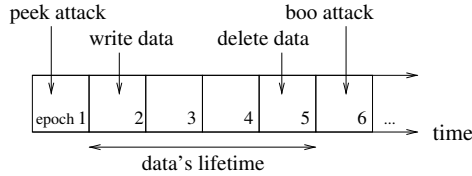


Figure 2: Example timeline for secure deletion. Time is divided into discrete purging epochs. Data is written in epoch 2 and deleted in epoch 5, and the data’s lifetime includes all epochs between these. Here, the peek attack (read access to the entire storage medium) occurs in epoch 1 and the boo attack (full compromise of the storage medium and secret keys/passphrases) in epoch 6. More generally, they can occur in any purging epochs outside the data’s lifetime.

3 DNEFS

In this section we describe our main contribution: a solution for efficient secure deletion for flash memory. We first list our requirements for secure deletion, and afterwards describe our solution.

3.1 Secure Deletion Requirements

We present four requirements for secure deletion solutions. The solution must be sound, fine-grained, efficient, and simple.

Soundness requires that the solution ensures guaranteed secure data deletion against a strong attacker; we use the peek-a-boo attacker defined in Section 2.

Fine-grained requires the solution to securely delete data, however small. This includes overwritten or truncated files, such as data removed from a long-lived database.

The solution must be *efficient* in terms of resource consumption. For flash memory and portable devices, the relevant resources are battery consumption, computation time, storage space, and device lifetime, i.e., minimizing and levelling wear.

Finally, *simplicity* requires that the solution can be easily implemented as part of existing systems. For our purposes, this means that adding secure deletion to existing file systems must be straightforward. We want to minimize the necessary changes to the existing code and isolate the majority of the implementation in new functions and separate data structures. We want the change to be easily audited and analyzed by security-minded professionals. Moreover, we must not remove or limit any existing feature of the underlying file system.

3.2 Our Solution

We now present our secure deletion solution and show how it fulfills the listed requirements.

In the spirit of Boneh and Lipton [3], DNEFS uses encryption to provide secure deletion. It encrypts each individual data node (i.e., the unit of read/write for the file system) with a different key, and then manages the storage, use, and purging of these keys in an efficient and transparent way for both users and applications. Data nodes are encrypted before being written to the storage medium and decrypted after being read; this is all done in-memory. The keys are stored in a reserved area of the file system called the key storage area.

DNEFS works independent of the notion of files; neither file count/size nor access patterns have any influence on the size of the key storage area. The encrypted file data stored on the medium is no different than any reversible encoding applied by the storage medium (e.g., error-correcting codes) because all legitimate access to the data only observes the unencrypted form. This is not an encrypted file system, although in Section 5 we explain that it can be easily extended to one. In our case, encryption is simply a coding technique that we apply immediately before storage to reduce the number of bits required to delete a data node from the data node size to the key size.

Key Storage Area. Our solution uses a small migrating set of erase blocks to store all the data nodes’ keys—this set is called the Key Storage Area (KSA). The KSA is managed separately from the rest of the file system. In particular, it does not behave like a log-structured file system: when a KSA erase block is updated, its contents are written to a new erase block, the logical reference to the KSA block is updated, and the previous version of the KSA erase block is then erased. Thus, except while updating, only one copy of the data in the KSA is available on the storage medium. Our solution therefore requires that the file system or flash controller that it modifies can logically reference the KSA’s erase blocks and erase old KSA erase blocks promptly after writing a new version.

Each data node’s header stores the logical KSA position that contains its decryption key. The erase blocks in the KSA are periodically erased to securely delete any keys that decrypt deleted data. When the file system no longer needs a data node—i.e, it is removed or updated—we mark the data node’s corresponding key in the KSA as deleted. This solution is independent of the notion of files; keys are marked

as deleted whenever a data node is discarded. A key remains marked as deleted until it is removed from the storage medium and its location is replaced with fresh, unused random data, which is then marked as unused.

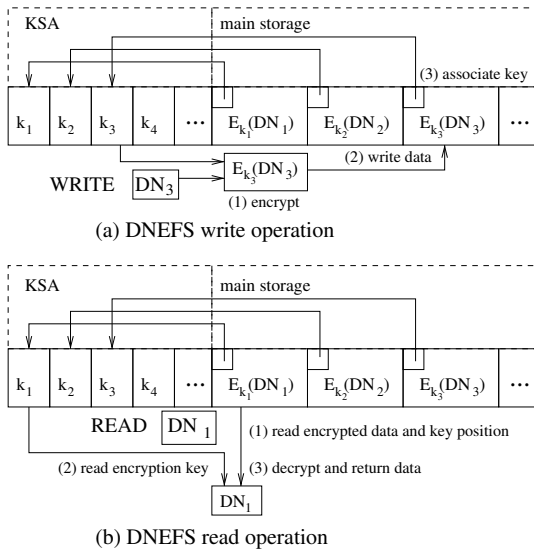


Figure 3: (a) writing a new data node DN_3 : DN_3 is first encrypted with an unused key k_3 and then written to an empty position in the main storage. A reference to the key’s position in the KSA is stored alongside $E_{k_3}(DN_3)$. (b) reading a data node DN_1 : $E_{k_1}(DN_1)$ is first read from the storage medium along with a reference to its key k_1 in the KSA. The key is then read and used to decrypt and return DN_1 .

When a new data node is written to the storage medium, an unused key is selected from the KSA and its position is stored in the data node’s header. DNEFS does this seamlessly, so applications are unaware that their data is being encrypted. Figure 3 illustrates DNEFS’s write and read algorithms.

Purging. Purging is a periodic procedure that securely deletes keys from the KSA. Purging proceeds iteratively over each of the KSA’s erase blocks: a new version of the erase block is prepared where the used keys remain in the same position and all other keys (i.e., unused and deleted keys) are replaced with fresh, unused, cryptographically-appropriate random data from a source of hardware randomness. Such random data is inexpensive and easy to generate, even for resource-constrained devices [38]. Fresh random data is then assigned to new keys as needed. We keep used keys logically-fixed because their corresponding data node has already stored—immutably until an erasure operation—its logical

position. The new version of the block is then written to an arbitrary empty erase block on the storage medium. After completion, all erase blocks containing old versions of the logical KSA erase block are erased, thus securely deleting the unused and deleted keys along with the data nodes they encrypt.

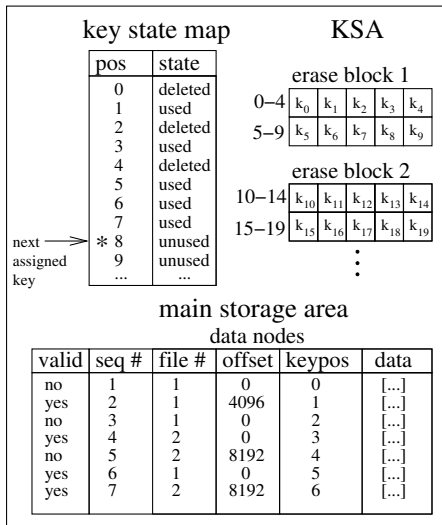
The security of our system necessitates that the storage medium can be properly instructed to erase an erase block. Therefore, for flash memory, DNEFS must be implemented either into the logic of a file system that provides access to the raw flash memory (e.g., UBIFS) or into the logic of the flash controller (e.g., solid state drive). As Swanson et al. [36] observe, any implementation of secure deletion on top of an opaque flash controller cannot guarantee deletion, as its interface for erase block erasure is not security focused and may neglect to delete internally created copies of data due to wear levelling. Our use of UBI bypasses obfuscating controllers and allows direct access to the flash memory.

By only requiring the secure deletion of small densely-packed keys, DNEFS securely deletes all the storage medium’s deleted data while only erasing a small number of KSA erase blocks. Thus, encryption is used to reduce the number of erasures required to achieve secure deletion. This comes at the cost of assuming a computationally-bounded adversary—an information-theoretic adversary could decrypt the encrypted file data. We replace unused keys with new random data to thwart the peek-a-boo attacker: keys are discarded if they are not used to store data in the same deletion epoch as they are generated.

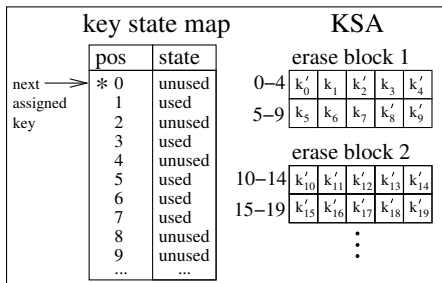
While DNEFS is designed to batch deleted data nodes, thus erasing fewer erase blocks per deleted data node, there is no technical reason that prohibits immediate secure deletion. In particular, files can be marked as sensitive [2] so that purging is triggered whenever a data node for such a file is deleted, resulting in one erase block erasure. Purging can also be triggered by an application, for example after it clears its cache.

If a KSA erase block becomes a bad block while erasing it, it is possible that its contents will remain readable on the storage medium without the ability to remove them [21]. In this case, it is necessary to re-encrypt any data node whose encryption key remains available and to force the garbage collection of those erase blocks on which the data nodes reside.

Key State Map. The *key state map* is an in-memory map that maps key positions to key states $\{unused, used, deleted\}$. Unused keys can be assigned and then marked as used. Used keys are keys that encrypt some valid data node, so they must be



(a) state before purging keys



(b) state after purging keys

Figure 4: Example of a key state map, key storage area, and main storage area during a purging operation. (a) shows the state before and (b) shows the state after purging. Some keys are replaced with new values after purging, corresponding to data nodes that were unused or deleted. The table of data nodes illustrate a log-structured file system, where newer versions of data nodes for the same file/offset invalidate older versions.

preserved to ensure availability of the file system’s data. Deleted keys are keys used to encrypt deleted data—i.e., data nodes that are no longer referenced by the index—and should be purged from the system to achieve secure deletion. Figure 4 shows an example key state map and a KSA before and after a purging operation: unused and deleted keys are replaced with new values and used keys remain on the storage medium.

While mounting, the key state map must be correctly constructed; the procedure for this depends on the file system in which it is integrated. However, log-structured file systems are capable of generating a file system *index* data structure that maps data nodes to their (most recently written) location in

flash memory. We require only that the file system also determines the key location for the data node in the index, and so the state of each key position can be generated by marking these key locations as used and assuming all other locations are deleted.

We define a *correct* key state map as one that has (with high probability) the following three properties: (1) every unused key must not decrypt any data node—either valid or invalid, (2) every used key must have exactly one data node it can decrypt and this data node must be referred to by the index, and (3) every deleted key must not decrypt any data node that is referred to by the index. Observe that an unused key that is marked as deleted will still result in a correct key state map, as it affects neither the security of deleted data nor the availability of used data.

The operation of purging performed on a correct key state map guarantees DNEFS’s soundness: purging securely deletes any key in the KSA marked as deleted; afterwards, every key decrypts at most one valid data node, and every data node referred to by the index can be decrypted. While the encrypted version of the deleted data node still resides in flash memory, our adversary is thereafter unable to obtain the key required to decrypt and thus read the data. A correct key state map also guarantees the integrity of our data during purging, because no key that is used to decrypt valid data will be removed.

We define DNEFS’s purging epoch’s duration (Section 2) as the time between two consecutive purging operations. When a data node is written, it is assigned a key that is currently marked as unused in the current purging epoch. The purging operation’s execution at the purging epochs’ boundaries ensures that all keys currently marked as unused were not available in any previous purging epoch. Therefore, a peek or boo attack that occurs in any prior purging epoch reveals neither the encrypted data node nor its encryption key. When data is deleted, its encryption key is marked as deleted in the current purging epoch. Purging’s execution before the next purging epoch guarantees that key marked as deleted in one epoch is unavailable in the KSA in the next epoch. Therefore, a peek or boo attack that occurs in any later purging epoch may reveal the encrypted data node but not the key. A computationally-bounded peek-a-boo attacker is unable to decrypt the data node, ensuring that the data is not recoverable and therefore securely deleted.

Conclusion. DNEFS provides guaranteed secure deletion against a computationally-bounded peek-a-boo attacker. When an encryption key is securely

deleted, the data it encrypted is then inaccessible, even to the user. All invalid data nodes have their corresponding encryption keys securely deleted during the next purging operation. Purging occurs periodically, so during normal operation the deletion latency *for all* data is bounded by this period. Neither the key nor the data node is available in any purging epoch prior to the one in which it is written, preventing any early peek attacks from obtaining this information.

4 UBIFSec

We now describe UBIFSec: our instantiation of DNEFS for the UBIFS file system. We first give an overview of the aspects of UBIFS relevant for integrating our solution. We then describe UBIFSec and conclude with an experimental validation.

4.1 UBIFS

UBIFS is a log-structured flash file system, where all file system updates occur out of place. UBIFS uses an index to determine which version of data is the most recent. This index is called the Tree Node Cache (TNC), and it is stored both in volatile memory and on the storage medium. The TNC is a B+ search tree [7] that has a small entry for every data node in the file system. When data is appended to the journal, UBIFS updates the TNC to reference its location. UBIFS implements truncations and deletions by appending special non-data nodes to the journal. When the TNC processes these nodes, it finds the range of TNC entries that correspond to the truncated or deleted data nodes and removes them from the tree.

UBIFS uses a commit and replay mechanism to ensure that the file system can be mounted after an unsafe unmounting without scanning the entire device. Commit periodically writes the current TNC to the storage medium, and starts a new empty journal. Replay loads the most recently-stored TNC into memory and chronologically processes the journal entries to update the stale TNC, thus returning the TNC to the state immediately before unmounting.

UBIFS accesses flash memory through UBI's logical interface, which provides two features useful for our purposes. First, UBI allows updates to KSA erase blocks (called KSA LEB's in the context of UBIFSec) using its atomic update feature; after purging, all used keys remain in the same *logical* position, so references to KSA positions remain valid after purging. Second, UBI handles wear-levelling for all the PEBs, including the KSA. This is useful

because erase blocks assigned to the KSA see more frequent erasure; a fixed physical assignment would therefore present wear-levelling concerns.

4.2 UBIFSec Design

UBIFSec is a version of UBIFS that is extended to use DNEFS to provide secure data deletion. UBIFS's data nodes have a size of 4096 bytes, and our solution assigns each of them a distinct 128-bit AES key. AES keys are used in counter mode, which turns AES into a semantically-secure stream cipher [20]. Since each key is only ever used to encrypt a single block of data, we can safely omit the generation and storage of initialization vectors (IVs) and simply start the counter at zero. Therefore, our solution requires about 0.4% of the storage medium's capacity for the KSA, although there exists a tradeoff between KSA size and data node granularity, which we discuss in Section 4.3.

Key Storage Area. The KSA is comprised of a set of LEBs that store random data used as encryption keys. When the file system is created, cryptographically-suitable random data is written from a hardware source of randomness to each of the KSA's LEBs and all the keys are marked as unused. Purging writes new versions of the KSA LEBs using UBI's atomic update feature; immediately after, `ubi_flush` is called to ensure all PEBs containing old versions of the LEB are synchronously erased and the purged keys are inaccessible. This flush feature ensures that any copies of LEBs made as a result of internal wear-levelling are also securely deleted. Figure 5 shows the LEBs and PEBs during a purging operation; KSA block 3 temporarily has two versions stored on the storage medium.

Key State Map. The key state map (Section 3.2) stores the key positions that are unused, used, and deleted. The correctness of the key state map is critical in ensuring the soundness of secure deletion and data integrity. We now describe how the key state map is created and stored in UBIFSec. As an invariant, we require that UBIFSec's key state map is always correct before and after executing a purge. This restriction—instead of requiring correctness at all times after mounting—is to allow writing new data during a purging operation, and to account for the time between marking a key as used and writing the data it encrypts onto the storage medium.

The key state map is stored, used, and updated in volatile memory. Initially, the key state map of a freshly-formatted UBIFSec file system is correct as it

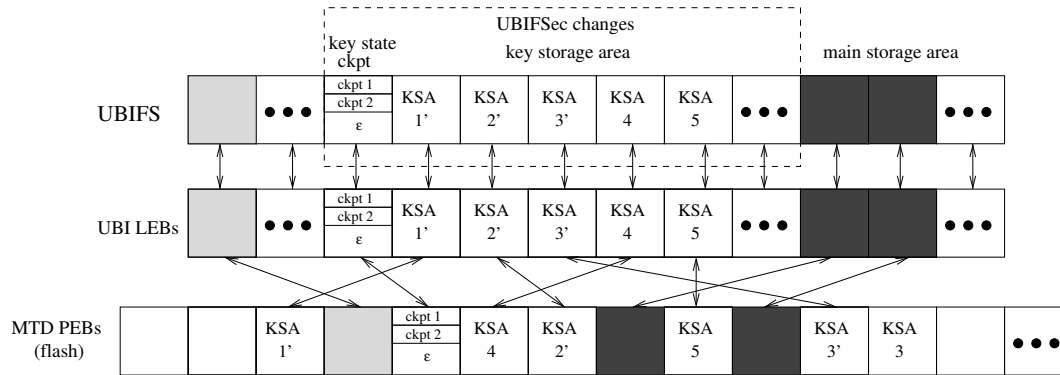


Figure 5: Erase block relationships among MTD, UBI, and UBIFSec, showing the new regions added by UBIFSec (cf. Figure 1). In this example, a purging operation is ongoing—the first three KSA LEBs have been updated and the remaining LEBs still have their old value. In the MTD layer, an old version of KSA 3 is temporarily available.

consists of no data nodes, and every key is fresh random data that is marked as unused. While mounted, UBIFSec performs appropriate key management to ensure that the key state map is always correct when new data is written, deleted, etc. We now show that we can always create a correct key state map when mounting an arbitrary UBIFSec file system.

The key state map is built from a periodic checkpoint combined with a replay of the most recent changes while mounting. We checkpoint the current key state map to the storage medium whenever the KSA is purged. After a purge, every key is either unused or used, and so a checkpoint of this map can be stored using one bit per key—less than 1% of the KSA’s size—which is then compressed. A special LEB is used to store checkpoints, where each new checkpoint is appended; when the erase block is full then the next checkpoint is written at the beginning using an atomic update.

The checkpoint is correct when it is written to the storage medium, and therefore it is correct when it is loaded during mounting if no other changes occurred to the file system. If the file system changed after committing and before unmounting, then UBIFS’s replay mechanism is used to generate the correct key state map: first the checkpoint is loaded, then the replay entries are simulated. Therefore, we always perform purging during regular UBIFS commits; the nodes that are replayed for UBIFS are exactly the ones that must be replayed for UBIFSec. If the stored checkpoint gets corrupted, then a full scan of the valid data nodes rebuilds the correct key state map. A consistency check for the file system also confirms the correctness of the key state map with a full scan.

As it is possible for the storage medium to fail during the commit operation (e.g., due to a loss of

power), we now show that our invariant holds regardless of the condition of unmounting. Purging consists of atomically updating each LEB containing deleted keys and afterwards writing a new checkpoint. UBI’s atomic update feature ensures that any failure before completing the update is equivalent to failing immediately before beginning. Therefore, the following is the complete list of possible failure points: before the first purge, between some purges, after all the purges but before the checkpoint, during the checkpoint, or after the checkpoint but before finishing other UBIFS commit actions.

First, failure can occur before purging the first LEB, which means the KSA is unchanged. When remounting the device, the loaded checkpoint is updated with the replay data, thereby constructing the exact key state map before purging—taken as correct by assumption.

Second, failure can occur after purging one, several, or indeed all of the KSA’s LEBs. When remounting the device, the loaded checkpoint merged with the replay data reflects the state before the first purge, so some purged LEBs contain new unused data while the key state map claims it is a deleted key. As these are cryptographically-suitable random values, with high probability they cannot successfully decrypt any existing valid data node.

Third, failure can occur while writing to the checkpoint LEB. When the checkpoint is written using atomic updates, then failing during the operation is equivalent to failing before it begins (cf. previous case). Incomplete checkpoints are detected and so the previous valid checkpoint is loaded instead. After replaying all the nodes, the key state map is equal to its state immediately before purging the KSA. This means that all entries marked as deleted are actually unused entries, so the invariant holds.

Old ckpt value	Replay's effect	Ckpt value	Value after recovery	Cause	Key's state
unused	nothing	unused	unused	no event	correct
unused	mark used	used	used	key assigned	correct
unused	mark deleted	unused	deleted	key assigned, deleted	correct
used	nothing	used	used	no event	correct
used	mark used	used	used	cannot occur	correct
used	mark deleted	unused	deleted	key deleted	correct

Table 1: Consequences of replaying false information during committing.

Finally, failure can occur after successfully purging the KSA and checkpointing the key state map, but before completing the regular UBIFS commit. In this case, the current key state map correctly reflects the contents of the KSA. When mounting, the replay mechanism incorrectly updates it with the journal entries of the previous iteration. Table 1 shows the full space of possibilities when replaying old changes on the post-purged checkpoint. It shows that it is only possible for an unused key to be erroneously marked as deleted, which still results in a correct key state map.

In summary, the correctness of the key state map before and after a purge is invariant, regardless of when or how the file system was unmounted. This ensures secure deletion's soundness as well as the integrity of the valid data on the storage medium.

Summary. UBIFSec instantiates DNEFS for UBIFS, and so it provides efficient fine-grained guaranteed secure deletion. UBIFSec is efficient in storage space: the overhead for keys is fixed and it needs less than one percent of the total storage medium's capacity. The periodic checkpointing of UBIFSec's key state map ensures that UBIFS's mounting time is not significantly affected by our approach.

Our implementation of UBIFSec is available as a Linux kernel patch for version 3.2.1 [37]. As of the time of writing, we are in the process of integrating UBIFSec into the standard UBIFS distribution.

4.3 Experimental Validation

We have patched an Android Nexus One smartphone's Linux kernel to include UBIFSec and modified the phone to use it as the primary data partition. In this section, we describe experiments with our implementation on both the Android mobile phone and on a simulator.

Our experiments measure our solution's cost: additional battery consumption, wear on the flash memory, and time required to perform file operations. The increase in flash memory wear is mea-

sured using a simulator, and the increase in time is measured on a Google Nexus One smartphone by instrumenting the source code of UBIFS and UBIFSec to measure the time it takes to perform basic file system operations. We further collected timing measurements from the same smartphone running YAFFS: the flash file system currently used on Android phones.

Android Implementation. To test the feasibility of our solution on mobile devices, we ported UBIFSec to the Android OS. The Android OS is based on the Linux kernel and it was straightforward to add support for UBIFS. The source code was already available and we simply applied our patch and configured the kernel compiler to include the UBI device and the UBIFS file system.

Wear Analysis. We measured UBIFSec's wear on the flash memory in two ways: the number of erase cycles that occur on the storage medium, and the distribution of erasures over the erase blocks. To reduce the wear, it is desirable to minimize the number of erasures that are performed, and to evenly spread the erasures over the storage medium's erase blocks.

We instrumented both UBIFS and UBIFSec to measure PEB erasure frequency during use. We varied UBIFSec's purging frequency and computed the resulting erase block allocation rate. This was done by using a low-level control (`ioctl`) to force UBIFS to perform a commit. We also measured the expected number of deleted keys and updated KSA LEBs during purging operation.

We simulated the UBI storage medium based on Nexus One specifications [11]. We varied the period between UBIFSec's purging operation, i.e., the duration of a purging epoch: one of 1, 5, 15, 30, and 60 minutes. We used a discrete event simulator to write files based on the writing behaviour collected from an Android mobile phone [32]. Writing was performed until the file system began garbage collection; thenceforth we took measurements for a week

Purge period	PEB erasures per hour	Updates per KSA purge	KSA updates per hour	Deleted keys per purged LEB	Wear ineq (%)	Lifetime (years)
Standard UBIFS	21.3 ± 3.0	-	-	-	16.6 ± 0.5	841
60 minutes	26.4 ± 1.5	6.8 ± 0.5	6.8 ± 0.5	64.2 ± 9.6	17.9 ± 0.2	679
30 minutes	34.9 ± 3.8	5.1 ± 0.6	9.7 ± 2.0	50.3 ± 9.5	17.8 ± 0.3	512
15 minutes	40.1 ± 3.6	3.7 ± 0.4	14.9 ± 1.6	36.3 ± 8.2	19.0 ± 0.3	447
5 minutes	68.5 ± 4.4	2.6 ± 0.1	30.8 ± 0.7	22.1 ± 4.3	19.2 ± 0.5	262
1 minute	158.6 ± 11.5	1.0 ± 0.1	61.4 ± 4.6	14.1 ± 4.4	20.0 ± 0.2	113

Table 2: Wear analysis for our modified UBIFS file system. The expected lifetime is based on the Google Nexus One phone’s data partition, which has 1571 erase blocks with a (conservative) lifetime estimate of 10^4 erasures.

of simulated time. We averaged the results from four attempts and computed 95% confidence intervals.

To determine if our solution negatively impacts UBI’s wear levelling, we performed the following experiment. Each time UBI unmaps an LEB from a PEB (thus resulting in an erasure) or atomically updates an LEB (also resulting in an erasure), we logged the erased PEB’s number. From this data, we then compute the PEBs’ erasure distribution.

To quantify the success of wear-levelling, we use the Hoover economic wealth inequality indicator—a metric that is independent of the storage medium size and erasure frequency. This metric comes from economics, where it quantifies the unfairness of wealth distributions. It is the simplest measure, corresponding to an appropriately normalized sum of the difference of each measurement to the mean. For our purposes, it is the fraction of erasures that must be reassigned to other erase blocks to obtain completely even wear. Assuming the observations are c_1, \dots, c_n , and $C = \sum_{i=1}^n c_i$, then the inequality measure is $\frac{1}{2} \sum_{i=1}^n \left\| \frac{c_i}{C} - \frac{1}{n} \right\|$.

Table 2 presents the results of our experiment. We see that the rate of block allocations increases as the purging period decreases, with 15 minutes providing a palatable tradeoff between additional wear and timeliness of deletion. The KSA’s update rate is computed as the product of the purging frequency and the average number of KSA LEBs that are updated during a purge. As such, it does not include the additional costs of executing UBIFS commit, which is captured by the disparity in the block allocations per hour. We see that when committing each minute, the additional overhead of committing compared to the updates of KSA blocks becomes significant. While we integrated purging with commit to simplify the implementation, it is possible to separate these operations. Instead, UBIFSec can add purging start and finish nodes as regular (non-data) journal entries. The replay mechanism is then extended to correctly update the key state map while processing these purging nodes.

The expected number of keys deleted per purged KSA LEB decreases sublinearly with the purging period and linearly with the number of purged LEBs. This is because a smaller interval results in fewer expected deletions per interval and fewer deleted keys.

Finally, UBIFSec affects wear-levelling slightly, but not significantly. The unfairness increases with the purging frequency, likely because the set of unallocated PEBs is smaller than the set of allocated PEBs; very frequent updates will cause unallocated PEBs to suffer more erasures. However, the effect is slight. It is certainly the case that the additional block erasures are, for the most part, evenly spread over the device.

Throughput and Battery Analysis A natural concern is that UBIFSec might introduce significant costs that discourage its use. We therefore experimentally evaluated the read/write throughput, battery consumption, and computation time of UBIFSec’s Android implementation (Linux version 2.6.35.7) on a Google Nexus One mobile phone. We compare measurements taken for both Android’s default file system (YAFFS) and for the standard version of UBIFS.

To measure battery consumption over time, we disabled the operating system’s suspension ability, thus allowing computations to occur continuously and indefinitely. This has the unfortunate consequence of maintaining power to the screen of the mobile phone. We first determined the power consumption of the device while remaining idle over the course of two hours starting with an 80% charged battery with a total capacity of 1366 mAh. The result was nearly constant at 121 mA. We subtract this value from all other power consumption measures.

To measure read throughput and battery use, we repeatedly read a large (85 MiB) file; we mounted the drive as read-only and remounted it after each read to ensure that all read caches were cleared. We read the file using `dd`, directing the output to `/dev/null` and recorded the observed throughput.

We began each experiment with an 80% charged battery and ran it for 10 minutes observing constant behaviour. Table 3 presents the results for this experiment. For all filesystems, the additional battery consumption was constant: 39 mA, about one-third of the idle cost. The throughput achieved with that power varied, and so along with our results we compute the amount of data that can be read using 13.7 mAh—1% of the Nexus One’s battery. The write throughput and battery consumption was measured by using `dd` to copy data from `/dev/zero` to a file on the flash file system. Compression was disabled for UBIFS for comparison with YAFFS. When the device was full, the throughput was recorded. We immediately started `dd` to write to the same file, which begins by overwriting it and thus measuring the battery consumption and reduction in throughput imposed by erase block erasure concomitant with writes.

We observe that the use of UBIFSec reduces the throughput for both read and write operations when compared to UBIFS. Some decrease is expected, as the encryption keys must be read from flash while reading and writing. To check if the encryption operations also induce delay, we performed these experiments with a modified UBIFSec that immediately returned zeroed memory when asked to read a key, but otherwise performed all cryptographic operations correctly. The resulting throughput for read and write was identical to UBIFS, suggesting that (for multiple reads) cryptographic operations are easily pipelined into the relatively slower flash memory read/write operations.

Some key caching optimizations can be added to UBIFSec to improve the throughput. Whenever a page of flash memory is read, the entire page can be cached at no additional read cost, allowing efficient sequential access to keys, e.g., for a large file. Long-term use of the file system may reduce its efficiency as gaps between used and unused keys result in new files not being assigned sequential keys. Improved KSA organization can help retain this efficiency.

Write throughput, alternatively, is easily improved with caching. The sequence of keys for data written in the next purging epoch is known at purging time when all these keys are randomly generated and written to the KSA. By using a heuristic on the expected number of keys assigned during a purging epoch, the keys for new data can be kept in memory as well as written to the KSA. Whenever a key is needed, it is taken and removed from this cache while there are still keys available.

Caching keys in memory opens UBIFSec to attacks. We ensure that all memory buffers contain-

	YAFFS	UBIFS	UBIFSec
Read rate (MiB/s)	4.4	3.9	3.0
Power usage (mA)	39	39	39
GiB read per %	5.4	4.8	3.7
Write rate (MiB/s)	2.4	2.1	1.7
Power usage (mA)	30	46	41
GiB written per %	3.8	2.2	2.0

Table 3: I/O throughput and battery consumption for YAFFS, UBIFS, and UBIFSec.

ing keys are overwritten when the key is no longer needed during normal decryption and encryption operations. Caches contain keys for a longer time but are cleared during a purging operation to ensure deleted keys never outlive their deletion purging epoch. Applications storing sensitive data in volatile memory may remain after the data’s deletion and so secure memory deallocation should be provided by the operating system to ensure its unavailability [5].

Timing Analysis. We timed the following file system functions: mounting/unmounting the file system and writing/reading a page. Additionally, we timed the following functions specific to UBIFSec: allocation of the cryptographic context, reading the encryption key, performing an encryption/decryption, and purging a KSA LEB. We collected dozens of measurements for purging, mounting and unmounting, and hundreds of measurements for the other operations (i.e., reading and writing). We controlled the delay caused by our instrumentation by repeating the experiments instead of executing nested measurements, i.e., we timed encryption and writing to a block in separate experiments.

We mounted a partition of the Android’s flash memory first as a standard UBIFS file system and then as UBIFSec file system. We executed a sequence of file I/O operations on the file system. We collected the resulting times and present the 80th percentile measurements in Table 4. Because of UBIFS’s implementation details, the timing results for reading data nodes contain also the time required to read relevant TNC pages (if they are not currently cached) from the storage medium, which is reflected in the increased delay. Because the data node size for YAFFS is half that of UBIFS, we also doubled the read/write measurements for YAFFS for comparison. Finally, the mounting time for YAFFS is for mounting after a safe unmount—for an unsafe unmount, YAFFS requires a full device scan, which takes several orders of magnitude longer.

The results show an increase in the time required for each of the operations. Mounting and unmount-

File system operation	80th percentile execution time (ms)		
	YAFFS	UBIFS	UBIFSec
mount	43	179	236
unmount	44	0.55	0.67
read data node	0.92	2.8	4.0
write data node	1.1	1.3	2.5
prepare cipher	-	-	0.05
read key	-	-	0.38
encrypt	-	-	0.91
decrypt	-	-	0.94
purge one block	-	-	21.2

Table 4: Timing results for various file system functionality on an android mobile phone.

ing the storage medium continues to take a fraction of a second. Reading and writing to a data node increases by a little more than a millisecond, an expected result that reflects the time it takes to read the encryption key from the storage medium and encrypt the data. We also tested for noticeable delay by watching a movie in real time from a UBIFSec-formatted Android phone running the Android OS: the video was 512x288 Windows Media Video 9 DMO; the audio was 96.0 kbit DivX audio v2. The video and audio played as expected on the phone; no observable latency, jitter, or stutter was observed during playback while background processes ran normally.

Each atomic update of an erase block takes about 22 milliseconds. This means that if every KSA LEB is updated, the entire data partition of the Nexus One phone can be purged in less than a fifth of a second. The cost to purge a device grows with its storage medium’s size. The erasure cost for purging can be reduced in a variety of ways: increasing the data node size to use fewer keys, increasing the duration of a purging epoch, or improving the KSA’s organization and key assignment strategy to minimize the number of KSA LEBs that contain deleted keys. The last technique works alongside lazy on-demand purging of KSA LEBs that contain no deleted keys, i.e., only used and unused keys.

Granularity Tradeoff Our solution encrypts each data node with a separate key allowing efficient secure deletion of data from long-lived files, e.g., databases. Other related work instead encrypts each file with a unique key, allowing secure deletion only at the granularity of an entire file [19]. This is well suited for media files, such as digital audio and photographs, which are usually created, read, and deleted in their entirety. However, if the encrypted file should permit random access and modification,

Data node size (flash pages)	KSA size (EBs per GiB)	Copy cost (EBs)
1	64	0
8	8	0.11
64	1	0.98
512	0.125	63.98
4096	0.016	511.98

Table 5: Data node granularity tradeoffs assuming 64 2-KiB pages per erase block.

then one of the following is true: (i) the cipher is used in an ECB-like mode, resulting in a system that is not semantically secure, (ii) the cipher is used in a CBC-like mode where all file modifications require re-encryption of the remainder of the file, (iii) the cipher is used in a CBC-like mode with periodic IVs to facilitate efficient modification, (iv) the cipher is used in counter mode, resulting in all file modifications requiring rewriting the entire file using a new IV to avoid the two-time pad problem [20], or (v) the cipher is used in counter mode with periodic IVs to facilitate efficient modifications.

We observe that the first option is inadequate as a lack of semantic security means that some information about the securely deleted data is still available. The second and fourth options are special cases of the third and fifth options respectively, where the IV granularity is one per file and file modifications are woefully inefficient. Thus, a tradeoff exists between the storage costs of IVs and additional computation for modifications. As the IV granularity decreases to the data node size, the extra storage cost required for IVs is equal to the KSA storage cost for DNEFS’s one key per data node, and the modification cost is simply that of the single data node.

We emphasize that a scheme where IVs were not stored but instead deterministically computed, e.g., using the file offset, would inhibit secure deletion: so long as the file’s encryption key and previous version of the data node were available, the adversary could compute the IV and decrypt the data. Therefore, all IVs for such schemes must be randomly generated, stored, and securely deleted.

Table 5 compares the encryption granularity tradeoff for a flash drive with 64 2-KiB pages per erase block. To compare DNEFS with schemes that encrypt each file separately, simply consider the data node size as equal to the IV granularity or the expected size file size. The KSA size, measured in erase blocks per GiB of storage space, is the amount of storage required for IVs and keys, and is the worst case number of erase blocks that must be erased during each purging operation. The copy cost, also

measured in erase blocks, is the amount of data that must be re-written to the flash storage medium due to a data node modification that affects only one page of flash memory. For example, with a data node size of 1024 KiB and a page size of 2 KiB, the copy cost for a small change to the data node is 1022 KiB. This is measured in erase blocks because the additional writes, once filling an entire erase block, result in an additional erase block erasure, otherwise unnecessary with a smaller data node size.

As we observed earlier, reducing the number of keys required to be read from flash per byte of data improves read and write throughput. From these definitions, along with basic geometry of the flash drive, it is easy to compute the values presented in Table 5. When deploying DNEFS, the administrator can choose a data node size by optimizing for the costs given how frequently small erasures and complete purges are executed.

5 Extensions and Optimizations

Compatibility with FTLs. The most widely-deployed interface for flash memory is the Flash Translation Layer (FTL) [1], which maps logical block device sectors (e.g., a hard drive) to physical flash addresses. While FTLs vary in implementation, many of which are not publicly available, in principle DNEFS can be integrated with FTLs in the following way. All file-system data is encrypted before being written to flash, and decrypted whenever it is read. A key storage area is reserved on the flash memory to store keys, and key positions are assigned to data. The FTL’s in-memory logical remapping of sectors to flash addresses must store alongside a reference to a key location. The FTL mechanism that rebuilds its logical sector to address mapping must also rebuild the corresponding key location. Key locations consist of a *logical* KSA erase block number and the actual offset inside the erase block. Logically-referenced KSA erase blocks are managed by storing metadata in the final page of each KSA erase block. This page is written immediately after successfully writing the KSA block and stores the following information: the logical KSA number so that key references need not be updated after purging, and an epoch number so that the most recent version of the KSA block is known. With this information, the FTL is able to replicate the features of UBI that DNEFS requires.

Generating a correct key state map when mounting is tied to the internal logic of the FTL. Assuming that the map of logical to physical addresses along with the key positions is correctly created, then it

is trivial to iterate over the entries to mark the corresponding keys as used. The unmarked positions are then purged to contain new data. The FTL must also generate cryptographically-secure random data (e.g., with an accelerometer [38]) or be able to receive it from the host. Finally, the file system mounted on the FTL must issue TRIM commands [16] when a sector is deleted, as only the file system has the semantic context to know when a sector is deleted.

Purging Policies. Purging is currently performed after a user-controlled period of time and before unmounting the device. More elaborate policies are definable, where purging occurs once a threshold of deleted keys is passed, ensuring that the amount of exposable data is limited, so the deletion of many files would thus act as a trigger for purging. A low-level control allows user-level applications to trigger a purge, such as an email application that purges the file system after clearing the cache. We can alternatively use a new extended attribute to act a trigger: whenever any data node belonging to a sensitive file is deleted, then DNEFS triggers an immediate purge. This allows users to have confidence that most files are periodically deleted, while sensitive files are promptly deleted.

Securely Deleting Swap. A concern for secure deletion is to securely delete any copies of data made by the operating system. Data that is quite large may be written to a swap file—which may be on the same file system or on a special cache partition. We leave as future work to integrate our solution to a secure deleting cache. (There exist encrypted swap partitions [31], but not one that securely deletes the memory when it is deallocated.) We expect it to be simple to design, as cache data does not need to persist if power is lost; an encryption-based approach can keep all the keys in volatile memory and delete them immediately when they are no longer needed.

Encrypted File System. Our design can be trivially extended to offer a passphrase-protected encrypted file system: we simply encrypt the KSA whenever we write random data, and derive the decryption key from a provided passphrase when mounting. Since, with high probability, each randomly-generated key in the KSA is unique, we can use a block cipher in ECB mode to allow rapid decryption of randomly accessed offsets without storing additional initialization vectors [20].

Encrypted storage media are already quite popular, as they provide confidentiality of stored data

against computationally-bounded non-coercive attackers, e.g., thieves, provided the master secret is unavailable in volatile memory when the attack occurs [13]. It is therefore important to offer our encrypted file system design to avoid users doubly-encrypting their data: first as an encrypted file system and then for secure deletion.

6 Related Work

Secure deletion for magnetic media is a well-researched area. Various solutions exist at different levels of system integration. User-level solutions such as *shred* [29] open a file and overwrite its entire content with insensitive data. This requires that the file system performs in-place updates, otherwise old data still remains. As such, these solutions are inappropriate for flash memory.

Kernel-level secure deletion has been designed and implemented for various popular block-structured file systems [2, 17]. These solutions typically change the file system so that whenever a block is discarded, its content is first sanitized before it is added to the list of free blocks. This ensures that even if a file is truncated or the user forgets to use a secure deletion tool, the data is still sanitized. It is also beneficial for journalled file systems where in-place updates do not immediately occur, so overwriting a file may not actually overwrite the original content. The sanitization of discarded blocks still requires in-place updates, and is therefore inapplicable to flash memory.

The use of encryption to delete data was originally proposed by Boneh and Lipton [3], where they used the convenience of deleting small encryption keys to computationally-delete data from magnetic backup tapes. Peterson et al. [28] used this approach to improve the efficiency of secure deletion in a versioned backup system on a magnetic storage medium. They encrypt each data block with an all-or-nothing cryptographic expansion transformation [33] and colocate the resulting key-sized tags for every version of the same file in storage. They use in-place updates to remove tags, and keys are colocated to reduce the cost of magnetic disk seek times when deleting all versions of a single file. DNEFS also colocates keys separately to improve the efficiency of secure deletion. However, DNEFS is prohibited from performing in-place updates, and our design focuses on minimizing and levelling erasure wear.

Another approach is Perlman's Ephemizer [27], a system that allows communication between parties where messages are securely deleted in the presence of a coercive attacker. Data is encrypted with ephemeral keys that are managed by the eponymous

trusted third party. Each message is given an expiration time at creation, and an appropriate key is generated accordingly. The Ephemizer stores the keys and provides them when necessary to the communicating parties using one-time session keys. When the expiration time passes, it deletes any material required to create the key, thus ensuring secure deletion of the past messages. Perlman's work is a protocol using secure deletion as an assumed primitive offered by the storage medium. Indeed, if this approach is implemented on a flash-based smart card, DNEFS can be used to implement it.

Reardon et al. [32] have shown how to securely delete data from log-structured file systems from user-space—that is, without modifying the hardware or the file system—provided that the user can access the flash directly and is not subjected to disk quota limitations. Their proposal is to fill the storage medium to capacity to ensure that no wasted space remains on the storage medium, thus ensuring the secure deletion of data. This is a costly approach in terms of flash memory wear and execution time, but from user-space it is the only solution possible. They also showed that occupying a large segment of the storage medium with unneeded data improves the expected time deleted data remains on the storage medium, and reduces the amount of space that needs to be filled to guarantee deletion.

Swanson et al. [36] considered verifiable sanitization for solid state drives—flash memory accessed through an opaque FTL. They observed that the manufacturers of the controller are unreliable even when implementing their own advertised sanitization procedure, and that the use of cryptography is insufficient when the ultimate storage location of the cryptographic key cannot be determined from the logical address provided by the FTL. They propose a technique for static sanitization of the entire flash memory—that is, all the storage medium's contained information is securely removed. It works by originally encrypting all the data on the drive before being written. When a sanitize command is issued, first the memory containing the keys is erased, and then every page on the device is written and erased. Our solution focuses on the more typical case of a user wanting to securely delete some sensitive data from their storage media but not wanting to completely remove all data available on the device. Our solution requires access to the raw flash, or a security-aware abstraction such as UBI that offers the `ubi_flush()` function to synchronously remove all previous versions (including copies) of a particular LEB number.

Wei et al. [39] have considered secure deletion on

flash memory accessed through an FTL (cf. Section 2). They propose a technique, called *scrubbing*, which writes zeros over the pages of flash memory without first erasing the block. This sanitizes the data because, in general, flash memory requires an erasure to turn a binary zero to a binary one, but writes turn ones into zeros. Sun et al. [35] propose a hybrid secure deletion method built on Wei et al.'s scheme, where they also optimize the case when there is less data to copy off a block than data to be zero overwritten.

Scrubbing securely deletes data immediately, and no block erasures must be performed. However, it requires programming a page multiple times between erasures, which is not appropriate for flash memory [22]. In general, the pages of an erase block must be programmed sequentially [26] and only once. An option exists to allow multiple programs per page, provided they occur at different positions in the page; multiple overwrites to the same location officially result in undefined behaviour [26]. Flash manufacturers prohibit this due to *program disturb* [21]: bit errors that can be caused in spatially proximate pages while programming flash memory.

Wei et al. performed experiments to quantify the rate at which such errors occur: they showed that they do exist but their frequency varies widely among flash types, a result also confirmed by Grupp et al. [12]. Wei et al. use the term scrub budget to refer to the number of times that the particular model of flash memory has experimentally allowed multiple overwrites without exhibiting a significant risk of data errors. When the scrub budget for an erase block is exceeded, then secure deletion is instead performed by invoking garbage collection: copying all the remaining valid data blocks elsewhere and erasing the block. Wei et al. state that modern densely packed flash memories are unsuitable for their technique as they allow as few as two scrubs per erase block [39]. This raises serious concerns for the future utility of Wei et al.'s approach and highlights the importance of following hardware specifications.

Lee et al. [19] propose secure deletion for YAFFS. They encrypt each file with a different key, store the keys in the file's header, and propose to modify YAFFS to colocate file headers in a fixed area of the storage medium. They achieve secure deletion by erasing the erase block containing the file's header, thus deleting the entire file. More recently, Lee et al. [18] built on this approach by extending it to perform standard data sanitization methods prescribed by government agencies (e.g., NSA [25], DoD [24]) on the erase blocks containing the keys.

We can only compare our approach with theirs in

design, as their approaches were not implemented. First, the approach causes an erase block deletion for every deleted file. This results in rapid wear for devices that create and delete many small cache files. Reardon et al. [32] observed that the Android phone's normal usage created 10,000 such files a day; if each file triggers an erase block erasure, then this solution causes unacceptable wear on the dedicated segment of the flash memory used for file headers. Their proposal encrypts the entire file before writing it to the storage medium with a single key and without mention of the creation or storage of IVs. (See Section 4.3 for more analysis on per-file versus per-data-node encryption.) Our solution differs from theirs by batching deletions and purging based on an interval, by considering the effect on wear levelling, by allowing fine-grained deletions of overwritten and truncated data, and by being fully implemented.

7 Conclusions

DNEFS and its instance UBIFSec are the first feasible solution for efficient secure deletion for flash memory operating within flash memory's specification. It provides *guaranteed secure deletion* against a computationally-bounded peek-a-boo attacker by encrypting each data node with a different key and storing the keys together on the flash storage medium. The erase blocks containing the keys are logically updated to remove old keys, replacing them with fresh random data that can be used as keys for new data. It is *fine-grained* in that parts of files that are overwritten are also securely deleted.

We have implemented UBIFSec and analyzed it experimentally to ensure that it is *efficient*, requiring a small evenly-levelled increase in flash memory wear and little additional computation time. UBIFSec was *easily added* to UBIFS, where cryptographic operations are added seamlessly to UBIFS's read/write data path, and changes to key state are handled by UBIFS's existing index of data nodes.

Acknowledgments

This work was partially supported by the Zurich Information Security Center. It represents the views of the authors. We would like to thank our anonymous reviews for their many helpful comments and Artem Bityutskiy for his help integrating UBIFSec into UBIFS.

References

- [1] BAN, A. Flash file system. US Patent, no. 5404485, 1995.
- [2] BAUER, S., AND PRIYANTHA, N. B. Secure Data Deletion for Linux File Systems. *Usenix Security Symposium* (2001), 153–164.
- [3] BONEH, D., AND LIPTON, R. J. A revocable backup system. In *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6* (Berkeley, CA, USA, 1996), USENIX Association, pp. 91–96.
- [4] CHARLES MANNING. How YAFFS Works. 2010.
- [5] CHOW, J., PFAFF, B., GARFINKEL, T., AND ROSENBLUM, M. Shredding Your Garbage: Reducing Data Lifetime through Secure Deallocation. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14* (Berkeley, CA, USA, 2005), SSYM'05, USENIX Association.
- [6] CHUNG, T.-S., PARK, D.-J., PARK, S., LEE, D.-H., LEE, S.-W., AND SONG, H.-J. A survey of Flash Translation Layer. *Journal of Systems Architecture* 55, 5-6 (2009), 332–343.
- [7] CORMEN, T., LEISERSON, C., AND RIVEST, R. *Introduction to Algorithms*. McGraw Hill, 1998.
- [8] GAL, E., AND TOLEDO, S. Algorithms and Data Structures for Flash Memories. *ACM Computing Surveys* 37 (2005), 138–163.
- [9] GARFINKEL, S., AND SHELAT, A. Remembrance of Data Passed: A Study of Disk Sanitization Practices. *IEEE Security & Privacy* (January 2003), 17–27.
- [10] GLEIXNER, T., HAVERKAMP, F., AND BITYUTSKIY, A. UBI - Unsorted Block Images. 2006.
- [11] GOOGLE, INC. Google Nexus Phone.
- [12] GRUPP, L. M., CAULFIELD, A. M., COBURN, J., SWANSON, S., YAAKOBI, E., SIEGEL, P. H., AND WOLF, J. K. Characterizing flash memory: anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2009), MICRO 42, ACM, pp. 24–33.
- [13] HALDERMAN, J. A., SCHOEN, S. D., HENINGER, N., CLARKSON, W., PAUL, W., CALANDRINO, J. A., FELDMAN, A. J., APPELBAUM, J., AND FELTEN, E. W. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM* 52 (May 2009), 91–98.
- [14] HUNTER, A. A Brief Introduction to the Design of UBIFS. 2008.
- [15] INTEL CORPORATION. Understanding the Flash Translation Layer (FTL) Specification. 1998.
- [16] INTEL CORPORATION. Intel Solid-State Drive Optimizer. 2009.
- [17] JOUKOV, N., PAPAXENOPOULOS, H., AND ZADOK, E. Secure Deletion Myths, Issues, and Solutions. *ACM Workshop on Storage Security and Survivability* (2006), 61–66.
- [18] LEE, B., SON, K., WON, D., AND KIM, S. Secure Data Deletion for USB Flash Memory. *Journal of Information Science and Engineering* 27 (2011), 933–952.
- [19] LEE, J., YI, S., HEO, J., AND PARK, H. An Efficient Secure Deletion Scheme for Flash File Systems. *Journal of Information Science and Engineering* (2010), 27–38.
- [20] MENEZES, A. J., VAN OORSCHOT, P. C., AND VANSTONE, S. A. *Handbook of Applied Cryptography*. CRC Press, 2001.
- [21] MICRON, INC. Design and Use Considerations for NAND Flash Memory Introduction. 2006.
- [22] MICRON TECHNOLOGY, INC. Technical Note: Design and Use Considerations for NAND Flash Memory. 2006.
- [23] Memory Technology Devices (MTD): Subsystem for Linux. 2008.
- [24] National Industrial Security Program Operating Manual. July 1997.
- [25] NSA/CSS Storage Device Declassification Manual. November 2000.
- [26] OPEN NAND FLASH INTERFACE. Open NAND Flash Interface Specification, version 3.0. 2011.
- [27] PERLMAN, R. The Ephemerizer: Making Data Disappear. Tech. rep., Mountain View, CA, USA, 2005.
- [28] PETERSON, Z., BURNS, R., AND HERRING, J. Secure Deletion for a Versioning File System. *USENIX Conference on File and Storage Technologies* (2005).
- [29] PLUMB, C. shred(1) - Linux man page.
- [30] PÖPPER, C., BASIN, D., CAPKUN, S., AND CREMERS, C. Keeping Data Secret under Full Compromise using Porter Devices. In *Computer Security Applications Conference* (2010), pp. 241–250.
- [31] PROVOS, N. Encrypting virtual memory. In *Proceedings of the 9th USENIX Security Symposium* (2000), pp. 35–44.
- [32] REARDON, J., MARFORIO, C., CAPKUN, S., AND BASIN, D. Secure Deletion on Log-structured File Systems. *7th ACM Symposium on Information, Computer and Communications Security* (2012).
- [33] RIVEST, R. L. All-Or-Nothing Encryption and The Package Transform. In *Fast Software Encryption Conference* (1997), Springer Lecture Notes in Computer Science, pp. 210–218.
- [34] ROSENBLUM, M., AND OUSTERHOUT, J. K. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems* 10 (1992), 1–15.
- [35] SUN, K., CHOI, J., LEE, D., AND NOH, S. Models and Design of an Adaptive Hybrid Scheme for Secure Deletion of Data in Consumer Electronics. *IEEE Transactions on Consumer Electronics* 54 (2008), 100–104.
- [36] SWANSON, S., AND WEI, M. SAFE: Fast, Verifiable Sanitization for SSDs. October 2010.
- [37] UBIFSec Patch.
- [38] VORIS, J., SAXENA, N., AND HALEVI, T. Accelerometers and randomness: perfect together. In *Proceedings of the fourth ACM conference on Wireless network security* (New York, NY, USA, 2011), WiSec '11, ACM, pp. 115–126.
- [39] WEI, M., GRUPP, L. M., SPADA, F. M., AND SWANSON, S. Reliably Erasing Data from Flash-Based Solid State Drives. In *Proceedings of the 9th USENIX conference on File and Storage Technologies* (Berkeley, CA, USA, 2011), pp. 105–117.
- [40] WOODHOUSE, D. JFFS: The Journalling Flash File System. In *Ottawa Linux Symposium* (2001).

Throttling Tor Bandwidth Parasites

Rob Jansen
U.S. Naval Research Laboratory
{rob.g.jansen, paul.syverson}@nrl.navy.mil

Nicholas Hopper
University of Minnesota
hopper@cs.umn.edu

Abstract

Tor is vulnerable to network congestion and performance problems due to bulk data transfers. A large fraction of the available network capacity is consumed by a small percentage of Tor users, resulting in severe service degradation for the majority. Bulk users continuously drain relays of excess bandwidth, creating new network bottlenecks and exacerbating the effects of existing ones. While this problem may currently be attributed to rational users utilizing the network, it may also be exploited by a relatively low-resource adversary using similar techniques to contribute to a network denial of service (DoS) attack. Degraded service discourages the use of Tor, affecting both Tor's client diversity and anonymity.

Equipped with mechanisms from communication networks, we design and implement three Tor-specific algorithms that throttle bulk transfers to reduce network congestion and increase network responsiveness. Unlike existing techniques, our algorithms adapt to network dynamics using only information local to a relay. We experiment with full-network deployments of our algorithms under a range of light to heavy network loads. We find that throttling results in significant improvements to web client performance while mitigating the negative effects of bulk transfers. We also analyze how throttling affects anonymity and compare the security of our algorithms under adversarial attack. We find that throttling reduces information leakage compared to unthrottled Tor while improving anonymity against realistic adversaries.

1 Introduction

The Tor [19] anonymity network was developed in an attempt to improve anonymity on the Internet. Onion Routing [23,48] serves as the cornerstone for Tor's overlay network design. Tor *clients* encrypt messages in several "layers" while packaging them into 512-byte packets called *cells*, and send them through a collection of *relays* called a *circuit*. Each relay decrypts its layer and forwards the message to the next relay in the circuit. The last relay forwards the message to the user-specified destination. Each relay can determine only its predecessor and successor in the path from source to destination, preventing any single relay from linking the sender and re-

ceiver. Clients choose their first relay from a small set of *entry guards* [44,59] in order to help defend against passive logging attacks [58]. Traffic analysis is still possible [8,22,28,30,39,42,46,49], but slightly complicated by the fact that each relay simultaneously services multiple circuits.

Tor relays are run by volunteers located throughout the world and service hundreds of thousands of Tor clients [37] with high bandwidth demands. A relay's utility to Tor is dependent on both the bandwidth *capacity* of its host network and the bandwidth *restrictions* imposed by its operator. Although bandwidth donations vary widely, the majority of relays offer less than 100 KiB/s and may become bottlenecks when chosen for a circuit. Bandwidth bottlenecks lead to network congestion and impair client performance.

Bottlenecks are further aggravated by bulk users, which make up roughly five percent of connections and forty percent of the bytes transferred through the network [38]. Bulk traffic increases network-wide congestion and punishes interactive users as they attempt to browse the web and run SSH sessions. Bulk traffic also constitutes a simple denial of service (DoS) attack on the network as a whole: with nothing but a moderate number of bulk clients, an adversary can intentionally significantly degrade the performance of the entire Tor network for most users. This is a malicious attack as opposed to an opportunistic use of resources without regard for the impact on legitimate users, and could be used by censors [16] to discourage use of Tor. Bulk traffic effectively averts potential users from Tor, decreasing both Tor's client diversity and anonymity [10,18].

There are three general approaches to alleviate Tor's performance problems: increase network capacity; optimize resource utilization; and reduce network load.

Increasing Capacity. One approach to reducing bottlenecks and improving performance is to add additional bandwidth to the network from new relays. Previous work has explored recruiting new relays by offering performance incentives to those who contribute [32,41,43]. While these approaches show potential, they have not been deployed due to a lack of understanding of the anonymity and economic implications they would impose on Tor and its users. It is unclear how an incentive

scheme will affect users' anonymity and motivation to contribute: Acquisti *et al.* [6] discuss how differentiating users by performance may reduce anonymity while competition may reduce the sense of community and convince users that contributions are no longer warranted.

New high-bandwidth relays may also be added by the Tor Project [4] or other organizations. While effective at improving network capacity, this approach is a short-term solution that does not scale. As Tor increases speed and bandwidth, it will likely attract more users. More significantly, it will attract more high-bandwidth and BitTorrent users, resulting in a *Tragedy of the Commons* [26] scenario: the bulk users attracted to the faster network will continue to leech the additional bandwidth.

Optimizing Resource Utilization. Another approach to improving performance is to better utilize the available network resources. Tor's path selection algorithm ignores the slowest small fraction of relays while selecting from the remaining relays in proportion to their available bandwidth. The path selection algorithm also ignores circuits with long build times [12], removing the worst of bottlenecks and improving usability. Congestion-aware path selection [57] is another approach that aims to balance load by using opportunistic and active client measurements while building paths. However, low bandwidth relays must still be chosen for circuits to mitigate anonymity problems, meaning there are still a large number of circuits with tight bandwidth bottlenecks.

Tang and Goldberg previously explored modifications to the Tor circuit scheduler in order to prioritize bursty (i.e. web) traffic over bulk traffic using an exponentially-weighted moving average (EWMA) of relayed cells [52]. Early experiments show small improvements at a single relay, but full-network experiments indicate that the new scheduler has an insignificant effect on performance [31]. It is unclear how performance is affected when deployed to the live Tor network. This scheduling approach attempts to shift network load to better utilize the available bandwidth, but does not reduce bottlenecks introduced by the massive amount of bulk traffic currently plaguing Tor.

Reducing Load. All of the previously discussed approaches attempt to increase performance, but none of them directly address or provide adequate defense against performance degradation problems created by bulk traffic clients. In this paper, we address these by adaptively throttling bulk data transfers at the client's entry into the Tor network.

We emphasize that throttling is *fundamentally different* than scheduling, and the distinction is important in the context of the Tor network. Schedulers optimize the utilization of available bandwidth by following policies set by the network engineer, allowing the enforcement of fairness among flows (e.g. max-min fairness [24, 34]

or proportional fairness [35]). However, throttling may *under-utilize* local bandwidth resources by intentionally imposing restrictions on clients' throughput to reduce aggregate network load.

By reducing bulk client throughput in Tor, we effectively reduce the bulk data transfer rate through the network, resulting in *fewer bottlenecks* and a less congested, more responsive Tor network that can better handle the burstiness of web traffic. Tor has recently implemented token buckets, a classic traffic shaping mechanism [55], to statically (non-adaptively) throttle client-to-guard connections at a given rate [17], but currently deployed configurations of Tor do not enable throttling by default. Unfortunately, the throttling algorithm implemented in Tor requires static configuration of throttling parameters: the Tor network must determine network-wide settings that work well and update them as the network changes. Further, it is not possible to automatically tune each relay's throttling configuration with the current algorithm.

Contributions. To the best of our knowledge, we are the first to explore throttling algorithms that *adaptively adjust* to the fluctuations and dynamics of Tor and each relay independently without the need to adjust parameters as the network changes. We also perform the first detailed investigation of the performance and anonymity implications of throttling Tor client connections.

In Section 3, we introduce and test three algorithms that dynamically and adaptively throttle Tor clients using a basic token bucket rate-limiter as the underlying throttling mechanism. Our new adaptive algorithms use local relay information to dynamically select which connections get throttled and to adjust the rate at which those connections are throttled. Adaptively tuned throttling mechanisms are paramount to our algorithm designs in order to avoid the need to re-evaluate parameter choices as network capacity or relay load changes. Our *bit-splitting* algorithm throttles each connection at an adaptively adjusted, but reserved and equal portion of a guard node's bandwidth, our *flagging* algorithm aggressively throttles connections that have historically exceeded the statistically fair throughput, and our *threshold* algorithm throttles connections above a throughput quantile at a rate represented by that quantile.

We implement our algorithms in Tor¹ and test their effectiveness at improving performance in large scale, full-network deployments. Section 4 compares our algorithms to static (non-adaptive) throttling under a varied range of network loads. We find that the effectiveness of static throttling is highly dependent on network load and configuration whereas our adaptive algorithms work well under various loads with no configuration changes or parameter maintenance: web client performance was

¹Software patches for our algorithms have been made publicly available to the community [5].

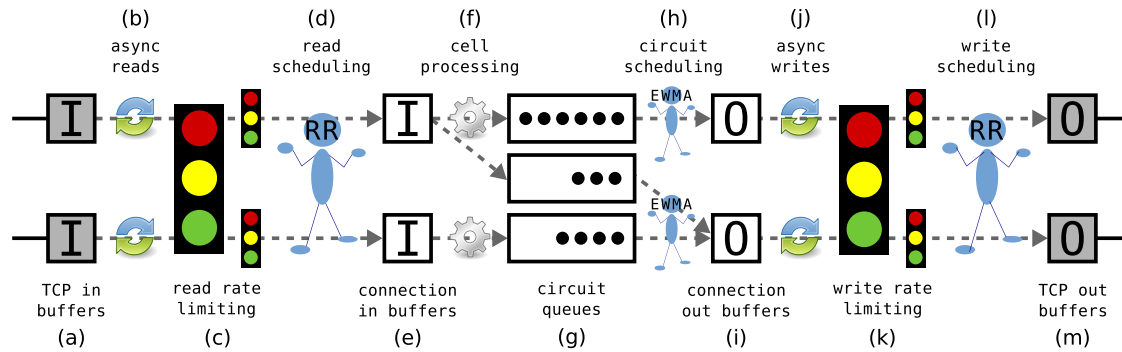


Figure 1: A Tor relay's internal architecture.

improved for every parameter setting we tested. We conclude that throttling is an effective approach to achieve a more responsive network.

Having shown that our adaptive throttling algorithms provide significant performance benefits for web clients and have a profound impact on network responsiveness, Section 5 analyzes the security of our algorithms under adversarial attack. We discuss several realistic attacks on anonymity and compare the information leaked by each algorithm relative to unthrottled Tor. Against intuition, we find that throttling clients reduces information leakage and improves network anonymity while minimizing the false positive impact on honest users.

2 Background

This section discusses Tor's internal architecture, shown in Figure 1, to facilitate an understanding of how internal processes affect client traffic flowing through a Tor relay.

Multiplexed Connections. All relays in Tor communicate using pairwise TCP *connections*, i.e. each relay forms a single TCP connection to each other relay with which it communicates. Since a pair of relays may be communicating data for several *circuits* at once, all circuits between the pair are multiplexed over their single TCP connection. Each circuit may carry traffic for multiple services or *streams* that a user may be accessing. TCP offers reliability, in-order delivery of packets between relays, and potentially unfair kernel-level congestion control when multiplexing connections [47]. The distinction between and interaction of connections, circuits, and streams is important for understanding Tor.

Connection Input. Tor uses libevent [1] to handle input and output to and from kernel TCP buffers. Tor registers sockets that it wants to read with libevent and configures a notification callback function. When data arrives at the kernel TCP input buffer (Figure 1a), libevent learns about the active socket through its polling interface and asynchronously executes the corresponding

callback (Figure 1b). Upon execution, the read callback determines read eligibility using token buckets.

Token buckets are used to rate-limit connections. Tor fills the buckets as defined by configured bandwidth limits in one-second intervals while tokens are removed from the buckets as data is read, although changing that interval to improve performance is currently being explored [53]. There is a global read bucket that limits bandwidth for reading from all connections as well as a separate bucket for throttling on a per-connection basis (Figure 1c). A connection may ignore a read event if either the global bucket or its connection bucket is empty. In practice, the per-connection token buckets are only utilized for edge (non-relay) connections. Per-connection throttling reduces network congestion by penalizing noisy connections, such as bulk transfers, and generally leads to better performance [17].

When a TCP input buffer is eligible for reading, a round-robin (RR) scheduling mechanism is used to read the smaller of 16 KiB and $\frac{1}{8}$ of the global token bucket size per connection (Figure 1d). This limit is imposed in an attempt at fairness so that a single connection can not consume all the global tokens on a single read. However, recent research shows that input/output scheduling leads to unfair resource allocations [54]. The data read from the TCP buffer is placed in a per-connection application input buffer for processing (Figure 1e).

Flow Control. Tor uses an end-to-end flow control algorithm to assist in keeping a steady flow of cells through a circuit. Clients and exit relays constitute the *edges* of a circuit: each are both an ingress and egress point for data traversing the Tor network. Edges track data flow for both circuits and streams using cell counters called *windows*. An ingress edge decrements the corresponding stream and circuit windows when sending cells, stops reading from a stream when its stream window reaches zero, and stops reading from all streams multiplexed over a circuit when the circuit window reaches zero. Windows are incremented and reading resumes upon receipt of SENDME acknowledgment cells from egress edges.

By default, circuit windows are initialized to 1000 cells (500 KiB) and stream windows to 500 cells (250 KiB). Circuit `SENDMEs` are sent to the ingress edge after the egress edge receives 100 cells (50 KiB), allowing the ingress edge to read, package, and forward 100 additional cells. Stream `SENDMEs` are sent after receiving 50 cells (25 KiB) and allow an additional 50 cells. Window sizes can have a significant effect on performance and recent work suggests an algorithm for dynamically computing them [7].

Cell Processing and Queuing. Data is immediately processed as it arrives in connection input buffers (Figure 1f) and each cell is either encrypted or decrypted depending on its direction through the circuit. The cell is then switched onto the circuit corresponding to the next hop and placed into the circuit's first-in-first-out (FIFO) queue (Figure 1g). Cells wait in circuit queues until the circuit scheduler selects them for writing.

Scheduling. When there is space available in a connection's output buffer, a relay decides which of several multiplexed circuits to choose for writing. Although historically this was done using round-robin, a new exponentially-weighted moving average (EWMA) scheduler was recently introduced into Tor [52] and is currently used by default (Figure 1h). EWMA records the number of packets it schedules for each circuit, exponentially decaying packet counts over time. The scheduler writes one cell at a time chosen from the circuit with the lowest packet count and then updates the count. The decay means packets sent more recently have a higher influence on the count while bursty traffic does not significantly affect scheduling priorities.

Connection Output. A cell that has been chosen and written to a connection output buffer (Figure 1i) causes an activation of the write event registered with `libevent` for that connection. Once `libevent` determines the TCP socket can be written, the write callback is asynchronously executed (Figure 1j). Similar to connection input, the relay checks both the global write bucket and per-connection write bucket for tokens. If the buckets are not empty, the connection is eligible for writing (Figure 1k) and again is allowed to write the smaller of 16 KiB and $\frac{1}{8}$ of the global token bucket size per connection (Figure 1l). The data is written to the kernel-level TCP buffer (Figure 1m) and sent to the next hop.

3 Throttling Client Connections

Client performance in Tor depends heavily on the traffic patterns of others in the system. A small number of clients performing bulk transfers in Tor are the source of a large fraction of total network traffic [38]. The overwhelming load these clients place on the network increases congestion and creates additional bottlenecks,

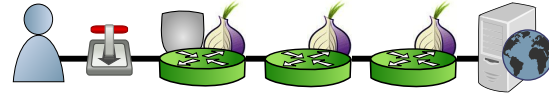


Figure 2: Throttling occurs at the connection between the client and guard to capture all streams to various destinations.

causing interactive applications, such as instant messaging and remote SSH sessions, to lose responsiveness.

This section explores client throttling as a mechanism to prevent bulk clients from overwhelming the network. Although a relay may have enough bandwidth to handle all traffic locally, bulk clients that continue producing additional traffic cause bottlenecks at other low-capacity relays. The faster a bulk downloader gets its data, the faster it will pull more into the network. Throttling bulk and other high-traffic clients prevents them from pushing or pulling too much data into the network too fast, reducing these bottlenecks and improving performance for the majority of users. Therefore, interactive applications and Tor in general will become much more usable, attracting new users who improve client diversity and anonymity.

We emphasize that throttling algorithms are not a replacement for congestion control or scheduling algorithms, although each approach may cooperate to achieve a common goal. Scheduling algorithms are used to *manage the utilization of bandwidth*, throttling algorithms *reduce the aggregate network load*, and congestion control algorithms attempt to do both. The distinction between congestion control and throttling algorithms is subtle but important: congestion control reduces *circuit load* while attempting to maximize network utilization, whereas throttling reduces *network load* in an attempt to improve circuit performance by explicitly under-utilizing connections to bulk clients using too many resources. Each approach may independently affect performance, and they may be combined to improve the network.

3.1 Static Throttling

Recently, Tor introduced the functionality to allow entry guards to throttle connections to clients [17] (see Figure 2). This client-to-guard connection is targeted because all client traffic (using this guard) will flow over this connection regardless of the number of streams or the destination associated with each.² The implementation uses a token bucket for each connection in addition to the global token bucket that already limits the total amount of bandwidth used by a relay. The size of the per-connection token buckets can be specified using the `PerConnBWBurst` configuration option, and the bucket refill rate can be specified by configuring the `PerConnBWRate`. The configured throttling rate en-

²This work does not consider modified Tor clients.

sure that all client-to-guard connections are throttled to the specified long-term-average throughput while the configured burst allows deviations from the throttling rate to account for bursty traffic. The configuration options provide a static throttling mechanism: Tor will throttle all connections using these values until directed otherwise. Note that Tor does not enable or configure static throttling by default.

While static throttling is simple, it has two main drawbacks. First, static throttling requires constant monitoring and measurements of the Tor network to determine which configurations work well and which do not in order to be effective. We have found that there are many configurations of the algorithm that cause no change in performance, and worse, there are configurations that harm performance for interactive applications [33]. This is the opposite of what throttling is attempting to achieve. Second, it is not possible under the current algorithm to auto-tune the throttling parameters for each Tor relay. Configurations that appear to work well for the network as a whole might not necessarily be tuned for a given relay (we will show that this is indeed the case in Section 4). Each relay has very different capabilities and load patterns, and therefore may require different throttling configurations to be most useful.

3.2 Adaptive Throttling

Given the drawbacks of static throttling, we now explore and present three new algorithms that adaptively adjust throttling parameters according to local relay information. This section details our algorithms while Section 4 explores their effect on client performance and Section 5 analyzes throttling implications for anonymity.

There are two main issues to consider when designing a client throttling algorithm: *which connections* to throttle and at *what rate* to throttle them. The approach discussed above in Section 3.1 throttles *all* client connections at the *statically* specified rate. Each of our three algorithms below answers these questions *adaptively* by considering information local to each relay. Note that our algorithms dynamically adjust the `PerConnBWRate` while keeping a constant `PerConnBWBurst`.³

Bit-splitting. A simple approach to adaptive throttling is to split a guard's bandwidth equally among all active client connections and throttle them all at this *fair split rate*. The `PerConnBWRate` will therefore be adjusted as new connections are created or old connections are destroyed: more connections will result in lower rates. No connection will be able to use more than its allot-

³Our experiments [33] indicate that a 2 MiB burst is ideal as it allows directory requests to be downloaded unthrottled during bootstrapping while also throttling bulk traffic relatively quickly. The burst may need to be increased if the directory information grows beyond 2 MiB.

Algorithm 1 Throttling clients by splitting bits.

```
1:  $B \leftarrow \text{getRelayBandwidth}()$ 
2:  $L \leftarrow \text{getConnectionList}()$ 
3:  $N \leftarrow L.\text{length}()$ 
4: if  $N > 0$  then
5:    $\text{splitRate} \leftarrow \frac{B}{N}$ 
6:   for  $i \leftarrow 1$  to  $N$  do
7:     if  $L[i].\text{isClientConnection}()$  then
8:        $L[i].\text{throttleRate} \leftarrow \text{splitRate}$ 
9:     end if
10:  end for
11: end if
```

ted share of bandwidth unless it has unused tokens in its bucket. Inspired by Quality of Service (QoS) work from communication networks [11, 50, 60], bit-splitting will prevent bulk clients from unfairly consuming bandwidth and ensure that there is a minimum “reserved” bandwidth for clients of all types.

Note that Internet Service Providers employ similar techniques to throttle their customers, however, their client base is much less dynamic than the connections an entry guard handles. Therefore, our adaptive approach is more suitable to Tor. We include this algorithm in our analysis of throttling to determine what is possible with such a simple approach.

Flagging Unfair Clients. The bit-splitting algorithm focuses on adjusting the throttle rate and applying this to all client connections. Our next algorithm takes the opposite approach: configure a static throttling rate and adjust which connections get throttled. The intuition behind this approach is that if we can properly identify the connections that use too much bandwidth, we can throttle them in order to maximize the benefit we gain per throttled connection. Therefore, our flagging algorithm attempts to classify and throttle bulk traffic while it avoids throttling web clients.

Since deep packet inspection is not desirable for privacy reasons, and is not possible on encrypted Tor traffic, we instead draw upon existing *statistical fingerprinting* classification techniques [14, 29, 36] that classify traffic solely on its statistical properties. When designing the flagging algorithm, we recognize that Tor already contains a statistical throughput measure for scheduling traffic on *circuits* using an exponentially-weighted moving average (EWMA) of recently sent cells [52]. We can use the same statistical measure on client *connections* to classify and throttle bulk traffic.

The flagging algorithm, shown in Algorithm 2, requires that each guard keeps an EWMA of the number of recently sent cells per client connection. The per-connection cell EWMA is computed in much the same way as the per-circuit cell EWMA: whenever the cir-

Algorithm 2 Throttling clients by flagging bulk connections, considering a moving average of throughput.

Require: $flagRate, \mathcal{P}, \mathcal{H}$

- 1: $B \leftarrow getRelayBandwidth()$
- 2: $L \leftarrow getConnectionList()$
- 3: $N \leftarrow L.length()$
- 4: $\mathcal{M} \leftarrow getMetaEWMA()$
- 5: **if** $N > 0$ **then**
- 6: $splitRate \leftarrow \frac{B}{N}$
- 7: $\mathcal{M} \leftarrow \mathcal{M}.increment(\mathcal{H}, splitRate)$
- 8: **for** $i \leftarrow 1$ to N **do**
- 9: **if** $L[i].isClientConnection()$ **then**
- 10: **if** $L[i].EWMA > \mathcal{M}$ **then**
- 11: $L[i].flag \leftarrow True$
- 12: $L[i].throttleRate \leftarrow flagRate$
- 13: **else if** $L[i].flag = True \wedge$
 $L[i].EWMA < \mathcal{P} \cdot \mathcal{M}$ **then**
- 14: $L[i].flag \leftarrow False$
- 15: $L[i].throttleRate \leftarrow infinity$
- 16: **end if**
- 17: **end if**
- 18: **end for**
- 19: **end if**

cuit’s cell counter is incremented, so is the cell counter of the connection to which that circuit belongs. Note that clients can not affect others’ per-connection EWMA since all of a client’s circuits are multiplexed over a single throttled guard-to-client connection.⁴ The per-connection EWMA is enabled and configured independently of its circuit counterpart.

We rely on the observation that bulk connections will have higher EWMA values than web connections since bulk clients are steadily transferring data while web clients “think” between each page download. Using this to our advantage, we can flag connections as containing bulk traffic as follows. Each relay keeps a single separate meta-EWMA \mathcal{M} of cells transferred. \mathcal{M} is adjusted by calculating the fair bandwidth split rate as in the bit-splitting algorithm, and tracking its EWMA over time. \mathcal{M} does not correspond with any real traffic, but represents the upper bound of a connection-level EWMA if a connection were continuously sending only its fair share of traffic through the relay. Any connection whose EWMA exceeds \mathcal{M} is flagged as containing bulk traffic and penalized by being throttled.

There are three main parameters for the algorithm. As mentioned above, a per-connection half-life \mathcal{H} allows configuration of the connection-level half-life independent of that used for circuit scheduling. \mathcal{H} affects how

⁴The same is not true for the unthrottled connections between relays since each of them contain several circuits and each circuit may belong to a different client (see Section 2).

Algorithm 3 Throttling clients considering the loudest threshold of connections.

Require: $\mathcal{T}, \mathcal{R}, \mathcal{F}$

- 1: $L \leftarrow getClientConnectionList()$
- 2: $N \leftarrow L.length()$
- 3: **if** $N > 0$ **then**
- 4: $selectIndex \leftarrow floor(\mathcal{T} \cdot N)$
- 5: $L \leftarrow reverseSortEWMA(L)$
- 6: $thresholdRate \leftarrow L[selectIndex].$
 $getMeanThroughput(\mathcal{R})$
- 7: **if** $thresholdRate < \mathcal{F}$ **then**
- 8: $thresholdRate \leftarrow \mathcal{F}$
- 9: **end if**
- 10: **for** $i \leftarrow 1$ to N **do**
- 11: **if** $i \leq selectIndex$ **then**
- 12: $L[i].throttleRate \leftarrow thresholdRate$
- 13: **else**
- 14: $L[i].throttleRate \leftarrow infinity$
- 15: **end if**
- 16: **end for**
- 17: **end if**

long the algorithm remembers the amount of data a connection has transferred, and has precisely the same meaning as the circuit priority half-life [52]. Larger half-life values increase the ability to differentiate bulk from web connections while smaller half-life values make the algorithm more immediately reactive to throttling bulk connections. We would like to allow for a specification of the length of each penalty once a connection is flagged in order to recover and stop throttling connections that may have been incorrectly flagged. Therefore, we introduce a penalty fraction parameter \mathcal{P} that affects how long each connection remains in a flagged and throttled state. If a connection’s cell count EWMA falls below $\mathcal{P} \cdot \mathcal{M}$, its flag is removed and the connection is no longer throttled. Finally, the rate at which each flagged connection is throttled, i.e. the `FlagRate`, is statically defined and is not adjusted by the algorithm.

Note that the flagging parameters need only be set based on system-wide policy and generally do not require independent relay tuning, but provides the flexibility to allow individual relay operators to deviate from system policy if they desire.

Throttling Using Thresholds. Recall the two main issues a throttling algorithm must address: selecting *which connections* to throttle and the *rate* at which to throttle them. Our bit-splitting algorithm explored adaptively adjusting the throttle rate and applying this to all connections while our flagging algorithm explored statically configuring a throttle rate and adaptively selecting the throttled connections. We now describe our final algorithm which attempts to adaptively address both issues.

The threshold algorithm also makes use of a connection-level cell EWMA, which is computed as described above for the flagging algorithm. However, EWMA is used here to sort connections by the loudest to quietest. We then select and throttle the loudest fraction \mathcal{T} of connections, where \mathcal{T} is a configurable threshold. For example, setting \mathcal{T} to 0.1 means the loudest ten percent of client connections will be throttled. The selection is adaptive since the EWMA changes over time according to each connection’s bandwidth usage.

We have adaptively selected which connections to throttle and now must determine a throttle rate. To do this, we require that each connection tracks its throughput over time. We choose the average throughput rate of the connection with the minimum EWMA from the set of connections being throttled. For example, when $\mathcal{T} = 0.1$ and there are 100 client connections sorted from loudest to quietest, the chosen throttle rate is the average throughput of the tenth connection. Each of first ten connections is then throttled at this rate. In our prototype, we approximate the throughput rate as the average number of bytes transferred over the last \mathcal{R} seconds, where \mathcal{R} is configurable. \mathcal{R} represents the period of time between which the algorithm re-selects the throttled connections, adjusts the throttle rates, and resets each connection’s throughput counters.

There is one caveat to the algorithm as described above. In our experiments in Section 4, we noticed that occasionally the throttle rate chosen by the threshold algorithm was zero. This would happen if the mean throughput of the threshold connection (line 6 in Algorithm 3) did not send data over the last \mathcal{R} seconds. To prevent a throttle rate of zero, we added a parameter to statically configure a throttle rate floor \mathcal{F} so that no connection would ever be throttled below \mathcal{F} . Algorithm 3 details threshold adaptive throttling.

4 Experiments

In this section we explore the performance benefits possible with each throttling algorithm specified in Section 3. We perform experiments with Shadow [2, 31], an accurate and efficient discrete event simulator that runs real Tor code over a simulated network. Shadow allows us to run an entire Tor network on a single machine and configure characteristics such as network latency, bandwidth, and topology. Since Shadow runs real Tor, it accurately characterizes application behavior and allows us to focus on experimental comparison of our algorithms. A direct comparison between Tor and Shadow-Tor performance is presented in [31].

Experimental Setup. Using Shadow, we configure a private Tor network with 200 HTTP servers, 950 Tor web clients, 50 Tor bulk clients, and 50 Tor relays. The dis-

tribution of clients in our experiments approximates that found by McCoy *et al.* [38]. All of our nodes run inside the Shadow simulation environment.

In our experiments, each client node runs Tor in client-only mode as well as an HTTP client application configured to download over Tor’s SOCKS proxy available on the local interface. Each web client downloads a 320 KiB file⁵ from a randomly selected one of our HTTP servers, and pauses for a length of time drawn from the UNC “think time” data set [27] before downloading the next file. Each bulk client repeatedly downloads a 5 MiB file from a randomly selected HTTP server without pausing. Clients track the time to the first and the last byte of the download as indications of network responsiveness and overall performance.

Tor relays are configured with bandwidth parameters according to a Tor network consensus document.⁶ We configure our network topology and latency between nodes according to the geographical distribution of relays and pairwise PlanetLab node ping times. Our simulated network mirrors a previously published Tor network model [31] that has been compared to and shown to closely approximate the load of the live Tor network [3].

We focus on the time to the first data byte for web clients as a measure of network responsiveness, and the time to the last data byte—the download time—for both web and bulk clients as a measure of overall performance. In our results, “vanilla” represents unmodified Tor using a round-robin circuit scheduler and no throttling—the default settings in the Tor software—and can be used to compare relative performance between experiments. Each experiment uses network-wide deployments of each configuration. To further reduce random variances, we ran all configurations five times each. Therefore, every curve on every CDF shows the cumulative results of five experiments.

Results. Our results focus on the algorithmic configurations that we found to maximize web client performance [33] while we show how the algorithms perform when the network load varies from light (25 bulk clients) to medium (50 bulk clients) to heavy (100 bulk clients). The experimental setup is otherwise unmodified from the model described above. Running the algorithms under various loads allows us to highlight the unique and novel features each provides.

Figure 3 shows client performance for our algorithms. The time to first byte indicates network responsiveness for web clients while the download time indicates overall client performance for web and bulk clients. Client performance is shown for the lightly loaded network in Figures 3a–3c, the normally loaded network in Figures 3d–3f, and the heavily loaded network in Figures 3g–3i.

⁵The average webpage size reported by Google web metrics [45].

⁶Retrieved on 2011-04-27 and valid from 03-06:00:00

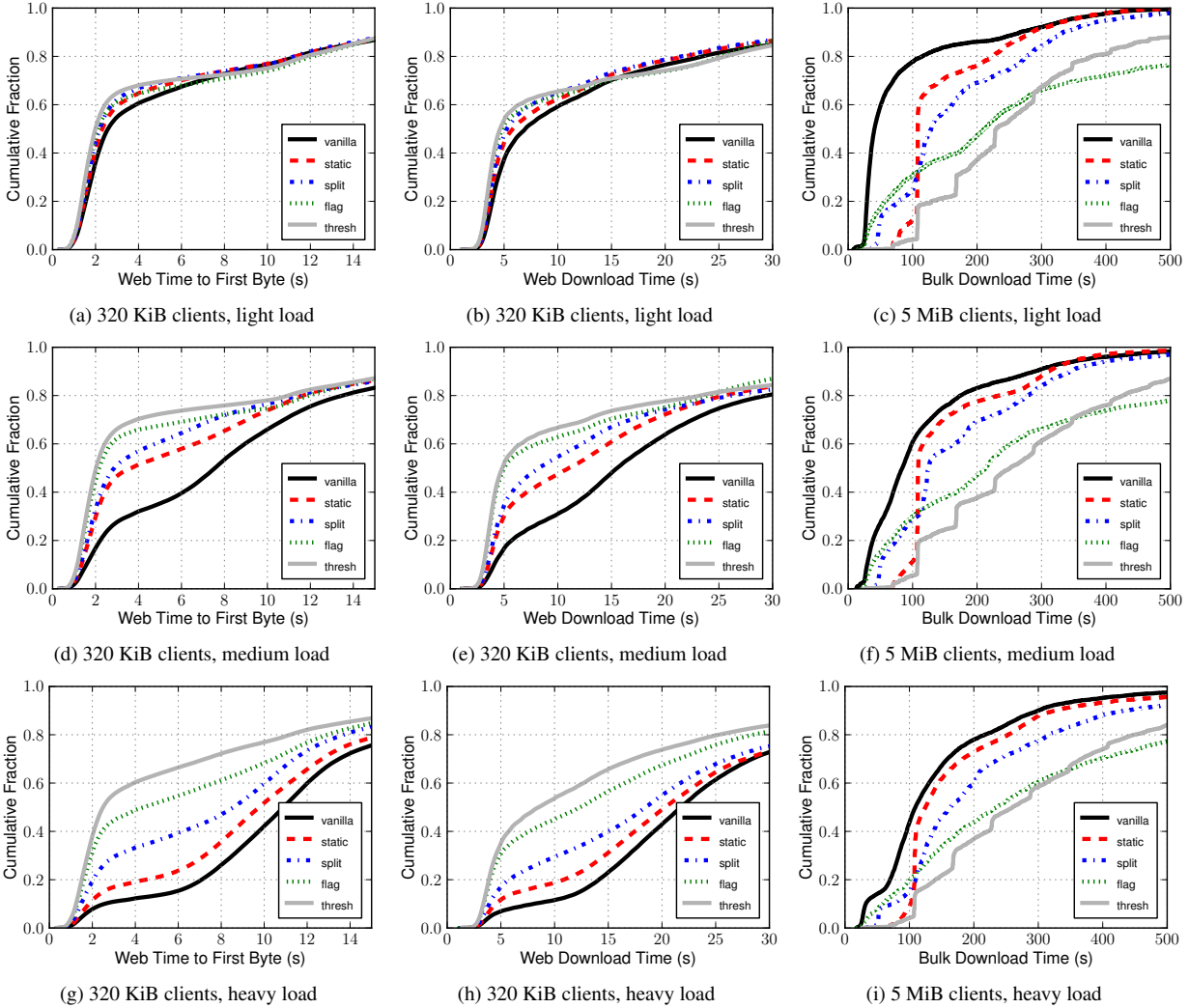


Figure 3: Comparison of client performance for each throttling algorithm and vanilla Tor, under various load. All experiments use 950 web clients. We vary the load between “light,” “medium,” and “heavy” by setting the number of bulk clients to 25 for 3a–3c, to 50 for 3d–3f, and to 100 for 3g–3i. The time to first byte indicates network responsiveness while the download time indicates overall client performance. The parameters for each algorithm are tuned based on experiments presented in [33].

Overall, static throttling results in the least amount of bulk traffic throttling while providing the lowest benefit to web clients. For the bit-splitting algorithm, we see improvements over static throttling for web clients for both time to first byte and overall download times, while download times for bulk clients are also slightly increased. Flagging and threshold throttling perform somewhat more aggressive throttling of bulk traffic and therefore also provide the greatest improvements in web client performance.

We find that each algorithm is effective at throttling bulk clients independent of network load, as evident in Figures 3c, 3f and 3i. However, performance benefits for web clients vary slightly as the network load changes. When the number of bulk clients is halved, throughput in Figure 3b is fairly similar across algorithms. How-

ever, when the number of bulk clients is doubled, responsiveness in Figure 3g and throughput in Figure 3h for both the static throttling and the adaptive bit-splitting algorithm lag behind the performance of the flagging and threshold algorithms. Static throttling would likely require a reconfiguration of throttling parameters while bit-splitting adjusts the throttle rate less effectively than our flagging and threshold algorithms.

As seen in Figures 3a, 3d, and 3g, as the load changes, the strengths of each algorithm become apparent. The flagging and threshold algorithms stand out as the best approaches for both web client responsiveness and throughput, and Figures 3c, 3f, and 3i show that they are also most aggressive at throttling bulk clients. The flagging algorithm appears very effective at accurately classifying bulk connections regardless of network

		vanilla	static	split	flag	thresh
light	Data (GiB)	88.3	80.3	78.3	72.1	69.8
	Web (%)	74.5	83.7	85.9	92.7	90.1
	Bulk (%)	25.5	16.3	14.1	7.3	9.9
medium	Data (GiB)	92.2	88.6	84.7	77.7	76.3
	Web (%)	65.8	72.4	75.0	86.2	82.8
	Bulk (%)	34.2	27.6	25.0	13.8	17.2
heavy	Data (GiB)	94.7	91.1	85.0	81.7	85.0
	Web (%)	55.8	60.5	64.3	75.4	71.2
	Bulk (%)	44.2	39.5	35.7	24.6	28.8

Table 1: Total data downloaded in our simulations by client type. Throttling reduces the bulk traffic share of the load on the network. The flagging algorithm is the best at throttling bulk traffic under light, medium, and heavy loads of 25, 50, and 100 bulk clients, respectively.

load. The threshold algorithm maximizes web client performance in our simulations among all loads and all algorithms tested, since it effectively throttles the worst bulk clients while utilizing extra bandwidth when possible. Both the threshold and flagging algorithms perform well over all network loads tested, and their usage in Tor would require little-to-no maintenance while providing significant performance improvements for web clients.

Aggregate download statistics are shown in Table 1. The results indicate that we are approximating the load distribution measured by McCoy *et al.* [38] reasonably well. The data also indicates that as the number of bulk clients in our simulation increases, so does the total amount of data downloaded and the bulk fraction of the total as expected. The data also shows that all throttling algorithms reduce the total network load. Static throttling reduces load the least, while our adaptive flagging algorithm is both the best at reducing both overall load and the bulk percentage of network traffic. Each of our adaptive algorithms are better at reducing load than static throttling, due to their ability to adapt to network dynamics. The relative difference between each algorithm’s effectiveness at reducing load roughly corresponds to the relative difference in web client performance in our experiments, as we discussed above.

Discussion. The best algorithm for Tor depends on multiple factors. Although not maximizing web client performance, bit-splitting is the simplest, the most efficient, and the most network neutral approach (every connection is allowed the same portion of a guard’s capacity). This “subtle” or “delicate” approach to throttling may be favorable if supporting multiple client behaviors is desirable. Conversely, the flagging algorithm may be used to identify a specific class of traffic and throttle it aggressively, creating the potential for the largest increase in performance for unthrottled traffic. We are currently exploring improvements to our statistical classification techniques to reduce false positives and to improve the

control over traffic of various types. For these reasons, we feel the bit-splitting and flagging algorithms will be the most useful in various situations. We suggest that perhaps bit-splitting is the most appropriate throttling algorithm to use initially, even if something more aggressive is desirable in the long term.

While requiring little maintenance, our algorithms were designed to use only local relay information. Therefore, they are incrementally deployable while relay operators may choose the desired throttling algorithm independent of others. Our algorithms are already implemented in Tor and software patches are available [5].

5 Analysis and Discussion

Having shown the performance benefits of throttling bulk clients in Section 4, we now analyze the security of throttling against adversarial attacks on anonymity. We will discuss the direct impact of throttling on anonymity: what an adversary can learn when guards throttle clients and how the information leaked affects the anonymity of the system. We lastly discuss potential strategies clients may use to elude the throttles.

Before exploring practical attacks, we introduce two techniques an adversary may use to gather information about the network given that a generic throttling algorithm is enabled at all guards. Similar techniques used for throughput-based traffic analysis outside the context of throttling are discussed in detail by Mittal *et al.* [39]. Discussion about the security of our throttling algorithms in the context of practical attacks will follow.

5.1 Gathering Information

Our analysis uses the following terminology. At time t , the throughput of a connection between a client and a guard is λ_t , the rate at which the client will be throttled is α_t , and the allowed data burst is β . Note that, as consistent with our algorithms, the throttle rate may vary over time but the burst is a static system-wide parameter.

Probing Guards. Using the above terminology, a connection is throttled if, over the last s seconds, its throughput exceeds the allowed initial burst and the long-term throttle rate:

$$\sum_{k=t-s}^t (\lambda_k) \geq \beta + \sum_{k=t-s}^t (\alpha_k) \quad (1)$$

A client may perform a simple technique to probe a specific guard node and determine the rate at which it gets throttled. The client may open a single circuit through the guard, selecting other high-bandwidth relays to ensure that the circuit does not contain a bottleneck. Then, it may download a large file and observe the change in throughput after receiving a burst of β payload bytes.

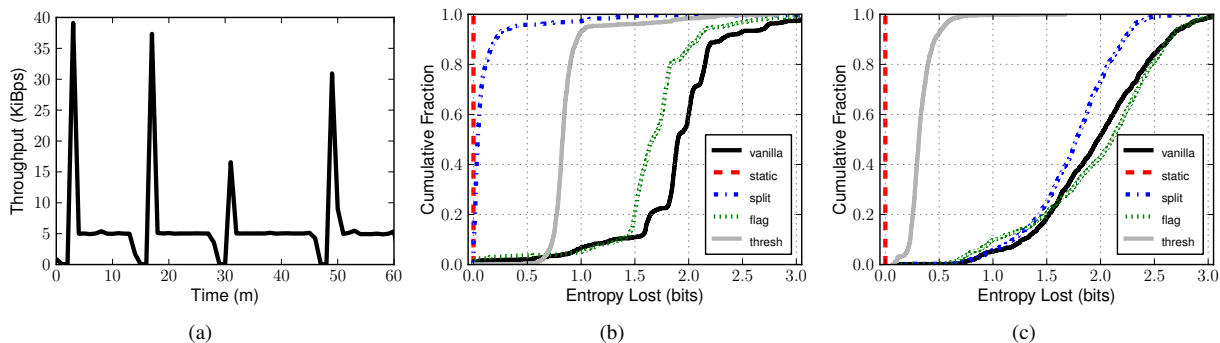


Figure 4: 4a: Client’s may discover the throttle rate by probing guards. 4b: Information leaked by learning circuit throughputs. 4c: Information leaked by learning guards’ throttle rates.

If the first β bytes are received at time t_1 and the download finishes at time $t_2 \geq t_1$, the throttle rate at any time t in this interval can be approximated by the mean throughput leading up to t :

$$\forall t \in [t_1, t_2], \alpha_t \approx \frac{\sum_{k=t_1}^t (\lambda_k)}{t - t_1} \quad (2)$$

Therefore, α_{t_2} approximates the actual throttle rate. Note that this approximation may under-estimate the actual throttle rate if the throughput falls below the throttle rate during the measured interval.

We simulate probing in Shadow [2, 31] to show its effectiveness against the static throttling algorithm. As apparent in Figure 4a, the throttle rate was configured at 5 KiB/s and the burst at 2 MiB. With enough resources, an adversary may probe every guard node to form a complete list of throttle rates.

Testing Circuit Throughput. A web server may determine the throughput of a connecting client’s circuit by using a technique similar to that presented by Hopper *et al.* [30]. When the server gets an HTTP request from a client, it may inject either special JavaScript or a large amount of garbage HTML into a form element included in the response. The injected code will trigger a second client request after the original response is received. The server may adjust the amount of returned data and measure the time between when it sent the first response and when it received the second request to approximate the throughput of the circuit.

5.2 Adversarial Attacks

We now explore several adversarial attacks in the context of client throttling algorithms, and how an adversary may use those attacks to learn information and affect the anonymity of a client.

Attack 1. In our first attack, an adversary obtains a distribution on throttle rates by probing all Tor guard relays.

We assume the adversary has resources to perform such an attack, e.g. by utilizing a botnet or other distributed network such as PlanetLab [13]. The adversary then obtains access to a web server and tests the throughput of a target circuit. With this information, the adversary may reduce the *anonymity set* of the circuit’s potential guards by eliminating those whose throttle rate is inconsistent with the measured circuit throughput.

This attack is somewhat successful against all of the throttling algorithms we have described. For bit-splitting, the anonymity set of possible guard nodes will consist of those whose bandwidth and number of active connections would throttle to the throughput of the target circuit or higher. By running the attack repeatedly over time, an intersection will narrow the set to those whose throttle rate is consistent with the target circuit throughput at all measured times.

The flagging algorithm throttles all flagged connections to the same rate system-wide. (We assume here that the set of possible guards is already narrowed to those whose bandwidth is consistent with the target circuit’s throughput irrespective of throttling.) A circuit whose throughput matches the system-wide rate is either flagged at some guard or just coincidentally matches the system-wide rate and is not flagged because its EWMA has remained below the `splitRate` (see Algorithm 2) for its guard long enough to not be flagged or become unflagged. The throttling rate is thus not nearly as informative as for bit-splitting. If we run the attack repeatedly however, we can eliminate from the anonymity set any guard such that the EWMA of the target circuit should have resulted in a throttling but did not. Also, if the EWMA drops to the throttling rate at precise times (ignoring unusual coincidence), we can eliminate any guard that would not have throttled at precisely those times. Note that this determination must be made after the fact to account for the burst bucket of the target circuit, but it can still be made precisely.

The potential for information going to the attacker in the threshold algorithm is a combination of the potential in each of the above two algorithms. The timing of when a circuit gets throttled (or does not when it should have been) can narrow the anonymity set of entry guards as in the flagging algorithm. Once the circuit has been throttled, then any fluctuation in the throttling rate that separates out the guard nodes can be used to further narrow the set. Note that if a circuit consistently falls below the throttling rate of all guards, an attacker can learn nothing about its possible entry guard from this attack. Attack 2 considerably improves the situation for the adversary.

We simulated this attack in Shadow [2, 31]. An adversary probes all guards and forms a distribution on the throttle rate at which a connection would become throttled. We then form a distribution on circuit throughputs over each minute, and remove any guard whose throttle rate is outside a range of one standard deviation of those throughputs. Since there are 50 guards, the maximum entropy is $\log_2(50) \approx 5.64$; the entropy lost by this attack for various throttling algorithms relative to vanilla Tor is shown in Figure 4b. We can see that the static algorithm actually loses no information, since all connections are throttled to the same rate, while vanilla Tor without throttling actually loses *more* information than any of the throttling algorithms. Therefore, the distribution on guard bandwidth leaks more information than throttled circuits' throughputs.

Attack 2. As in Attack 1, the adversary again obtains a distribution on throttle rates of all guards in the system. However, the adversary slightly modifies its circuit testing by continuously sending garbage responses. The adversary adjusts the size of each response so that it may compute the throughput of the circuit over time and approximates the rate at which the circuit is throttled. By comparing the estimated throttle rate to the distribution on guard throttle rates, the adversary may again reduce the anonymity set by removing guards whose throttle rate is inconsistent with the estimated circuit throttle rate.

For bit-splitting, by raising and lowering the rate of garbage sent, the attacker can match this with the throttled throughput of each guard. The only guards in the anonymity set would be those that share the same throttling rate that matches the flooded circuit's throughput at all times. To maximize what he can learn from flagging, the adversary should raise the EWMA of the target circuit at a rate that will allow him to maximally differentiate guards with respect to when they would begin to throttle a circuit. If this does not uniquely identify the guard, he can also use the rate at which he diminishes garbage traffic to try to learn more from when the target circuit stops being throttled. As in Attack 1 from the threshold algorithm, the adversary can match the signature of both fluctuations in throttling rate over time and

the timing of when throttling is applied to narrow the set of possible guards for a target circuit.

We simulated this attack using the same data set as Attack 1. Figure 4c shows that a connection's throttle rate generally leaks slightly more information than its throughput. As in Attack 1, guards' bandwidth in our simulation leaks more information than the throttle rate of each connection for all but the flagging algorithm.

Attack 3. An adversary controlling two malicious servers can link streams of a client connecting to each of them at the same time. The adversary uses the circuit testing technique to send a response of $\frac{\beta}{2}$ bytes in size to each of two requests. Then, small "test" responses are returned after receiving the clients' second requests. If the throughput of each circuit when downloading the "test" response is consistently throttled, then it is possible that the requests are coming from the same client. This attack relies on the observation that all traffic on the same client-to-guard connection will be throttled at the same time since each connection has a single burst bucket.

This attack is intended to indicate if and when a circuit is throttled, rather than the throttling rate. It will therefore not be effective against bit splitting, but will work against flagging or threshold throttling.

Attack 4. Our final attack is an active denial of service attack that can be used to confirm a circuit's entry guard with high probability. In this attack, the adversary attempts to adjust the throttle rate of each guard in order to identify whether it carries a target circuit. An adversary in control of a malicious server may monitor the throughput of a target circuit over time, and may then open a large number of connections to each guard node until a decrease in the target circuit's throughput is observed. To confirm that a guard is on the target circuit, the adversary can alternate between opening and closing guard connections and continue to observe the throughput of the target circuit. If the throughput is consistent with the adversary's behavior, it has found the circuit's guard with high probability.

The one thing not controlled by the adversary in Attack 2 is a guard's criterion for throttling at a given time – `splitRate` for bit splitting and flagging and `selectIndex` for threshold throttling (see Algorithms 1, 2, and 3). All of these are controlled by the number of circuits at the guard, which Attack 4 places under the control of the adversary. Thus, under Attack 4, the adversary will have precise control over which circuits get throttled at which rate at all times and can therefore uniquely determine the entry guard.

Note that all of Attacks 1, 2, and 4 are intended to learn about the possible entry guards for an attacked circuit. Even if completely successful, this does not fully de-anonymize the circuit. But since guards themselves are chosen for persistent use by a client, they can add

to pseudonymous profiling and can be combined with other information, such as that uncovered by Attack 3, to either reduce anonymity of the client or build a richer pseudonymous profile of it.

5.3 Eluding Throttles

A client may try multiple strategies to avoid being throttled. A client may instrument its downloading application and the Tor software to send application data over multiple Tor circuits. However, these circuits will still be subject to throttling since each of them uses the same throttled TCP connection to the guard. A client may avoid this by attempting to create multiple TCP connections to the guard. In this case, the guard may easily recognize that the connection requests come from the same client and can either deny the establishment of multiple connections or aggregate the accounting of all connections to that client. A client may use multiple guard nodes and send application data over each separate guard connection, but the client significantly decreases its anonymity by subverting the guard mechanism [58, 59]. Finally, the client could run and use its own guard node and avoid throttling itself. Although this strategy may actually benefit the network since it reduces the amount of Tor’s capacity consumed by the client, the cost of running a guard may be sufficient to prevent its wide-scale adoption.

It’s important to note that the “cheating” techniques outlined above do not decrease the *security* or *performance* below what unthrottled Tor provides. At worst, even if all clients somehow manage to elude the throttles, performance and security both regress to that of unthrottled Tor. In other words, throttling can only *improve* the situation whether or not “cheating” occurs in practice.

6 Related Work

6.1 Improving Tor’s Performance

Recent work on improving Tor’s performance covers a wide range of topics, which we now enumerate.

Incentives. A recognition that Tor is limited by its bandwidth resources has resulted in several proposals for developing performance incentives for volunteering bandwidth as a Tor relay. New relays would provide additional resources and improve network performance. Ngan *et al.* explore giving better performance to relays that attain the `fast` and `stable` relay flags [43]. These relays are marked with a “gold star” in the directory. Gold star relays may build circuits through other gold star relays, improving download performance. This scheme has a severe anonymity problem: any relay

on a gold star circuit can determine with absolute certainty that the client is also a gold star relay. Jansen *et al.* explore reducing anonymity problems from the gold star approach by distributing anonymous tickets to all clients [32]. Relays then collect tickets from clients in exchange for prioritized service and can prioritize their own traffic in return. However, a centralized bank limits the allowable number of tickets in circulation, leading to spending strategies that may reduce anonymity. Finally, Moore *et al.* independently explored using static throttling configurations as a way to produce incentives for users to run relays in Tortoise [41]. Tortoise’s throttling configurations must be monitored as network load changes, and anonymity with Tortoise is slightly worse than with the gold star scheme: the intersection attack is improved since gold star nodes retain their gold stars for several months after dropping from the consensus, whereas Tortoise only unthrottles nodes that are in the current consensus.

Relay Selection. Snader and Borisov [51] suggest an algorithm where relays opportunistically measure their peers’ performance, allowing clients to use empirical aggregations to select relays for their circuits. A user-tunable mechanism for selecting relays is built into the algorithm: clients may adjust how often the fast relays get chosen, trading off anonymity and performance while not significantly reducing either. It was shown that this approach increases accuracy of available bandwidth estimates and reduces reaction time to changes in network load while decreasing vulnerabilities to low-resource routing attacks. Wang *et al.* [57] propose a congestion-aware path selection algorithm where clients choose paths based on information gathered during opportunistic and active measurements of relays. Clients use latency as an indication of congestion, and reject congested relays when building circuits. Improvements were realized for a single client, but it’s unclear how the new strategy would affect the network if used by all clients.

Scheduling. Alternative scheduling approaches have recently gained interest. Tang and Goldberg [52] suggest each relay track the number of packets it schedules for each circuit. After a configurable time-period, packet counts are exponentially decayed so that data sent more recently has a greater influence on the packet count. For each scheduling decision, the relay flushes the circuit with the lowest cell count, favoring circuits that have not sent much data recently while preventing bursty traffic from significantly affecting scheduling priorities. Jansen *et al.* [32] investigate new schedulers based on the proportional differentiation model [21] and differentiable service classes. Relays track the delay of each service class and prioritize scheduling so that relative delays are proportional to configurable differentiation parameters, but the schedulers require a mecha-

nism (tickets) for differentiating traffic into classes. Finally, Tor's round-robin TCP read/write schedulers have recently been noted as a source of unfairness for relays that have an unbalanced number of circuits per TCP connection [54]. Tschorsch and Scheuermann suggest that a round-robin scheduler could approximate a max-min algorithm [24] by choosing among all circuits rather than all TCP connections. More work is required to determine the suitability of this approach in Tor.

Congestion. Improving performance and reducing congestion has been studied by taking an in-depth look at Tor's circuit and stream windows [7]. AlSabah *et al.* experiment with dynamically adjusting window sizes and find that smaller window sizes effectively reduce queuing delays, but also decrease bandwidth utilization and therefore hurt overall download performance. As a result, they implement and test an algorithm from ATM networks called the N23 scheme, a link-by-link flow control algorithm. Their adaptive N23 algorithm propagates information about the available queue space to the next upstream router while dynamically adjusting the maximum circuit queue size based on outgoing cell buffer delays, leading to a quicker reaction to congestion. Their experiments indicate slightly improved response and download times for 300 KiB files.

Transport. Tor's performance has also been analyzed at the socket level, resulting in suggestions for a UDP-based mechanism for data delivery [56] or using a user-level TCP stack over a DTLS tunnel [47]. While Tor currently multiplexes all circuits over a single kernel TCP stream to control information leakage, the TCP-over-DTLS approach suggests separate user TCP streams for each circuit and sends all TCP streams between two relays over a single kernel DTLS-secured [40] UDP socket. As a result, a circuit's TCP window is not unfairly reduced when other high-bandwidth circuits cause queuing delays or dropped packets.

6.2 Bandwidth Management

Our approach to bandwidth management in this paper has been to use a token bucket rate-limiter, a classic traffic shaping mechanism [55], to ensure that traffic conforms to the desired policies. We now briefly discuss other approaches to bandwidth management.

Quality of Service. Networks often want to provide a certain quality of service (QoS) to their subscribers. There are two main approaches to QoS: Integrated Services (IntServ) and Differentiated Services (DiffServ).

In the IntServ [11, 50] model, applications request resources from the network using the resource reservation protocol [60]. Since the network must maintain the expected quality for its current commitments, it must ensure the load of the network remains below a certain

level. Therefore, new requests may be denied if the network is unable to provide the resources requested. This approach does not work well in an anonymity network like Tor since clients would be able to request unbounded resources without accountability and the network would be unable to fulfill most requests due to bottlenecks.

In the DiffServ [9] model, applications notify the network of the desired service type by setting bits in the IP header. Routers then tailor performance toward an expected notion of fairness (e.g. max-min fairness [24, 34] or proportional fairness [20, 21, 35]). Leaking this type of information about a client's traffic flows is a significant risk to privacy and ways to provide differentiated service without such risk do not currently exist.

Scheduling. Scheduling algorithms, such as fair queuing [15] and round robin [24, 25], affect the order in which packets are sent out of a given node, but generally do not change the total number of packets being sent. Therefore, unless the sending rate is explicitly reduced, the network will still contain similar load regardless of the relative priority of individual packets. As explained in Section 1 and Section 3, scheduling does not directly reduce network congestion, but may cooperate with other bandwidth management techniques to achieve the desired performance characteristics of traffic classes.

7 Conclusion

This paper analyzes client throttling by guard relays to reduce Tor network bottlenecks and improve responsiveness. We explore static throttling configurations while designing, implementing, and evaluating three new throttling algorithms that adaptively select which connections get throttled and dynamically adjust the throttle rate of each connection. Our adaptive throttling techniques use only local relay information and are considerably more effective than static throttling since they do not require re-evaluation of throttling parameters as network load changes. We find that client throttling is effective at both improving performance for interactive clients and increasing Tor's network resilience. We also analyzed the effects throttling has on anonymity and discussed the security of our algorithms against realistic adversarial attacks. We find that throttling improves anonymity: a guard's bandwidth leaks more information about its circuits when throttling is *disabled*.

Future Work. There are many directions for future research. Our current algorithms may be modified to optimize performance by improving classification of bulk traffic, considering alternative strategies for distinguishing web from bulk connections. Additional approaches to rate-tuning are also of interest, e.g. it may be possible to further improve web client performance using proportional fairness to schedule traffic on circuits. Also of

interest is an analysis of throttling in the context of congestion and flow control to determine the interrelation and effects the algorithms have on each other. Finally, a deeper understanding of our algorithms and their effects on client performance would be possible through analysis on the live Tor network.

Acknowledgements. We thank Roger Dingledine for helpful discussions regarding this work and the anonymous reviewers for their feedback and suggestions. This research was supported by NFS grant CNS-0917154, ONR, and DARPA.

References

- [1] The Libevent Event Notification Library, Version 2.0. <http://monkey.org/~provos/libevent/>.
- [2] The Shadow Simulator. <http://shadow.cs.umn.edu/>.
- [3] The Tor Metrics Portal. <http://metrics.torproject.org/>.
- [4] The Tor Project. <https://www.torproject.org/>.
- [5] Throttling Algorithms Code Repository. <https://github.com/robjansen/torclone>.
- [6] ACQUISTI, A., DINGLEDINE, R., AND SYVERSON, P. On the Economics of Anonymity. In *Proceedings of Financial Cryptography* (January 2003), R. N. Wright, Ed., Springer-Verlag, LNCS 2742.
- [7] ALSABAH, M., BAUER, K., GOLDBERG, I., GRUNWALD, D., MCCOY, D., SAVAGE, S., AND VOELKER, G. DefenestraTor: Throwing out Windows in Tor. In *Proceedings of the 11th International Symposium on Privacy Enhancing Technologies* (2011).
- [8] BACK, A., MOLLER, U., AND STIGLIC, A. Traffic Analysis Attacks and Trade-offs in Anonymity Providing Systems. In *Proceedings of Information Hiding Workshop* (2001), pp. 245–257.
- [9] BLAKE, S., BLACK, D., CARLSON, M., DAVIES, E., WANG, Z., AND WEISS, W. An Architecture for Differentiated Services, 1998.
- [10] BORISOV, N., DANEZIS, G., MITTAL, P., AND TABRIZ, P. Denial of Service or Denial of Security? In *Proceedings of the 14th ACM conference on Computer and communications security* (2007), ACM, pp. 92–102.
- [11] BRADEN, B., CLARK, D., AND SHENKER, S. Integrated Service in the Internet Architecture: an Overview.
- [12] CHEN, F., AND PERRY, M. Improving Tor Path Selection. <https://gitweb.torproject.org/torspec.git/blob/HEAD:proposals/151-path-selection-improvements.txt>.
- [13] CHUN, B., CULLER, D., ROSCOE, T., BAVIER, A., PETERSON, L., WAWRZONIAK, M., AND BOWMAN, M. PlanetLab: an Overlay Testbed for Broad-coverage Services. *SIGCOMM Computer Communication Review* 33 (2003), 3–12.
- [14] CROTTI, M., DUSI, M., GRINGOLI, F., AND SALGARELLI, L. Traffic Classification Through Simple Statistical Fingerprinting. *SIGCOMM Comput. Commun. Rev.* 37 (January 2007), 5–16.
- [15] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and Simulation of a Fair Queuing Algorithm. In *ACM SIGCOMM Computer Communication Review* (1989), vol. 19, ACM, pp. 1–12.
- [16] DINGLEDINE, R. Iran Blocks Tor. <https://blog.torproject.org/blog/iran-blocks-tor-tor-releases-same-day-fix>.
- [17] DINGLEDINE, R. Research problem: adaptive throttling of Tor clients by entry guards. <https://blog.torproject.org/blog/research-problem-adaptive-throttling-tor-clients-entry-guards>.
- [18] DINGLEDINE, R., AND MATHEWSON, N. Anonymity Loves Company: Usability and the Network Effect. In *Proceedings of the Fifth Workshop on the Economics of Information Security (WEIS 2006)*, Cambridge, UK, June (2006).
- [19] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The Second-Generation Onion Router. In *Proceedings of the 13th USENIX Security Symposium* (2004).
- [20] DOVROLIS, C., AND RAMANATHAN, P. A Case for Relative Differentiated Services and the Proportional Differentiation Model. *Network, IEEE* 13, 5 (1999), 26–34.
- [21] DOVROLIS, C., STILIADIS, D., AND RAMANATHAN, P. Proportional Differentiated Services: Delay Differentiation and Packet Scheduling. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (1999), pp. 109–120.
- [22] EVANS, N., DINGLEDINE, R., AND GROTHOFF, C. A Practical Congestion Attack on Tor Using Long Paths. In *Proceedings of the 18th USENIX Security Symposium* (2009), pp. 33–50.
- [23] GOLDSCHLAG, D. M., REED, M. G., AND SYVERSON, P. F. Hiding Routing Information. In *Proceedings of Information Hiding Workshop* (1996), pp. 137–150.
- [24] HAHNE, E. Round-robin Scheduling for Max-min Fairness in Data Networks. *IEEE Journal on Selected Areas in Communications* 9, 7 (1991), 1024–1039.
- [25] HAHNE, E., AND GALLAGER, R. Round-robin Scheduling for Fair Flow Control in Data Communication Networks. *NASA STI/Recon Technical Report N 86* (1986), 30047.
- [26] HARDIN, G. The Tragedy of the Commons. *Science* 162, 3859 (December 1968), 1243–1248.
- [27] HERNANDEZ-CAMPOS, F., JEFFAY, K., AND SMITH, F. Tracking the Evolution of Web Traffic: 1995-2003. In *The 11th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer Telecommunications Systems* (2003), pp. 16–25.
- [28] HINTZ, A. Fingerprinting Websites using Traffic Analysis. In *Proceedings of Privacy Enhancing Technologies Workshop* (2002), pp. 171–178.
- [29] HJELMVIK, E., AND JOHN, W. Statistical Protocol Identification with SPID: Preliminary Results. In *Swedish National Computer Networking Workshop* (2009).
- [30] HOPPER, N., VASSERMAN, E., AND CHAN-TIN, E. How Much Anonymity Does Network Latency Leak? *ACM Transactions on Information and System Security* 13, 2 (2010), 1–28.
- [31] JANSEN, R., AND HOPPER, N. Shadow: Running Tor in a Box for Accurate and Efficient Experimentation. In *Proceedings of the 19th Network and Distributed System Security Symposium* (2012).
- [32] JANSEN, R., HOPPER, N., AND KIM, Y. Recruiting New Tor Relays with BRAIDS. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (2010), pp. 319–328.
- [33] JANSEN, R., SYVERSON, P., AND HOPPER, N. Throttling Tor Bandwidth Parasites. Tech. Rep. 11-019, University of Minnesota, 2011.
- [34] KATEVENIS, M. Fast Switching and Fair Control of Congested Flow in Broadband Networks. *Selected Areas in Communications, IEEE Journal on* 5, 8 (1987), 1315–1326.

- [35] KELLY, F., MAULLOO, A., AND TAN, D. Rate Control for Communication Networks: Shadow Prices, Proportional Fairness and Stability. *Journal of the Operational Research society* 49, 3 (1998), 237–252.
- [36] KOHNEN, C., UBERALL, C., ADAMSKY, F., RAKOCEVIC, V., RAJARAJAN, M., AND JAGER, R. Enhancements to Statistical Protocol IDentification (SPID) for Self-Organised QoS in LANs. In *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on* (2010), IEEE, pp. 1–6.
- [37] LOESING, K. Measuring the Tor network: Evaluation of client requests to directories. Tech. rep., Tor Project, 2009.
- [38] MCCOY, D., BAUER, K., GRUNWALD, D., KOHNO, T., AND SICKER, D. Shining Light in Dark Places: Understanding the Tor Network. In *Proceedings of the 8th International Symposium on Privacy Enhancing Technologies* (2008), pp. 63–76.
- [39] MITTAL, P., KHURSHID, A., JUEN, J., CAESAR, M., AND BORISOV, N. Stealthy Traffic Analysis of Low-Latency Anonymous Communication Using Throughput Fingerprinting. In *Proceedings of the 18th ACM conference on Computer and Communications Security* (October 2011).
- [40] MODADUGU, N., AND RESCORLA, E. The Design and Implementation of Datagram TLS. In *Proceedings of the 11th Network and Distributed System Security Symposium* (2004).
- [41] MOORE, W. B., WACEK, C., AND SHERR, M. Exploring the Potential Benefits of Expanded Rate Limiting in Tor: Slow and Steady Wins the Race With Tortoise. In *Proceedings of 2011 Annual Computer Security Applications Conference* (December 2011).
- [42] MURDOCH, S., AND DANEZIS, G. Low-cost Traffic Analysis of Tor. In *IEEE Symposium on Security and Privacy* (2005), pp. 183–195.
- [43] NGAN, T.-W. J., DINGLEDINE, R., AND WALLACH, D. S. Building Incentives into Tor. In *The Proceedings of Financial Cryptography* (2010).
- [44] ØVERLIER, L., AND SYVERSON, P. Locating Hidden Servers. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (May 2006), IEEE CS.
- [45] RAMACHANDRAN, S. Web Metrics: Size and Number of Resources. <http://code.google.com/speed/articles/web-metrics.html>, 2010. Accessed February, 2012.
- [46] RAYMOND, J. Traffic Analysis: Protocols, Attacks, Design Issues, and Open Problems. In *Designing Privacy Enhancing Technologies* (2001), pp. 10–29.
- [47] REARDON, J., AND GOLDBERG, I. Improving Tor using a TCP-over-DTLS tunnel. In *Proceedings of the 18th USENIX Security Symposium* (2009).
- [48] REED, M., SYVERSON, P., AND GOLDSCHLAG, D. Anonymous Connections and Onion Routing. *IEEE Journal on Selected Areas in Communications* 16, 4 (1998), 482–494.
- [49] SERJANTOV, A., AND SEWELL, P. Passive Attack Analysis for Connection-based Anonymity Systems. *Computer Security—ESORICS* (2003), 116–131.
- [50] SHENKER, S., PARTRIDGE, C., AND GUERIN, R. RFC 2212: Specification of Guaranteed Quality of Service, Sept. 1997. Status: PROPOSED STANDARD.
- [51] SNADER, R., AND BORISOV, N. A Tune-up for Tor: Improving Security and Performance in the Tor Network. In *Proceedings of the 16th Network and Distributed Security Symposium* (2008).
- [52] TANG, C., AND GOLDBERG, I. An Improved Algorithm for Tor Circuit Scheduling. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (2010), pp. 329–339.
- [53] TSCHORSCH, F., AND SCHEUERMANN, B. Refill Intervals. <https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/183-refillintervals.txt>.
- [54] TSCHORSCH, F., AND SCHEUERMANN, B. Tor is Unfair—and What to Do About It, 2011.
- [55] TURNER, J. New directions in communications(or which way to the information age?). *IEEE communications Magazine* 24, 10 (1986), 8–15.
- [56] VIECCO, C. UDP-OR: A Fair Onion Transport Design. In *Proceedings of Hot Topics in Privacy Enhancing Technologies* (2008).
- [57] WANG, T., BAUER, K., FORERO, C., AND GOLDBERG, I. Congestion-aware Path Selection for Tor. In *Proceedings of Financial Cryptography* (2012).
- [58] WRIGHT, M., ADLER, M., LEVINE, B. N., AND SHIELDS, C. An Analysis of the Degradation of Anonymous Protocols. In *Proceedings of the Network and Distributed Security Symposium - NDSS '02* (February 2002), IEEE.
- [59] WRIGHT, M., ADLER, M., LEVINE, B. N., AND SHIELDS, C. Defending Anonymous Communication Against Passive Logging Attacks. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy* (May 2003), pp. 28–43.
- [60] ZHANG, L., DEERING, S., ESTRIN, D., SHENKER, S., AND ZAPPALA, D. Rsvp: A new resource reservation protocol. *Network, IEEE* 7, 5 (1993), 8–18.

Chimera: A Declarative Language for Streaming Network Traffic Analysis

Kevin Borders
National Security Agency
krborde@tycho.nsa.gov

Jonathan Springer
Reservoir Labs
springer@reservoir.com

Matthew Burnside
National Security Agency
msburns@tycho.nsa.gov

Abstract

Intrusion detection systems play a vital role in network security. Central to these systems is the language used to express policies. Ideally, this language should be powerful, implementation-agnostic, and cross-platform. Unfortunately, today's popular intrusion detection systems fall short of this goal. Each has their own policy language in which expressing complicated logic requires implementation-specific code. Database systems have adapted SQL to handle streaming data, but have yet to achieve the efficiency and flexibility required for complex intrusion detection tasks.

In this paper, we introduce Chimera, a declarative query language for network traffic processing that bridges the gap between powerful intrusion detection systems and a simple, platform-independent SQL syntax. Chimera extends streaming SQL languages to better handle network traffic by adding structured data types, first-class functions, and dynamic window boundaries. We show how these constructs can be applied to real-world scenarios, such as side-jacking detection and DNS feature extraction. Finally, we describe the implementation and evaluation of a compiler that translates Chimera queries into low-level code for the Bro event language.

1 Introduction

Intrusion detection systems (IDSs) continue to play an essential role in network security. One critical aspect of IDS design is how users express analytic tasks. In particular, *policy* should be separate from the *mechanism* [21]. This leads to simpler policies that are easier to write and easier to share because they have fewer implementation constraints. Separation also increases interoperability, which moves us closer to the goal of having a standardized language for network traffic analysis.

Unfortunately, current IDSs only partially separate policy from mechanism. They each have their own domain-specific languages, which are incompatible with one other. Snort uses a declarative rule language for defining signatures [23], which is limited in its ability

to express stateful analytics. Bro [21] offers a more powerful (Turing complete) event language, but complex operations require procedural programming and direct interaction with data structures, which is cumbersome and leads to dependency between policy and mechanism.

Database systems have been attacking this problem from the opposite end. SQL is a powerful, declarative, standard language, and recent work has extended it to support streaming queries [1, 3, 19]. While these systems are typically too slow to serve as an IDS, Gigascope [7] adopts a more limited SQL-based language and has successfully applied it to packet processing. Unfortunately, Gigascope cannot express many of the complex analytic tasks that are possible in Bro.

In this paper, we introduce Chimera, a declarative query language for network traffic processing that bridges the gap between powerful intrusion detection platforms and simple, implementation-agnostic queries. The goal is to provide an SQL-like syntax while maintaining as much expressive power as possible, and without significantly impacting performance. We achieve this goal by implementing Chimera as an independent language that is compiled down into low-level policies for other platforms. For this paper, we have written a compiler¹ that translates Chimera queries into the Bro event language [21].

Chimera is similar to streaming SQL languages, but has some additional features that make it better-suited for handling network traffic. First, it supports structured data types (lists and maps). This allows rows to more closely reflect the structure of application-layer protocols, almost all of which contain structured data. Chimera also makes dealing with these types easier by introducing a SPLIT operator to break up lists into multiple rows, as well as first-class functions that can be applied to data structures. Chimera also improves upon streaming SQL by introducing dynamic windows. Instead of enforcing strict window specifications at the table level, such as "range 60 minutes, slide 1 minute", Chimera allows window boundary computation using dynamic expressions,

¹Visit <http://www.chimera-query.org> for more information about obtaining the Chimera compiler source code.

such as “UNTIL count() > 10”. This makes it possible to output aggregate results as soon as they are ready, which is extremely important for intrusion prevention and active response scenarios.

We motivate the design of Chimera by examining real-world scenarios where detection requires complex state tracking that is unavailable in a simple system like Snort [23]. We look at existing work on detecting side-jacking – an attack that steals a session ID from an HTTP cookie [22] – and on finding malicious domains with the EXPOSURE system [5]. We also consider two examples of detecting DNS tunnels and identifying spam/phishing servers. After describing the Chimera syntax, we present example queries for these scenarios. When compared to a previous Bro implementation [22], the query for side-jacking demonstrates how analytics in the Chimera language are very concise. The Chimera queries for extracting features used by EXPOSURE [5] led us to identify ambiguities in the original text, highlighting the need for a standard network traffic analysis language.

In final part of the paper, we describe and evaluate the implementation of a Chimera to Bro compiler. The compiler operates in two main stages: (1) it translates queries into a relational algebra, and (2) it generates Bro event language code. We compared the compiler’s output to hand-optimized code for a number of queries by running each side by side on real network traffic. In the worst-case example, compiled code was 3% slower than hand-written code due to extra copying and event handlers. We plan to add optimizations to minimize these issue in the future, but our experiments show that the compiler generates code that with almost the same performance as hand-written code even it is current form.

The rest of the paper is laid out as follows. Section 2 motivates our work with examples of stateful analytics. Section 3 describes the Chimera language. Section 4 presents Chimera queries for example scenarios. Section 5 describes the Bro compiler. Section 6 evaluates teh compiler and discusses future optimizations. Finally, section 7 covers related work and section 8 concludes.

2 Motivation: Stateful Network Analytics

As attacks continue to increase in sophistication, so must analytics that detect them. Over time it is becoming more and more difficult to characterize malicious behavior with simple Snort rules [23]. As a result, many administrators rely on systems like Bro [21] that are able to perform stateful analysis on high-level protocol fields, rather than being constrained to individual packet or flow analysis.

This section outlines a number of scenarios in which simple filtering is not enough. The rest of the paper then

uses these scenarios to motivate the Chimera language and its design. Keep in mind that the analytic techniques presented in this section are not necessarily bulletproof, or even practical in all situations. The point is not to assess the quality of analytics, but to provide examples of logical constructs that we would like to express in the Chimera language.

2.1 Sidejacking

Sidejacking is a term used to describe the attack where a hacker steals a session token from an unencrypted HTTP cookie and then impersonates the legitimate user. This attack is easy to pull off in a coffee-shop environment where there is a public wireless network. Countermeasures include use of HTTPS, and are discussed in work by Riley et al. [22].

Sidejacking can also be detected by monitoring network traffic. An implementation of sidejacking detection has been written for Bro [26]. This script works in the following way:

1. Group incoming HTTP requests by session ID in cookie.
2. When a new request arrives, are the client IP and User-Agent the same?
3. If not, then report sidejacking.

As you can see, the analytic logic is straightforward, but implementation requires non-trivial maintenance of client state on a per-cookie basis.

2.2 Malicious Domains

A recent research project called EXPOSURE introduced a set of sixteen features for detecting malicious domain names [5]. Some of these features could operate on a single domain name, such as the percentage of numerical characters. Many of the features, however, require state tracking across multiple DNS packets. In this paper, we examine some of EXPOSURE’s stateful features. In particular, we will focus on a subset of the DNS answer- and TTL-based features:

- Number of distinct IP addresses per domain name
- Number of domains that share the same IP address
- Average TTL value
- Number of TTL value changes

These features all require parsing the DNS protocol. They also require per-domain state tracking, and the second feature needs additional per-IP state tracking.

The authors of EXPOSURE also identify time-based features that we do not discuss here. It would be possible to adapt the change point detection (CPD) algorithms used by EXPOSURE to run in the Chimera framework. However, describing the implementation of complex algorithms in a streaming model is outside of the scope of this work, and is orthogonal to the design of Chimera.

2.3 DNS Tunnels

The DNS protocol is designed to resolve information about domain names. However, it can also be used for covert communication by storing data in the requested domain name (e.g., *<encoded data>.hacker.com*) and sending data back to the client inside of the IP address field. While this is a low-bandwidth channel, the ubiquity of the DNS protocol makes it likely to bypass firewalls even in restricted networks.

There are many ways to detect DNS tunnels, but we will discuss a particular method here because it highlights an interesting analytic technique. In this method, the following steps are taken to find DNS tunnels:

1. Keep track of all DNS response A records, indexing by the A record IP address.
2. When a packet is seen going to an IP address, remove the corresponding DNS response record.
3. If no packet is ever sent to the A record IP (within a window), increment a counter for the client and server IP addresses from the DNS message.
4. Report tunneling for clients or servers that exceed a threshold of orphaned responses.

This analysis logic is again very straightforward. It assumes that IP address values in DNS responses from tunnels will not actually be used as IP addresses, so most of them will never see follow-up packets. Counting a threshold will eliminate false positives from command-line DNS look-ups (e.g., using the *nslookup* UNIX command) that do not have ensuing connections.

2.4 Phishing/Spam Detection

A lot of research has gone into phishing and spam detection. Some approaches look at message contents, while others look at aggregate measurements like e-mail volume and rate of sending. Here, we will consider a detector that looks for new mail transfer agents (MTAs) through which e-mail is sent to a large number of distinct recipients. The analysis happens as follows:

1. Identify SMTP messages that have a “new” MTA in their path.
2. For 24 hours after a new MTA is seen, count the number of distinct recipients in messages that traverse that MTA.
3. If the count for a new MTA exceeds a threshold, then report phishing/spam.

Though the description of this analytic is concise, implementing it requires a few complicated operations. First, there must be a data structure, such as a Bloom filter, that keeps track of whether each MTA has been seen before. That structure must have at least two windows so that it does not start emitting old values after each time it is purged. The next challenge is that the MTA path is stored in multiple headers within each SMTP message.

Checking whether each MTA on the path is new either requires applying a function to each value or splitting up the SMTP message into one tuple for each MTA. When new SMTP messages arrive, checking to see if one of the MTAs is new within the past 24 hours again requires splitting the tuple prior to a join operation.

3 The Chimera Language

3.1 Query Syntax

The highest level element in the Chimera language is a query statement. Since Chimera operates passively, the only type of query allowed right now is SELECT. Chimera also includes a CREATE VIEW statement, which is effectively a macro that can be used in place of sub-queries. The syntax for a Chimera SELECT query is very similar to SQL, and can be seen in Figure 1. Many elements are shared and behave the same way, including the FROM, WHERE, and UNION. The input and output specifications are a bit different. Explicit data sources are allowed in the query, including a file (PCAP or user-defined CSV), network interface, or list of file names from standard input (the default). Similarly, output will be sent to standard output unless a file is specified. Chimera begins to differ more significantly for the GROUP BY and JOIN operations, as well as the newly introduced SPLIT, which we discuss next. It also supports an expression syntax with different data and function types, which are described in sections 3.2 and 3.4.

3.1.1 GROUP BY

The Chimera language diverges from SQL and traditional streaming database in its semantics for the GROUP BY clause. To support streaming, we have added a TABLESIZE parameter and the UNTIL keyword with an optional GLOBAL parameter and a Boolean expression. TABLESIZE specifies the maximum number of items to hold before discarding old values. (Chimera does not yet implement more intelligent QoS or load shedding like Aurora [1], but TABLESIZE effectively enables memory limits.) The UNTIL condition determines when GROUP BY will generate output. It may contain aggregate functions, such as count or average. If GLOBAL is specified, then the aggregate functions are evaluated with a single global state object, instead of separately for each key. In this case, GROUP BY will output everything in the table when the UNTIL expression becomes true. This is similar to window-based grouping in traditional streaming databases. If GLOBAL is omitted, then each item in the GROUP BY table will be evaluated and flushed independently. This allows implementation of partitioned windows, which are described by Arasu et al. [3].

```

<select_query> ::=
  [SOURCE {STDIN | FILE <fname> | INTERFACE <if>}]
  <select_body>
  [INTO {STDOUT | FILE <fname>}]
<create_view> ::=
  CREATE VIEW <alias> AS <select_body>
<select_body> ::=
  SELECT { * | <expr> [AS <alias>]
    [, <expr> [AS <alias>]]* }
  FROM <table_ref>
  [WHERE <bool_expr>]
  [GROUP BY <expr> [, <expr>]*
    UNTIL [GLOBAL] <bool_expr>
    [TABLESIZE <row_count>]
    [HAVING <bool_expr>]
    [ORDER BY <expr> [, <expr>]* [ASC | DESC]
    [LIMIT <row_count>]]]
  [UNION <select_body>]
<table_ref> ::=
  <table_instance>
  | <table_ref> [[EXCLUSIVE] {LEFT | RIGHT | FULL}
    [OUTER]] [UNORDERED] [SINGLE] JOIN
    <table_instance> ON <expr> EQUALS <expr>
    [TABLESIZE <row_count>]
    [WINDOW <expr>[, <expr>]]
  | <table_ref> SPLIT <expr> AS <alias>, <alias>
<table_instance> ::=
  <table_name> [AS <alias>]
  | ( <select_body> ) AS <alias>

```

Figure 1: Query syntax for the Chimera language

The GROUP BY clause may also include an ORDER BY keyword that takes a sorting parameter. Because Chimera is a stream processing system, some values will inevitably be discarded. ORDER BY ensures that the highest values are kept in the GROUP BY table instead of the newest values (the default). Chimera uses a heap structure to discard rows with the lowest ORDER BY value. This allows computation of “heavy hitters” on a high-volume data stream using very little memory. LIMIT specifies how many rows to output at the end of each window. It defaults to TABLESIZE and is only used if GLOBAL is specified.

3.1.2 JOIN

Chimera introduces a few non-standard features for joins that improve efficiency and enable new analytic semantics. The first difference is that joins are *ordered* by default. This means that the left tuple must arrive before the right tuple. This lets Chimera use only one hash table instead of two, improving efficiency. The keyword UNORDERED can be added to the JOIN clause for standard join semantics.

Because Chimera is a stream processing system, only equi-joins are supported, hence the mandatory EQ (equals) syntax. Furthermore, only one tuple is allowed

per key in the join table. If a new tuple arrives on the same side with the same key, then the old one is discarded without being matched. This ensures that each new tuple will generate at most one output, keeping overhead down to $O(1)$. Support for multi-tuple joins could be added in the future, but their use could negatively affect performance.

The next feature supported by Chimera is a SINGLE JOIN, which enforces one-to-one matching between left and right tuples. Normally, a row from one side of a join is allowed to match multiple rows from the other side. When a match occurs in a SINGLE JOIN, the matching tuple is removed from the join table so that it frees up space and cannot match any other tuples. This is useful when performing an EXCLUSIVE OUTER JOIN, which is similar to a typical outer join, except that the inner part of the join is excluded, leaving only tuples that do not have a match. An EXCLUSIVE LEFT SINGLE JOIN can be used, for example, to detect ICMP ping packets that never receive a reply. Here, SINGLE effectively increases the time that can elapse before declaring a packet unmatched by removing matched packets from the table.

The maximum number of elements stored in the JOIN table can be set with TABLESIZE, just as with GROUP BY, which guarantees a limit on memory utilization. In addition to a size-based limit, JOIN also supports a conditional WINDOW clause, which allows it to selectively age off old tuples from the window. The conditional expression for the WINDOW clause is evaluated in a special context where the oldest tuple is assigned the name *old* in the root object, and the newest tuple given the name *new*. For each new tuple, it and the oldest tuple are used to evaluate the WINDOW expression. If the expression is false, then the old tuple is removed and the expression is re-evaluated against the next oldest tuple. For example, `[new].[time] - [old].[time] < 60` enforces a 60 second time window. There can be two window conditions if the join is UNORDERED, which are applied to incoming left and right tuples, respectively.

3.1.3 SPLIT

Chimera includes the SPLIT keyword to its query language to make it easier to handle structured data types. There are some cases where it makes more sense to process a list or map structure as a single object (e.g., looking up a value at an index), but others where it is better to split the list and handle each item in its own tuple (e.g., examining DNS resource records). The SPLIT keyword takes an expression that evaluates to a structured data type (list or map, discussed in section 3.2) as an argument, as well as an alias name for each individual item, and an alias for the item index (which cannot be derived if there are duplicate items). When a split occurs, Chimera creates a new tuple for each item in the object

argument. These tuples have references to all of the original data, including the structured object, but also contain the individual item (map items are emitted as two-value [key, value] lists) and its index as extra values. If the SPLIT object is empty, then Chimera will emit one tuple with NULL values for both the item and the index.

3.2 Data Types

The Chimera language has several data types that it uses to represent message fields in network traffic. Chimera takes a minimalist approach to typing modeled after the types used in JSON [8]. This makes data manipulation much simpler by reducing the number of functions and operators that are required.

Chimera supports six primitive data types: `Integer`, `Float`, `String`, `Bool`, `Null`, and `IPAddress`. The first five correspond to the four primitive types in JSON, with the additional distinction between integer and floating-point numbers. The `Integer` type does not have any constraint on its size. It will be expanded as necessary if it overflows the bounds of a 32- or 64-bit integer. `Float` types are all double precision. All `String` types are binary strings, which is appropriate for network traffic analysis. The `Bool` and `Null` types are self-explanatory. The remaining data type, `IPAddress`, could have been encapsulated in a `String` or `Integer`. Its existence is not necessary, but it is frequently used in network traffic analysis so we decided to add it out of convenience.

Chimera also supports two structured types: a `List`, and a `Map`. The `List` type directly corresponds to an array in JSON. The `Map` type is similar to maps in other languages, but it also supports ordering and duplicate keys. This makes it better-suited for network protocols that contain map-like structures. The ordering of map elements in a network message may have significance. Keys can also be repeated, both for legitimate and malicious purposes. Internally, `Map` objects are implemented by hash tables when they are created by assigning to a key, and by lists or numerically-indexed tables when they are created by appending key-value pairs. Iterating through a map will yield list objects with two items: a key and a value. The objects will be in the original insertion order if the map was created by appending items.

3.3 Naming and Schemata

One core part of the Chimera query language is the set of available schemata. In general, Chimera is not tied to any specific schemata or naming system. In fact, it supports CSV file input with user-defined column names. In this mode, Chimera reads the column names from the first line of a CSV file and applies them to each row.

When dealing with network traffic instead of user-defined meta-data, it is important to have a common naming scheme that is the same across all platforms.

Right now, the only platform supported by Chimera is Bro. We could have just used the Bro names exactly, but they contain some implementation artifacts. Instead, we opted to create our own protocol schemata and write a Bro translation function for each one. This way, the naming and structure is more closely tied to actual protocol messages than to implementation choices specific to Bro.

Table 1: The schema for HTTP requests in Chimera

Name	Type
packets	List(tcp_packet)
method	String
path	String
version	String
headers	Map(String→String)
body	String

We will not enumerate the schema of every protocol here due to space constraints, but provide an example of the schema for HTTP requests in table 1. This schema is simple and corresponds directly to the protocol structure. In addition, there is a list of `packets` in the schema. All top-level protocols in Chimera have this field, which refers to the original packets that make up the message. This allows you to retrieve original IP addresses, port numbers, etc. It also allows more flexible handling of time because each individual packet's arrival time is exposed in the schema.

Another important aspect of the HTTP request schema is that it does not expose any anomalies or low-level parsing details. For example, we assume that the parser strips out any chunked-encoding headers from the `body` field. Once these are gone, we do not know whether the body was split up into many one-byte chunks, one hundred-byte chunks, or any other chunk sizes. If there were anomalies in a chunk header, such as only having a new-line character instead of a carriage return and a newline, then the parser will either fail altogether or discard the information and continue silently. The problem becomes even more serious for DNS messages where hiding data in slack space is a well-known technique. This is a systemic problem that affects all protocol parsers and is orthogonal to the design of Chimera. The problem could be addressed by adding more fields to the parser that contain raw bytes. If these fields were added to low-level parsers, it would be easy to extend the Chimera naming scheme to include them.

3.4 Functions

In Chimera, functions are essential building blocks used for data manipulation and extraction. Chimera supports four different types of functions, which are described in this section. Functions can be defined by the user in the target language (Bro in this case). The set of available

functions and their definitions are considered outside of the core Chimera language, with the exception of functions for which there are syntactic shortcuts. Examples of other specific functions are given later in section 4, which provides example Chimera queries for analysis scenarios.

3.4.1 Methods

The first function type that Chimera supports is a method. Methods operate on objects and can be chained together using a dot syntax (Example: `<object>.a().b().c()`). Each method function can operate on one or more types of input data, and can generate multiple output types. If any function in a method chain generates a NULL output, then evaluation stops and later functions in the chain are not called.

Within an expression in a Chimera query statement, methods may be called without an explicit base object. In this case, Chimera uses the implicit default object, which is a Map representing a tuple in the current schema. Chimera also supports a square-bracket syntax: `[<field>]`. This is syntactic sugar for calling the get function `get('<field>')`, which will retrieve the first value in the map that has a key matching the given input string. If the get function or bracket syntax is used on a List object, then Chimera assumes that the list consists of Map objects and will add an implicit iterator over the list, returning the first object that is not NULL. Such “apply” functions are discussed more later in this section.

In the Chimera language, arguments to method functions must be literals and cannot be derived from the default tuple object. Functions that need to manipulate multiple elements in the default tuple must be written as static functions (described in the next section) instead of method functions. This was a choice that we made based on readability and it does not effect expressiveness.

3.4.2 Static Functions and Operators

Chimera supports static functions that can operate on multiple objects (Example: `concat(<string1>, <string2>, ...)`). The arguments to static functions can be literals or chains of method functions. Chimera also has a number of basic operators. These operators are essentially syntactic sugar for static function calls, though they may be compiled down to the same operator in the target language if it exists and has the same semantics. Chimera currently support most of the C operators, including:

- Arithmetic: `+`, `-` (subtraction and unary), `*`, `/`, `%` (modulo)
- Comparison: `==`, `!=`, `<`, `>`, `<=`, `>=`
- Logical: `!` (NOT), `&&` (AND), `||` (OR)
- Bitwise: `~` (NOT), `&` (AND), `|` (OR), `^(XOR)`, `<<` (Left Shift), `>>` (Right Shift)

For arithmetic and comparison operators between integers and floats, integers are promoted to floats. Bitwise operations are only allowed on integers, and left shifting an integer will never truncate bits that are set. Instead, it will be expanded so that it can hold the value. For strings and IP addresses, only the comparison operators are supported. For Boolean values, only the equality, inequality, and logical operators are supported.

3.4.3 Aggregate Functions

The next type of function available in Chimera is an aggregate function. Aggregate functions are used to compute some result over multiple data items. Aggregate functions are typically seen in expressions that are part of the SELECT, HAVING, or UNTIL clauses in a statement that uses GROUP BY. In these places, a different aggregate value will be computed for each unique GROUP BY key (each key has a different state). Aggregate functions may also be used in WHERE clauses or statements without GROUP BY, but they will have a single global state in these cases.

The syntax for an aggregate function is exactly the same as for static functions. However, the definition must specify four routines, which are shown in Table 2. These routines are similar to those for defining an aggregate function in a standard relational database.

Table 2: User-defined aggregate function routines

	Arguments	Returns	When Called
<i>Initialization</i>	None	State	Before the first input
<i>Iteration</i>	State, Inputs	State	For each new input
<i>Evaluation</i>	State	Outputs	To read current output
<i>Termination</i>	State	State	At end of each window

Each of the routines in an aggregate function deals with a state object. This state object is returned from calls to aggregate routines (except evaluation, which does not update the state), stored, and then passed back to the next aggregate routine call. This state object is opaque to the rest of the system and can contain anything.

The termination function works a bit differently than in a traditional database due to the streaming nature of Chimera. This function will be called at the end of each window as specified in an UNTIL clause in an aggregate statement. The state object that it returns will be passed back to the next iteration call for the first item in the next window. This allows aggregate functions to maintain state across multiple windows.

Some traditional databases also support a *Merge* routine for user-defined aggregates. This allows intermediate results to be merged together, which allows parallel computation. Chimera does not yet support merging, but could be extended to do so in the future.

3.4.4 Apply Functions

The final type of function available in Chimera is an apply function. An apply function is a method on a structured object that takes another function as a first-class object and applies it to items in the structured object. How the argument is applied depends on the particular function. Apply functions can take normal arguments in parentheses, but use a curly bracket syntax for their function argument (e.g., `[list].apply(<args>){<fnarg>}`) to clearly differentiate them from other function types. Arguments can be passed to inner functions using the symbols `$`, `$2`, `$3`, etc. (“1” omitted from first argument for brevity). This lets user-defined apply functions pass an arbitrary number of arguments. It also allows the inner functions to be methods, static functions, or aggregate functions (e.g., `[list].apply{$.strlen()}` or `[list].apply{count($)}`). This syntax is slightly different from other languages like Javascript, but we felt it to be more concise and easy to read in this context.

Chimera does support multiple levels of apply functions. When there are multiple levels, however, inner functions *cannot* directly reference parent arguments. First-class functions in Chimera are not full closures.

Apply functions can be defined by the user, but a few examples are provided here to illustrate the concept. Note that when iterating over a map instead of a list, each key-value pair is represented as two-item [key, value] list.

- **foreach** – Apply the function to each item in the list and update it with the output value. Example: `[list].foreach{$.substr(3)}`
- **foridx(index)** – Apply the function to the item in the list at the given index and update it with the output value. Example: `[map].foreach{$.foridx(0){$.substr(3)}`
- **iter** – Iteratively apply the function to each item in the list and return the first value that is not NULL. Example: `[list].iter{$.match('as.*df')}`
- **iterall** – Apply the function to all items in the list and return the last output value. Example: `[list].iterall{count($)}`
- **filter** – Apply the function to each item in the list and discard items for which it evaluates to false or NULL. Example: `[map].filter{$.first().strlen() > 3}`
- **find** – Apply the function to each item in the list and return the first item for which it does not evaluate to false or NULL. Example: `[map].find{$.first() == 'A'}`

4 Implementing Analytics in Chimera

In section 2, we introduced several attack scenarios that require advanced analysis capabilities to detect. Now that we have presented the Chimera language, we show here how it can be used to implement analytics for these scenarios. While these scenarios demonstrate many of Chimera’s features and capabilities, they are by no means a complete exposition of its power. The goal here is to provide examples of how the language can be used in practice that serve as a starting point for future work.

4.1 Sidejacking

As you may recall, sidejacking involves searching for multiple clients that are using the same session identifier for a web service. For simplicity, clients can be represented as an IP address and User-Agent pair. Now that we have an understanding of Chimera’s query model, we can break down the analysis task into some key facts:

- This query requires aggregation using the session ID as the GROUP BY key.
- The session ID is inside of a key-value list in the “Cookie” header and will need to be broken out of the list.
- Detection requires counting more than one distinct client. This will be the UNTIL trigger condition.

This leads us to the following query, which cleanly implements sidejacking detection and is much more shorter than the previous Bro implementation [26] (though the Bro implementation contains a few more additional features not included here):

```
SELECT
  list_agg(distinct(concat(
    [packets].[srcip], ':',
    [headers].[User-Agent]))
  AS clientlist
[headers].[Cookie].split(';').
  foreach{$.split('=')}.
  find{$.first() == 'SID'}.last()
  AS sessionid
FROM http
WHERE [sessionid] != NULL
GROUP BY [sessionid]
UNTIL [clientlist].size() > 1
```

The first expression in the SELECT statement extracts the source IP address from the first packet in the connection (HTTP messages are comprised of one or more packets), finds the value of the “User-Agent” header (or NULL if it is missing), and concatenates the two together to form a client identifier string. Because [packets] is a list of map objects, the bracket operator that follows includes an implicit iteration, thus extracting [srcip] from

the first packet in the list. The query then passes this string to the aggregate function `distinct`, which will check each incoming value to see if it has occurred before. If not, it will pass through the value, otherwise it will output `NULL`. The `distinct` function can be implemented with a Bloom filter, or with a hash table if more accuracy is desired. Our implementation of `distinct` in the Bro language currently uses a hash table. Finally, the `list_agg` aggregate function will take each non-`NULL` input item and append it to a list.

The next expression in the `SELECT` statement pulls out the session ID from the "Cookie" header. If there is more than one "Cookie" header, then the implicit call to `get()` made by the square brackets will just grab the first one. The expression then splits the the cookie header value up into a list of strings separated by the `';` character. Next, it iterates over this list with `foreach`, further splitting each string using the `'='` character into a key-value list. Finally, the `find` function extracts the first pair in the list where its first item is the string `'SID'`, and `last` pulls out the corresponding value. If at any point during this chain of functions there is a `NULL` value, then processing will stop and the result will be `NULL`.

The remainder of the query is pretty straightforward. The `WHERE` clause filters out only HTTP messages that have Cookie headers and session IDs. `GROUP BY` aggregates based on the session ID, and `UNTIL` will trigger an output whenever it sees more than one client using the same session ID.

4.2 Malicious Domains

There were several DNS features presented earlier in section 2.2. These features each perform some aggregate computation on DNS responses. In the Chimera language, lists of objects can be split into one tuple for each item using the `SPLIT` command. DNS responses contain lists of answer records in a single DNS packet, which can be split up into individual answer records. However, Chimera also includes a schema for individual resource records (essentially pre-split) that corresponds to resource record events in the Bro language. The queries below use the DNS resource record schema out of convenience, but could use the DNS schema and `SPLIT` as well. Here are queries for each of the listed features:

4.2.1 Number of distinct IP addresses per domain

```
SELECT count_distinct([aip]), [name]
FROM dns_rr
WHERE [aip] != NULL
GROUP BY [name]
UNTIL GLOBAL
    nextwindow([packets].[time], 86400)
```

One thing to note about this query is the `count_distinct` function. Counting the number

of distinct items can be done more efficiently than by keeping a list and computing its size. This query also uses an aggregate function `nextwindow` to compute when the packet timestamp has transitioned into the next 86400-second (one day) time window. It essentially performs integer division and change detection. When this occurs, the entire table will be flushed and the computation will restart.

4.2.2 Number of domains that share the same IP

```
SELECT [name], [ip], [count]
FROM (
    SELECT
        [aip] AS ip
        list_agg(distinct([name])) AS names
        count_distinct([name]) AS count
    FROM dns_rr
    WHERE [aip] != NULL
    GROUP BY [aip]
    UNTIL GLOBAL
        nextwindow([packets].[time], 86400)
) SPLIT names AS name, nameidx
```

This query will keep a list of domains for each IP address, maintain a count of its size, and then output each domain along with an IP address and count every day. As you may have noticed, this query does not precisely quantify the "number of domains that share the same IP address" because a domain name can have multiple IPs, and the original EXPOSURE paper was not clear about whether all the domains on all the IPs should be counted [5]. This query will actually output multiple counts for each domain name, one for each IP address that it uses. This is an example where having a common query language would make explicit analytic descriptions much easier, allowing researchers to more precisely describe their techniques.

4.2.3 Average TTL value

```
SELECT avg([ttl]), [name]
FROM dns_rr
WHERE [ttl] != NULL
GROUP BY [name]
UNTIL GLOBAL
    nextwindow([packets].[time], 86400)
```

This query is very similar to the first, except that it employs the `avg` (average) aggregate function instead of `count_distinct`. Another point of ambiguity in EXPOSURE is whether the TTL values should be counted for all types of resource records (as is done above), or just for A records.

4.2.4 Number of TTL value changes

```
SELECT count(), [name]
FROM (
    SELECT [name]
```

```

FROM dns_rr
WHERE [ttl] != NULL
GROUP BY [name]
UNTIL
  last([ttl]) != last([ttl], 2, true) &&
  last([ttl], 2, true) != NULL
)
GROUP BY [name]
UNTIL GLOBAL
  nextwindow([packets].[time], 86400)

```

This query uses a nested statement with two instances of the aggregate function `last`. In its first form, `last` just outputs the current tuple value. The second call to `last([ttl], 2, true)` actually outputs the second-to-last value (2 parameter) and persists across windows (true parameter). For the sequence {A, B, A}, the UNTIL statement will become true and flush the result after B arrives. Because the second call to `last` persists, it will hold on to the B value and output another change when the next A arrives. This is an example of aggregate functions that maintain state across windows.

4.3 DNS Tunnels

The DNS tunnel detection algorithm described in section 2.3 works by identifying responses that never have follow-up connections. Here are some key facts about this analytic:

- DNS responses may contain several A records, but only the first one will be likely to receive a connection. It is thus better to use the whole-message DNS schema rather than the individual resource record schema.
- We only want to count responses that do *not* have matching packets, so we need to use an EXCLUSIVE LEFT SINGLE JOIN.
- Because individual false positives may occur, we should apply a per-client threshold to unmatched responses, which will require a GROUP BY using the client as the key.

Here is a query that implements DNS tunnel detection:

```

SELECT
  [dns].[packets].[dstip] AS client,
  last([dns].[packets].[time]) AS start,
  first([dns].[packets].[time]) AS end
FROM dns EXCLUSIVE LEFT SINGLE JOIN ip_packet
  ON [answers].[aip] EQUALS [dstip]
  WINDOW [new].[packets].[time] -
    [old].[packets].[time] < 300
WHERE [dns].[answers].[aip] != NULL
GROUP BY [client]
UNTIL count() > 100
HAVING [end] - [start] < 3600

```

This query counts the number of DNS answers with an A-record IP address that have no matching packets

within a five-minute time window. It then groups those unmatched responses by their destination IP (the client who made the request) and applies a threshold of 100 responses. Note that the threshold is applied in an UNTIL clause. This makes it so that detection happens immediately when the threshold is reached, instead of having to wait for the end of a time window. The timestamps of the first and last responses can then be checked in the HAVING clause to make sure they occurred within some reasonable amount of time (one hour in this case). This query demonstrates the latency benefit from using UNTIL instead of a time- or count-based window like in existing streaming databases.

4.4 Phishing/Spam Detection

Section 2.4 describes a method for detecting spam and phishing e-mails based on filtering SMTP messages with "new" mail transfer agents (MTAs) and then counting the number of recipients to which the new MTAs send e-mail in the first 24 hours. Here is a Chimera query that implements this analytic:

```

CREATE VIEW mtasmtp AS
SELECT *
FROM smtp SPLIT [headers].
  filter{$.first() == 'Received'}.
  foreach{$.second().regex_extract
    ('.*by ([^ ]*)')} AS mta, midx;
SELECT
  merge([b].[headers].[To].split(','),
    [b].[headers].[Cc].split(','),
    [b].[headers].[Bcc].split(','))
  iterall{count_distinct($.strip())}
  AS recipient_count,
  [a].[mta] AS mta
FROM (
  SELECT *
  FROM mtasmtp
  WHERE unique([mta])
) AS a JOIN mtasmtp AS b
  ON [mta] EQUALS [mta]
WHERE [b].[packets].[time] -
  [a].[packets].[time] < 86400
GROUP BY [a].[mta]
UNTIL [recipient_count] > 50

```

This query contains a number of more complicated operations to achieve the desired result. The CREATE VIEW statement is used for the first time to set up a table of SMTP messages that are split by MTAs. The MTAs are extracted from "Received" headers in the SMTP message using a regular expression that searches for the string "by " and pulls out the following word.

In the first part of the select statement that follows, all of the destination e-mail addresses are extracted by splitting the "To", "Cc", and "Bcc" headers by commas, and then merging them into one list. The apply function

iterall is then used to pass each recipient through the aggregate count_distinct function to count the number of unique recipients for each MTA.

The sub-query in the left part of the join uses a stateful function unique in a WHERE clause. This means that it will use one global state instead of having a different state for each aggregate key. Furthermore, the unique function will accumulate values indefinitely. This function is different from distinct in a subtle way; it is designed to only output "new" values. It will silently add items to a Bloom filter during a learning phase at start-up, and then start generating output once a certain percentage of its inputs have already been seen. As the Bloom filter fills up, distinct will stop adding to it and create a new one. Once the new filter becomes full, the old one will be discarded and process will continue so that there are always two Bloom filters in use. With a Bloom filter it is possible to falsely label new items as not unique a small percentage of the time. This trade-off buys reduced memory utilization. The false match rate of a Bloom filter will depend on its size and the number of insertions that are made before rolling it over.

The final part of the query joins new MTAs with future e-mails that contain those MTAs, using the WHERE clause to cut off the count after 24 hours. The UNTIL clause will trigger as soon as the unique recipient count exceeds 50 and generate a final query output.

5 Bro Compiler Implementation

For this paper, we implemented a Chimera compiler that produces policies for the Bro event language [21]. While we have only implemented one specific target, it would be possible to extend the compiler to target other languages. The work that we describe in section 5.1 on translating a declarative query to an intermediate relational algebra will be applicable for all targets. The code generation phase, which is described in section 5.2, will depend on the target language.

5.1 Translation to Relational Algebra

Because Chimera is very similar to SQL, we begin the compilation process in the same way as traditional database systems: by parsing the query and translating it into an intermediate relational algebra. We used a simple YACC parser [16] and the syntax from section 3 to convert the original query into an abstract syntax tree (AST) representation. From there, the compiler translates the AST into a data-flow representation that loosely corresponds to relational algebra, which we call the *Chimera Core*. The Chimera Core operators are shown in figure 2. This step is performed using syntax-directed translation [2], wherein syntactic elements are converted into data-

```

source(source)
parser(parser)
split(exprlist, aliasitem)
projection(expr1, alias1, ..., exprn, aliasn)
selection(expr)
rename(newlabel)
join(labelleft, labelright, exprleft, exprright,
      exprwindow, joinkind, tablesize)
group(exprgroupby, expruntil, options, tablesize,
      aggexpr1, alias1, ..., aggexprn, aliasn)
output(dest)

```

Figure 2: Chimera Core language constructs

flow operators as shown in figure 3. During this process, the compiler uses a symbol table to map aliases to locations in the data-flow graph, but does not need to perform full data-flow analysis because all data-flow connections are explicit in the Chimera syntax.

CREATE VIEW	→	add alias to symbol table
SOURCE	→	source
<proto-name>	→	parser
SPLIT	→	split
<table> AS ...	→	rename
JOIN	→	join
WHERE	→	selection
GROUP BY ... UNTIL ... ORDER BY ... LIMIT	→	group
HAVING	→	selection
SELECT	→	projection
INTO	→	output

Figure 3: Summary of translation to Chimera Core

To illustrate translation from a Chimera query to the Chimera Core language, consider the following example:

```

SOURCE STDIN
SELECT avg([b].[z]) AS avgz
FROM dns AS a JOIN smtp AS b ON [x] EQ [y]
WHERE [a].[x] > 5
GROUP BY [a].[x]
UNTIL avgz > 3
INTO STDOUT

```

Figure 4 shows the data-flow graph that results from this example query. Using top-down syntax-directed translation, the first node emitted is a **source** node corresponding to SOURCE STDIN. The FROM statement is processed next. Because there is a JOIN, the compiler first translates the left and right tables, adding **parser** nodes to the **source**. The parser outputs are then fed through **rename** operators so that they can be referenced in the **join** operator, which combines them into a single data flow. Next, the data flows through a **selection**

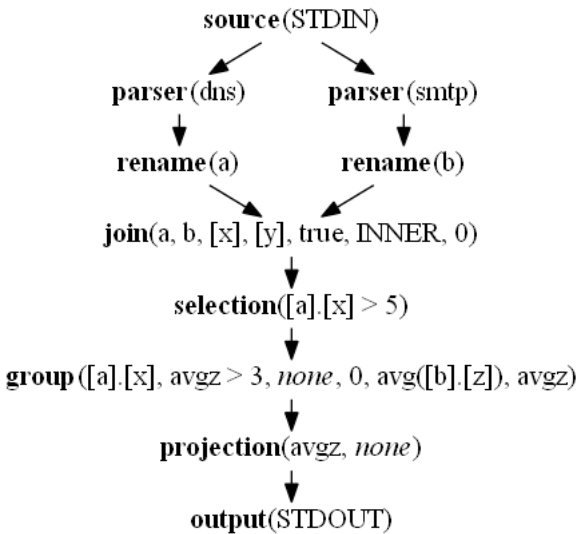


Figure 4: Chimera Core data-flow graph for example

operator that filters tuples using the WHERE expression. The tuples are then aggregated with a **group** operator, which also computes and adds aggregate expressions from HAVING and SELECT clauses to the data flow. Finally, expressions in the SELECT clause are extracted with the **projection** operator, and **output** sends data to standard output.

5.2 Code Generation

The next step in compilation is to translate the data-flow graph into Bro code. This process happens in two main stages: (1) type computation, and (2) event code generation. The event code generation step further depends on the implementation of user-defined functions, which written natively in the Bro language. Also note that data sources in Bro are specified on the command line, so the **source** operator is emitted as a shell script wrapper and not as part of the Bro language.

Type computation involves visiting each edge in the data-flow graph, determining the contents of tuples that flow through that edge, and then creating a record type for those tuples. Edges coming from operators that do not change the data – **selection** and **rename** – can be ignored during this pass. It would have been possible to use a table of the *any* type in Bro for tuples, or to create another dynamic data structure. We chose to use custom record types instead because they are better-documented and do not require modifying Bro internals.

After types have been defined for each input and output tuple, the compiler generates code for each node in the data flow graph in the form of an event handler:

- **parser** – This node adds a Bro protocol parser at the beginning of the file (if it does not yet exist) and defines an event handler that converts Bro protocol

events into output tuples.

- **split** – This node takes a tuple with a list expression and outputs a new tuple for each item in the list, which also includes all the original tuple items.
- **projection** – This node outputs an event handler that executes one or more expressions on each input tuple and assigns their results to an output tuple.
- **selection** – This node evaluates an expression on each input tuple and passes that tuple as output if the expression is true.
- **rename** – This node passes tuples through unchanged, but renames the event.
- **join** – This node stores tuples in a hash table keyed on their join expression values and later matches them against tuples from the other side of the join. When there is a match (or no match for OUTER joins), this node will generate a new output tuple with one or both elements. To support the WINDOW expression, we extended the Bro table data structure to expose its oldest element.
- **group** – This node maintains a hash table keyed on the GROUP BY expression value. The table contains state objects for each aggregate function, all of which have their *Iteration* routine called for each new tuple. When the UNTIL expression becomes true, this node calls each aggregate function’s *Evaluation* routine, adds the results to an output tuple, and then calls the aggregate *Termination* routines to flush the state objects.
- **output** – This special-purpose node outputs tuples in CSV format, or, if the tuple only has one packet, sends output to a PCAP file.

Some of the operator nodes take function expressions as arguments. As mentioned before, each function is written natively in Bro. A few functions, such as those that use a Bloom filter, also required some implementation in the Bro internal function (BIF) language. When a function is encountered during code generation, its definition is included in the Bro code and it is called with a standard Bro expression. Bro does not support method-style function calls using a syntax like $x(arg1).y(arg2)$, so these are re-written as $y(x(arg1), arg2)$. Apply functions are implemented by generating inline anonymous first-class function definitions, which are supported by Bro.

5.3 Example

Here we demonstrate Bro code generation with a simple example. In the interest of space, the example does not include **join** and **group** operators. Consider the following Chimera query:

```

SELECT [path]
FROM http-request
WHERE [method] == "GET"
  
```

This query extracts the path from all HTTP GET requests. It translates to the following data-flow graph, where each operator sends data to the next:

```
10: source(STDIN)
11: parser(http-request)
12: selection([method] == "GET")
13: projection([path], none)
14: output()
```

Finally, this is compiled down to the following Bro script. Note that Bro splits up the HTTP headers and body into multiple events. To have everything available in one tuple, we also add an event handler for `http_all_headers` that saves the headers in the session table, which is omitted here to save space.

```
@http-reply
type http_request_type: record {
    method: string;
    path: string;
    headers: listmap;
    body: string;
    packetlist: packetlist_type;
};
type l3_type: record {
    v1: string;
};
event l3(t: l3_type) {
    print t$v1;
}
event l2(t: http_request_type) {
    local out: l3_type;
    out$v1 = t$path;
    event l3(t);
}
event l1(t: http_request_type) {
    if (!(t$method == "GET")) return;
    event l2(t);
}
event http_message_done(c: connection, ...) {
    local t = http_request_translate(c);
    event l1(t);
}
```

6 Evaluation & Future Optimizations

We have presented the implementation of a compiler that translates Chimera queries into the Bro event language. Because functionality was our primary focus, we have not yet implemented any performance optimizations. However, there are many areas that have potential for optimization. Here we evaluate the compiler's processing performance in its current unoptimized form and discuss opportunities for future performance optimization. This section does not evaluate memory utilization because it is highly dependent on the particular query,

desired window size, and data rate of the connection. Windows for JOIN and GROUP BY operations can be scaled according to the operating environment and analytic needs.

6.1 Performance Measurement

The performance measurements in this section were taken by processing a 2 GB PCAP file (stored on a ram disk) with Bro and recording the execution time. The PCAP file was generated by capturing traffic at a U.S. government network gateway, so it includes data from a variety of protocols. It contains approximately 81k HTTP, 58k SMTP, and 32k DNS messages.

To test the compiler's performance, we compare the Bro event code generated by the Chimera compiler to hand-written Bro code that implements the same functionality. For example, the Bro code in section 5.3 could be written by hand as follows:

```
@http-reply
event http_request(c: connection,
    method: string, original_URI: string,
    unescaped_URI: string, version: string) {
    if (method == "GET")
        print original_URI;
}
```

This shorter implementation has three optimizations:

1. Data is not copied into new record types.
2. Events with only one handler are evaluated inline.
3. An earlier event handler (`http_request`) is used because the headers and body are not needed.

Our first experiment tests the effect of each optimization by applying them one-by-one to the section 5.3 example. We ran each configuration 30 times against the test data. Table 3 summarizes our results. Bypassing data copying saves about 1.5% execution time. Inlining event code makes no significant difference in this case. Switching to a single earlier handler saves another 1.5%, for a 3.0% overall speed-up. While the difference between current compiled code and hand-written code is noticeable, it does not have a major impact.

Table 3: Execution times with different optimizations

Configuration	Base	Opt-1	Opt-1+2	Opt-1+2+3
Avg. Time (s)	14.21	14.00	14.01	13.79
Std. Dev. σ (s)	0.084	0.083	0.074	0.081
Speed-up (%)	-	1.5%	1.4%	3.0%

For the next part of our evaluation, we tested a selection of more complicated queries from sections 4.1, 4.2.1, and 4.4. We ran the queries as they were compiled to Bro code, and after they were optimized by hand by eliminating unnecessary copying and event handlers. Because a Bro implementation of side-jacking was already available for query 4.1 [26], we used that as a basis for

Table 4: Execution times for different queries, with and without hand-optimization

Query	Base Time (s)	Optimized (s)	Speed-up (%)
4.1	15.64 ($\sigma = 0.081$)	15.48 ($\sigma = 0.067$)	1.1%
4.2.1	8.81 ($\sigma = 0.085$)	8.72 ($\sigma = 0.021$)	0.96%
4.4.	2.77 ($\sigma = 0.027$)	2.75 ($\sigma = 0.019$)	0.79%

comparison. We optimized the other two queries ourselves. Table 4 shows the results averaged across 30 runs for each measurement. Much like the first experiment, the overhead added by extra copying and event handlers only has a minor impact on overall performance, increasing running time by about 1%. Though the Bro code generator could benefit from some optimizations, in its current form it generates code that is almost exactly equivalent to hand-written code for these real-world scenarios.

6.2 Other Optimizations

The previous section discussed optimizations in the code generator related to event and data handling. There are also opportunities for optimization at the relational algebra level before any code is generated, and in the analysis logic. Prior work on query optimization for databases [11, 15, 25] is directly applicable here because it operates on relational algebra that is almost exactly the same as the Chimera Core language. One common trick is to break up selection operators into sub-expressions and put the cheapest one with the greatest data reduction first. Similarly, selection operators that occur after joins can have sub-expressions that do not depend on both join sides pushed before the join, thus reducing the number of items in the join table. Finally, any nodes that duplicate one another, including parsers, can be merged together. We plan to incorporate all of these optimizations in future versions of the Chimera compiler.

Another area of optimization that we plan to explore is improving the actual analysis logic. For example, an ordered EXCLUSIVE RIGHT JOIN is effectively an existence check; there is no need to actually store left tuples in the join table because they will never be emitted as output. Going further down this route, an existence check can be approximated efficiently using a Bloom filter. For analytics where complete precision is not necessary, an exclusive right join could be implemented with a windowed bloom filter.

Finally, queries in the Chimera language lend themselves well to parallel processing using a map-reduce model. Tuples can be mapped to a processing node using their join or group key right before each join or group operator in the data-flow graph. Each node will then execute the operator to perform the reduction. Global aggregates can be computed by extending aggregate functions to have a merge routine that combines partial answers as discussed in section 3.4.3 (though not all aggregates can

be merged efficiently). We plan to extend the Chimera compiler in the future to automatically produce code that can run in a parallel environment.

7 Related Work

There has been a lot of prior research on streaming database systems. STREAM [3, 19] and Aurora [1] were pioneers in this area. Following initial work, others have developed improved techniques for windowed query evaluation [17] and load shedding [24]. An effort has also been made to create a standard for streaming SQL [14] that accounts for semantic differences between various systems. Others have focused on window specification semantics for streaming queries [6, 20].

Streaming database research is useful and serves as a basis for ideas in this paper, but Chimera goes beyond what has been done in prior work. It is the first language designed to translate into external intrusion detection frameworks like Bro. Chimera also adds two new capabilities that are very important for handling network traffic. The first is support for structured data types, which includes the new SPLIT operator and apply functions. The second major contribution is the addition of dynamic window conditions using the UNTIL trigger for aggregates, and the WINDOW condition for joins. This gives the query writer full control over window boundaries, allowing for immediate response after a detection threshold has been reached, rather than having to wait until the window expires as with traditional fixed window specifications.

One project that is related to Chimera is Gigascope [7]. Gigascope is a platform for performing network traffic analysis that uses an SQL query language. However, Gigascope is different from Chimera in a few key ways. First, it is a vertically integrated query language and platform for performing analysis. Its language is therefore tied to the implementation and has not been adapted to target other platforms. Chimera, on the other hand, is designed to be implementation-agnostic and serve as a general-purpose language for network processing. Furthermore, Gigascope's SQL query language has the same limitations as traditional streaming database systems. As far as we are aware, it only supports flat schemata, which prevents it from properly handling structured data. It also uses standard window specifications instead of dynamic window boundaries, which limits flexibility for join and aggregate queries.

IBM's Stream Processing Language (SPL) [13] is also related to Chimera. Unlike Chimera, SPL is not entirely declarative. Its *logic* clause uses procedural code and one must specify data flow paths to define analysis logic. SPL does support dynamic window boundaries using a

punct type of *tumbling* window in which boundaries are set by messages from upstream operators. These operators can use arbitrarily complex logic to generate *punct* messages, which in theory provides the same power as dynamic window conditions in Chimera, but in a less concise manner. We view SPL as largely analogous to the Bro event language, except that it is data-flow-based rather than event-based. It is a powerful lower-level language that provides greater control, but suffers from the same problems of being less concise and more complicated than Chimera. We imagine that it would be possible to adapt the Chimera compiler to generate code for SPL in the future.

There are a number of procedural language extensions for traditional relational databases, including PL/SQL [10], Transact-SQL [9], and PL/pgSQL [18]. These procedural languages offer powerful constructs like conditional statements and looping. PL/SQL also offers array data types, and arrays can be simulated with delimiter-separated strings. These languages do not directly offer apply functions or SPLIT operations, but the same result can be achieved (albeit not as elegantly) with nested queries. While it is possible to express Chimera queries and data types in these procedural programming languages (they are Turing complete), we believe that the Chimera language is more intuitive for processing structured network protocol traffic. Chimera also goes further by running in a streaming environment and translating to the Bro event language.

The idea of having a high-level language that translates into low-level policy has been applied previously to other areas. One particularly relevant example is for router and firewall configurations [4, 12]. Low-level firewall policies precisely describe the mechanism for filtering traffic in a level of detail that goes beyond the high-level goals behind them. This makes firewall configuration policies difficult to read and error-prone. Previous work by Guttman et al. and Bartal et al. distills out the underlying security goals into a high-level language, and then translates that into low-level policies, thus eliminating the need for administrators to write those low-level policies. Chimera is applying the same idea of separating policy from mechanism, but for a much different different domain.

8 Conclusion

In this paper, we introduced Chimera, a new query language for processing network traffic. Chimera effectively separates policy from mechanism, leading to concise queries that are independent of implementation. Chimera is based on a streaming SQL syntax, which it extends by adding structured data, first-class functions,

and dynamic window boundaries. These additional features allow Chimera to better handle complex network traffic analysis tasks.

This paper looks at example scenarios to motivate Chimera's design and demonstrate its utility. Two of the examples – side-jacking and DNS feature extraction – are taken from prior work. Writing Chimera queries for these examples showed how they are more compact than lower-level Bro event code and more precise than human language descriptions. The other two scenarios – detecting DNS tunnels and identifying spam/phishing e-mail – demonstrated some of Chimera's more advanced capabilities and showed how it can be used to express complex analysis logic with concise declarative queries.

Finally, we presented the design and implementation of a compiler that translates Chimera queries into the Bro event language. This compiler works in two phases by first transforming an abstract syntax tree into a data flow representation, and then translating that representation into Bro event code. We tested the compiler's output against hand-optimized code for several queries and showed that it is only 3% slower in the worst case. This experiment highlighted opportunities for optimization by eliminating unnecessary copying and event handlers, but also showed that the Compiler generates code that is almost as efficient as hand-written code in its current form. In the future, we hope to implement these optimizations and also incorporate optimizations at the relational algebra level so that Chimera obviates the need to write low-level code for network analysis logic.

References

- [1] ABADI, D. J., CARNEY, D., ĀĖTINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. Aurora: A new model and architecture for data stream management. *The VLDB Journal* 12, 2 (2003).
- [2] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*. 1986.
- [3] ARASU, A., BABU, S., AND WIDOM, J. CQL: A language for continuous queries over streams and relations. *Lecture Notes in Computer Science* 2921, 123–124 (2004).
- [4] BARTAL, Y., MAYER, A., NISSIM, K., AND WOOL, A. Firmato: A novel firewall management toolkit. In *IEEE Symposium on Security and Privacy* (1999).
- [5] BILGE, L., KIRDA, E., KRUEGEL, C., AND BALDUZZI, M. Exposure: Finding malicious domains

- using passive dns analysis. In *Network and Distributed System Security Symposium* (2011).
- [6] BOTAN, I., DERAKHSHAN, R., DINDAR, N., HAAS, L., MILLER, R. J., AND TATBUL, N. SECRET: A model for analysis of the execution semantics of stream processing systems. *Proceedings of the VLDB Endowment* 3, 1–2 (2010).
- [7] CRANOR, C., JOHNSON, T., SPATASCHEK, O., AND SHKAPENYUK, V. Gigascope: A stream database for network applications. In *2003 ACM SIGMOD International Conference on Management of Data* (2003).
- [8] CROCKFORD, D. The application/json media type for javascript object notation (json). RFC 4627, Internet Engineering Task Force, July 2006.
- [9] DARNOVSKY, M., AND BOWMAN, G. Transact-sql user’s guide. Tech. Rep. 3231-21, Sybase, Inc., 1987.
- [10] FEUERSTEIN, S. *Oracle PL/SQL Programming*, third ed. O’Reilly & Associates, Sebastapol, CA, 2002.
- [11] GRAEFE, G. The volcano optimizer generator: Extensibility and efficient search. In *ICDE* (1993), pp. 209–218.
- [12] GUTTMAN, J. Filtering postures: Local enforcement for global policies. In *IEEE Symposium on Security and Privacy* (1997).
- [13] HIRZEL, M., ANDRADE, H., GEDIK, B., KUMAR, V., LOSA, G., MENDELL, M., NASGAARD, H., SOULÁL, R., AND WU, K.-L. Streams processing language (spl). Tech. Rep. RC24897, IBM, 2009.
- [14] JAIN, N., MISHRA, S., SRINIVASAN, A., GEHRKE, J., WIDOM, J., BALAKRISHNAN, H., ĀGETINTEMEL, U., CHERNIACK, M., TIBBETTS, R., AND ZDONIK, S. Towards a streaming SQL standard. *Proceedings of the VLDB Endowment* 1, 2 (2008).
- [15] JARKE, M., AND KOCH, J. Query optimization in database systems. *ACM Computing Surveys*, 2 (1984), 111–152.
- [16] JOHNSON, S. C. Yacc: Yet another compiler-compiler. Tech. Rep. 32, Bell Laboratories, 1975.
- [17] LI, J., MAIER, D., TUFTE, K., PAPADIMOS, V., AND TUCKER, P. A. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *Information Systems* 34, 1 (2005).
- [18] MONJIAN, B. *PostgreSQL: Introduction and Concepts*. Addison-Wesley, Boston, MA, 2000.
- [19] MOTWANI, R., WIDOM, J., ARASU, A., BABCOCK, B., BABU, S., DATAR, M., MANKU, G., OLSTON, C., ROSENSTEIN, J., AND VARMA, R. Query processing, resource management, and approximation in a data stream management system. Technical Report 2002-41, Stanford InfoLab, 2002.
- [20] PATROUMPAS, K., AND SELLIS, T. Maintaining consistent results of continuous queries under diverse window specifications. *Information Systems* 36, 1 (2011), 42–61.
- [21] PAXSON, V. Bro: a system for detecting network intruders in real-time. *Computer Networks* 31, 23–24 (1999), 2435–2463.
- [22] RILEY, R. D., ALI, N. M., AL-SENAIDI, K. S., AND AL-KUWARI, A. L. Empowering users against sidejacking attacks. In *ACM SIGCOMM 2010 conference* (2010).
- [23] ROESCH, M. Snort – lightweight intrusion detection for networks. In *USENIX LISA – 99 Conference* (1999).
- [24] TATBUL, N., AND ZDONIK, S. Window-aware load shedding for aggregation queries over data streams. In *32nd International Conference on Very Large Data Bases* (2009).
- [25] V. MARKL, G. M. LOHMAN, V. R. Leo: An automatic query optimizer for db2. *IBM Systems Journal* 42, 1 (2003).
- [26] VALLENTIN, M. Taming the sheep: Detecting sidejacking with bro. <http://matthias.vallentin.net/blog/2010/10/taming-the-sheep-detecting-sidejacking-with-bro/>, 2010.

New Attacks on Timing-based Network Flow Watermarks

Zi Lin

University of Minnesota
lin@cs.umn.edu

Nicholas Hopper

University of Minnesota
hopper@cs.umn.edu

Abstract

A network flow watermarking scheme attempts to manipulate the statistical properties of a flow of packets to insert a “mark” making it easier to detect the flow after passing through one or more relay hosts. Because an attacker that is willing to tolerate delay can (nearly) always eliminate such marks, recent schemes have concentrated on making the marks “invisible” so that a passive attacker cannot detect the presence of the mark. In this work, we argue that from a system’s perspective, security against passive detection is insufficient for successful traffic analysis. We introduce a stronger, but feasible attack model (a *known/chosen flow* attacker) and a second security goal (security against *copy attacks*) and argue that security against both of these attacks is required for successful traffic analysis. We also demonstrate successful attacks against two recent watermarking schemes, RAINBOW and SWIRL, and show how considering these stronger attacks can aid in the design of passive detection attacks against each as well.

1 Introduction

Active traffic analysis, so called network flow watermarking, is the practice of manipulating the timing of a network flow so that the same flow, relayed by one or more intermediate hosts, can later be recognized. This technique has been the subject of increased interest in the past decade, because it requires low computational and communication cost while providing high accuracy in linking traffic flows. In these schemes, the packet timings of a network flow are modified, usually by buffering and delaying, to contain a distinctive pattern. If the pattern is later detected in another flow, we can conclude the two flows are the same with high probability. Flow watermarking is one of the most effective methods both for breaking anonymous communications systems [14, 15, 18, 7] and detecting network intruders launching a stepping stone attack [16, 12, 8, 7].

In contrast to the variety of schemes proposed, there is a relative lack of systematic study on attacking watermarking schemes. Since an active attacker can arbitrarily delay packets, thus destroying any watermark, recent schemes focus on how to evade passive detection and

thus become “invisible” [15, 18, 7, 8]. However, many of these works consider a very limited attack model. Attackers are usually assumed to have access only to flows that comes from a black box in which a watermarker may have inserted marks to some of the flows. This assumption however, is often unrealistic: for example, in both cases mentioned above the adversary attacking the watermark has access to additional information about the marked flow. Furthermore, some adversaries (such as an anonymity network) may be better served by increasing the number of “unmarked” flows that appear to be marked, rather than trying to detect or remove a mark imperfectly.

Meanwhile, without a systematic view, it is often unfair to compare different detection attacks directly. In [11] Peng, Ning and Reeves studied how a stepping stone attacker, as a chosen flow attacker, could inject and analyze flows to detect the presence of a watermark and even replicate the parameters used by the watermarking system. Hereafter we refer to this attack as the PNR attack. In [9], Kiyavash et al. propose the *multi-flow attack* (MFA), which exposes a watermark by aligning multiple flows carrying the same watermark. A MFA can be launched by a single router in the network and is thus widely applicable. Very recently, Luo et al. [10], describe BACKLIT, a unique threat model in which the attacker acts as a known/chosen flow attacker and is thus able to detect state-of-the-art watermarking schemes. It is worth noting that the PNR attack model is stronger than BACKLIT, which is in turn stronger than the MFA model, so it is not surprising that their performance strengths also follow that order. On the other hand, MFA is more applicable than BACKLIT, which in turn is more applicable than PNR.

To address these concerns, we formalize two threat models for network flow watermarks. The first model, the *chosen flow attacker*, captures the capacities of a network intruder. This attacker may observe or even manipulate the input to the black box of a watermarker. In this case, packet delays due to deliberate watermarking and/or normal network processing become visible; and separating marked flows from unmarked flows becomes considerably easier. We apply this model to two recent watermarking schemes, RAINBOW [8] and SWIRL [7].

Both schemes use delaying as the basic operation to insert marks and are thus vulnerable to relatively straightforward chosen flow detection attacks.

Evaluating under the chosen flow attack may also help to develop ideas for new passive detection algorithms in the second threat model, the *isolated attacker*. For example, our initial chosen flow attacks on RAINBOW, which are based on testing jitter irregularity using cosine distance and histograms, led us to design a new passive detection attack based on testing the irregularities in the distribution of inter packet delays (IPDs). Similarly, a simple multi-flow chosen flow attack on SWIRL provides several important insights on a new multi-flow passive detection attack against SWIRL.

We implement and test our new attacks through experiments with real-world network traces. We show that chosen flow attacks can perfectly detect the watermarks with 100% recall and no false positives, outperforming the BACKLIT attacks. Our local, passive attacks on RAINBOW and SWIRL also achieve high ROC scores of ≥ 0.92 ; in some cases, ROC scores of 1.0 can be achieved.

Finally, we also introduce non-parametric *copy attacks*, which transfer marks between flows and eventually confuse traffic analysis without the knowledge of watermark secret keys and parameters. To our knowledge, this type of attack has not been studied before; yet every timing-based flow watermark is vulnerable due to the naïve attack that buffers enough traffic to mimic the inter-packet delays of marked flows. However, some schemes are vulnerable to less heavy-handed copy attacks. For example, the design of SWIRL allows us to demonstrate a very cost effective copy attack against it.

1.1 Paper outline

We briefly survey related work in section 2. In section 3, we establish the threat model for network flow watermarking schemes, identify new detection modes/attacks and a novel implementation of active copy attack. We describe our new detection attacks on RAINBOW and SWIRL, followed by evaluation on real-world datasets in section 4. We also present a detailed description of copy attacks on the aforementioned schemes along with experimental evaluations in section 5. Finally, we discuss possible defenses against these attacks and general defense strategies under our threat models in section 6.

2 Related Work

2.1 Stepping Stone Detection

Network intruders usually tunnel their attack traffic through one or more intermediate relays as “step-

ping stones”, making the traffic origin hardly traceable. Within large enterprise networks, stepping stones are good candidates of compromised hosts. Network administrators therefore take stepping stone detection as part of their security monitoring routines.

Detecting stepping stone is usually done by linking outgoing flows with incoming ones. Often the intruder encrypted their tunnel (for instance by SSH). As a result, only packet counts, sizes and timings are available for flow characterization. By passive recording of these flow features, many schemes have studied the problem of linking streams [19, 4, 17, 4, 2, 6]. Since flow characteristics could be affected by padding schemes, packet retransmission and repacketization, and network jitter, successful passive stepping stone detection needs large number of observations, which in turn incur large overhead in both storage and computation. To address these efficiency issues, active approach is proposed as watermarking [16]. We will next briefly review the literature of flow watermarking schemes.

2.2 Network Flow Watermarking Schemes

There are two entities involved in flow watermarking, the encoder and the decoder. Both are typically boundary routers and share some common states. The encoder embeds a timing watermark to each incoming flow by introducing timing distortion (usually through delaying specific packets). At the other end, the decoder examines each outgoing flow to see whether it displays the unique mark and thus identify a potential stepping stone.

Several packet delaying schemes [16, 14, 8] intend to embed message bits by introducing distinctive network jitter. Those bits can be easily picked up by the decoder but they look unintentional to other routers. In [16, 14], skewness in jitter distribution, caused by delaying selected individual packets, is directly manipulated/measured. In RAINBOW[8], jitter distortion is expressed as an artificial jitter sequence (up to a few thousands in length), which is orthogonal to natural observed jitter in the linear space. This unique jitter, compounded together with normal network noise, can still be recognized by the decoder, using the inner product.

Interval-based watermarking schemes [12, 15, 18, 7] divide a flow into a series of time intervals and embed bits by manipulating the packet timing characteristics within each interval. Such approaches focus on intervals rather than packets, and are thus generally more resilient to packet insertion, losses and repacketization.

2.3 Attacks on Watermarking

In 2006 Peng et al. presented the PNR attack for stepping stone attackers [11]. The attack is designed to re-

cover the secret keys and system parameters of the particular scheme by Wang et al. [16]. By sending packets with controlled timing, uncommon extra delays can be detected by a variety of statistical and data mining tools. They also designed a duplicate attack to confuse the decoder by raising the false positive rate. The duplicate attack mentioned in [11] is simply repeat the watermarking with the extracted watermark parameters and is essentially different from the non-parametric copy attack we discuss here.

The Multi-flow attack (MFA), a passive detection attack recently proposed by Kiyavash et al. [9], demonstrates that an alignment of multiple marked flows shows an unusual synchronized pattern of busy and idle periods. This pattern is strong enough to conclude the presence of a watermark. MFAs show the potential of passive detection attacks and have helped motivate the design of newer watermarking schemes like RAINBOW and SWIRL [8, 7], that claim to resist these attacks.

Most recently, Lou et al. [10] studied how to expose watermarks in BACKLIT, an attack scenario in which the watermark encoder manipulates return traffic from a server while the attacker acts as a traffic relay between the client and the server. The watermark attacker can probe irregularities by comparing “clean” forward flows and “marked” backward ones. By doing so, BACKLIT gains extra knowledge similar to chosen flow attacker and is able to expose RAINBOW, SWIRL and other timing-based watermarks. The success of BACKLIT, however, relies heavily on the specific threat model.

3 Threat Model of Network Flow Watermarking

In this section we first discuss the performance goals of network flow watermarking schemes. Then we briefly describe two threat models for watermarking schemes and define essential security properties that watermarks should achieve against adversaries. We note that anonymity systems and stepping stone intruders generally possess different capabilities and it is worthwhile to model them as different adversaries.

3.1 Performance Requirements

Active traffic analysis techniques, such as timing-based watermarking, aim to link network flows passing through one or more relay hosts efficiently and with high precision and robustness. This requires watermark detection to achieve both low false negative rates (FNR) and low false positive rates (FPR), even in the presence of network jitter and other distortion noise.

Low FNR requires the watermark not be easily erased by natural timing distortion. Detecting watermarked

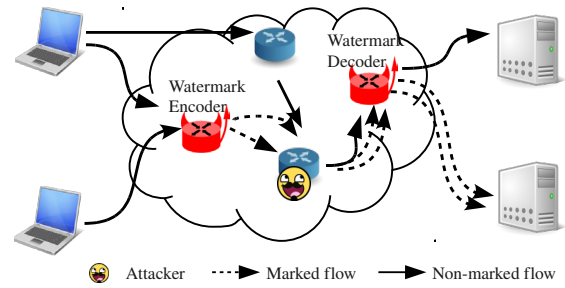


Figure 1: Isolated adversary: Accessible to output streams of the watermark encoder and input stream of the decoder.

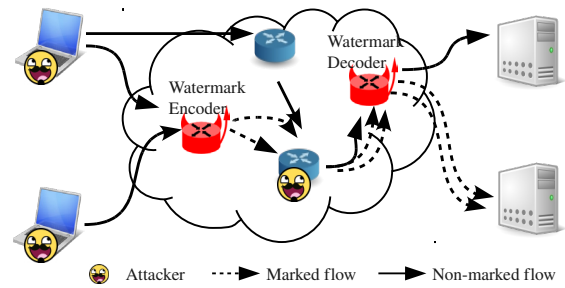


Figure 2: Chosen flow adversary: Inputs of the encoder are accessible too, in addition to outputs of the encoder and inputs of the decoder.

flows within a large network, although with keys, is sometimes challenging. Timing distortion such as delay by congestion, packet reordering, packet loss, and re-packetizing are not uncommon. To achieve low FNR, usually multiple copies of a watermark are inserted into different flow locations, so that timing distortion will not affect the majority of the marks.

When facing a determined active adversary, FNR can be arbitrarily high because any timing-based mark can be erased through drastic measures. In an attempt to preemptively remove any timing-based watermark, active attackers can drastically change the timing by adding dummy packets, introducing large delays and/or sending packets in batches. Although effective, these active counter-measures against flow watermarking induce high costs that are typically unacceptable to the attacker, especially for delay-intolerant applications like Tor and SSH stepping-stones. Therefore, active attackers generally would prefer to appear passive and reactively launch counter-watermark attacks.

The requirement of low FNR against adversaries triggers the goal of passive invisibility: if watermarks ap-

pear invisible to passive attackers, active countermeasures may not be employed. Still, there exists a trade-off between low FNR and invisibility. The more watermarks that are embedded in the network flow, the more signals can be potentially picked up by the attacker.

On the other hand, low FPR is required so that non-watermarked flows are not frequently mistaken for marked ones; the advantage of watermarking over passive traffic analysis diminishes quickly as FPR grows.

Low FPR against adversarial manipulation, however, has not attracted much attention. Attackers that recovers specific watermark parameters can duplicate watermarks on different flows, "confusing the detector" [11, 9]. However, given the lack of knowledge of secret keys and parameters, blindly manipulating a benign flow into a watermarked flow is perceived to be hard for adversaries. We introduce a novel implementation of active duplication attack designed to increase FPR without the knowledge of any watermark parameter, called the *best-effort copy attack*. Busy Tor relays can use copy attacks to replicate and spread watermarks on all outgoing circuits and as a result, considerably increase FPR.

3.2 Types of Adversaries

The adversary against watermarking is assumed to control at least one host that relay the traffic between the encoder and decoder. Such assumption is realistic for both stepping stone intruders and anonymity system relays. The concrete threat model can be categorized into two classes: *Isolated adversary* illustrated in Fig. 1 and *chosen flow adversary* in Fig. 2. Adversaries in both threat models can be either pure observers (passive) or traffic manipulators (active).

Anonymity networks such as Tor are generally isolated active adversaries. Although Tor relays can manipulate packet timing, they seldom do so because their ultimate goal is to forward traffic as soon as possible. However that doesn't mean Tor cannot do anything actively about watermarking. As long as timing watermarks don't incur a high delay, a Tor relay could inject them and even further exchange watermarks between different circuits as we will show later.

The seemingly strong chosen flow adversaries are not uncommon. A stepping-stone intruder, for example, is capable of sending traffic at will. And it is usually true that those attackers get root privilege on "stepping stones" and are therefore able to observe and manipulate the packet timing. A careful stepping-stone intruder can set up trial connections to test the existence of a watermark before actually using these "owned" workstations as stepping stones. The PNR attacker [11] is a good example.

3.3 Invisibility

In a nutshell, invisibility is defined as the ability to distinguish watermarked flows from non-watermarked ones. More formally, we define the Invisibility Game, played by an adversary (shown in Figure 3): Consider two sets of network flows S_0 and S_1 ; both of them are generated from the same distribution on flows. Flows in S_1 are manipulated by the watermarker while ones in S_0 are not. Both S_0 and S_1 are affected by similar network jitter. Now a random $i \in \{0, 1\}$ is generated by a fair coin flip, the adversary is given one or more flows from S_i and she outputs i' , she wins the game if $i = i'$. A watermark scheme is 'invisible' if no adversary can win the Invisibility Game with probability non-negligibly greater than "1/2". We further extend the definition of Invisibility Game to match up the ability of specific adversaries.

3.3.1 Invisibility with Isolated Adversaries

(Encoder) Output-only Detection To detect the presence of a watermark, there is little extra information the isolated adversary can obtain other than flow timing. Generally, we call such detection *Output-only detection* because the adversary only has access to (possible) outputs of the watermark encoder. Although such adversaries appear to be the weakest, their detection capabilities are still poorly understood.

Isolated passive invisibility has been the main focus in the literature to date. Various primitive analyses, such as entropy tests and distribution tests are utilized to examine the invisibility. These tests are usually carried out on individual flows. More powerful attacks come from the novel idea of collectively examining flows in S_1 or S_0 .

Multi-flow Attacks. The Multi-Flow Attack (MFA) was introduced by Kiyavash, Houmansadr and Borisov [9] to show that previous interval-based watermarking schemes [12, 15, 18] lack invisibility when multiple network flows carrying the same watermark are carefully aligned. The aggregated histogram of packet frequencies will show repeated cleared/crowded intervals that can rarely happen without watermarking. The MFA shows the potential power of a passive isolated attacker.

We argue there is still room for many more intelligent detection attacks. In this work, we demonstrate some effective output-only detection attacks against state-of-the-art watermarking systems, RAINBOW and SWIRL, which have taken MFA resistance into account.

3.3.2 Invisibility with Chosen Flow Adversaries

Known Flow Attack. In this attack, chosen flow adversaries can choose to observe arbitrary flows; flows are

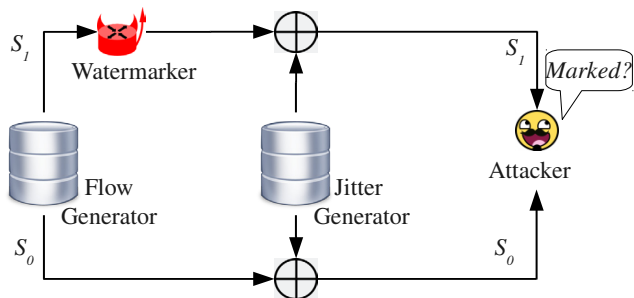


Figure 3: The Invisibility Game

examined by pure observers before and after they are watermarked. We modify the invisibility game to reflect the fact that now adversaries also have access to flows before they pass through the watermarker. The adversary has the ability to compute the actual jitter imposed by the network and the watermarker. This brings opportunities to detect jitter-based watermarking schemes. In particular, we will show how RAINBOW, a jitter-based watermarking scheme, is vulnerable to such an adversary in section 4. We note that Houmansadr *et al.* [8] discuss global invisibility of RAINBOW in terms of the Kolmogorov-Smirnov test on inter-packet delays and jitter vectors. Unfortunately, K-S tests assume no *a priori* knowledge of the data distribution and therefore underestimate the power of adversaries against RAINBOW, limiting the usefulness of the result. In particular, we find discriminators that work almost perfectly to detect RAINBOW even when its parameters are carefully selected to avoid detection.

Chosen Flow Attack. In this scenario, adversaries can inject flows with a specific timing pattern and observe the distortion possibly added by the watermarker. The invisibility game with such an adversary gives the adversary the ability to intervene with flow generation. An example of a chosen flow attack is studied in [11]: when sending packets with known timing, extra delays caused by watermarks are distinguishable from normal network jitter. None of the known network flow watermarking schemes will resist this attack. For instance, we will show how SWIRL is visible under chosen flow attacks.

3.4 Active Copy Attack Resistance

As shown in Figure 4, the purpose of a copy attack is to confuse the decoder between flows from S_0 and ones from S_1 . To our knowledge, copy attacks without knowing specific watermark parameters have not been studied for network flow watermark schemes. However, the concept of copy attacks on watermarking schemes is not new. Adelsbach *et al.* [1] introduced the notion of *pro-*

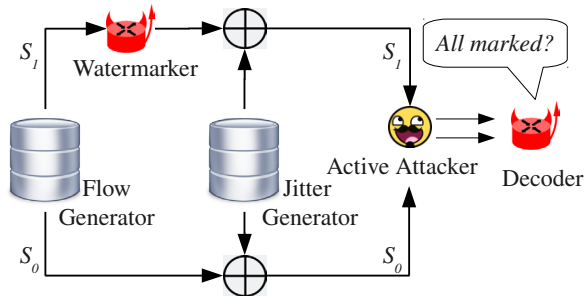


Figure 4: The Copy Attack Game

ocol attacks on multimedia watermarks where, for instance, an attacker can copy a watermark from one copy of a digital product to another, thus causing confusion and difficulties in the authorship/ownership dispute process. Peng *et al.* [11] designed a duplicate attack which can be seen as a copy attack by chose flow adversaries. Unlike the previous work, we show that it is possible to carry out copy attacks without knowing the secret key or concrete system parameters. naïvely, a straightforward replay of packet timing works for all timing-based watermarking schemes, including SWIRL and RAINBOW. However, we design a copy attack on SWIRL that is much simpler and extremely cost-effective. We show the effectiveness of this attack in section 5. With an active copy attack, two flows can replicate their watermarks to each other so the decoder is confused in flow linking.

4 Detection Attacks Based On Flow Characteristics

In this section we present detection attacks from different levels of adversaries. We demonstrate specific attacks on two exemplar schemes: RAINBOW and SWIRL. Following a brief recap of each scheme, we present a detection attack by a global adversary. We then show how the attack can be extended to work with a isolated adversary. All attacks are simulated and evaluated on CAIDA datasets [13].

4.1 Attack Implementation

We implement both watermarking schemes and attack algorithms in C++. To simulate the flow generator and jitter generator, we draw flows and jitter vectors from real-world network traces. Our simulation setup closely follows that of [7].

The network trace data are collected by the CAIDA project from its equinix-chicago OC192 link in January 2009 [13]. The dataset contains network flows that traverse in both direction of the link during a 4-hour pe-

riod. We selected SSH flows (destination port 22) because stepping-stone attacks are usually conducted over SSH, and got a total pool of 33 flows and 2.78 million packets. The packet rate ranges from 2 pps to 180 pps.

To simulate normal jitter caused by network delays, we adopt the RTT (round-trip time) measured between PlanetLab nodes [3] by Houmansadr and Borisov in their SWIRL work. In each simulation, a random trace of round-trip delays is chosen and applied to the watermarked flow.

4.2 RAINBOW

In the RAINBOW watermarking scheme [8], the watermark encoder and decoder share a database (DB) which records packet inter-arrival timing. In addition, they share a watermark key $w = [w_1, w_2, \dots, w_n]$. The components of w take binary values of $+a$ and $-a$ uniformly:

$$w_i = \begin{cases} a & \text{with prob. } 1/2; \\ -a & \text{with prob. } 1/2. \end{cases}$$

For each individual incoming network flow, the encoder computes and stores the packet inter-arrival timing in DB, then inserts w as extra jitter. Considered the packet arrives at time $[t_1, t_2, \dots, t_{n+1}]$, the inter-packet delays (IPDs) are $v = [t_2 - t_1, t_3 - t_2, \dots, t_{n+1} - t_n]$. Now the flow is marked to carry the inter-arrival time as $v + w$, i.e. each inter arrival time is lengthened or shortened by a milliseconds. In case $v_i + w_i < 0$, w_i is ignored to avoid packet reordering, which would severely degrade the watermark if it happened too often. The details are shown in Algorithm 1.

On the other end, for each outgoing flow with inter arrival time vector v' , the decoder computes a jitter vector $d = v' - v$ for each corresponding v recorded in the DB, and computes the cosine similarity score between d and w :

$$\cos(d, w) = \frac{\langle d, w \rangle}{\|d\| \|w\|}.$$

The detection algorithm is shown in Algorithm 2.

If v' is indeed the marked version of v , then we have $d = w + \delta$ where δ is the natural jitter introduced by the network delay. It is a widely adopted assumption that the distribution of jitter components can be modelled as a Laplacian distribution $Lap(0, \beta)$ [20, 8]. Subsequently we have

$$\cos(d, w) \sim Lap\left(\sqrt{a^2/(2\beta^2 + a^2)}, \frac{1}{\sqrt{2n}}\right).$$

On the other hand, when the flow is not marked or an incorrect v is chosen, we have $d = v' - v + \delta$, and $\cos(d, w) \approx Lap(0, \frac{1}{\sqrt{2n}})$. The two distribution are nicely separated when the proper a value is chosen. The decoder decides the flow is marked if it scores higher than

a threshold $\zeta = \frac{1}{2} \sqrt{a^2/(2\beta^2 + a^2)}$, and not marked otherwise.

4.2.1 Known Flow Attack

Under a known flow attack, the inter-packet delays before and after passing through the encoder are given, allowing the attacker to compute jitter. Therefore the task of detecting a RAINBOW watermark breaks down to distinguishing between vectors of the form $w + \delta$ and δ , where δ is normal jitter following a Laplacian distribution. In this case, there are several algorithms that could identify the watermark. We briefly introduce a detection algorithm that uses only one flow. Another detection algorithm that utilizes 2 or more flows and achieves perfect discrimination for most parameters is described in the appendix.

Single-flow Detection. We show how to detect the watermark with a single flow. Specifically, a histogram of jitter components within a time window serves as an excellent discriminator. In particular, we focus on the bin that counts the number of jitter components in the range $[-\beta/4, \beta/4]$.

By definition of the Laplacian distribution,

$$Pr(-\beta/4 < \delta_i < \beta/4) = 1 - e^{-1/4} \approx 0.221$$

so we expect over 1/5 components will fall in this bin. When the watermark w is added, each jitter component is translated to $\delta_i + a$ or $\delta_i - a$. Then under this new model, the probability that a watermarked jitter falls in the same bin is $Pr(-\beta/4 < x \pm a < \beta/4)$.

When we take $a \geq \beta/4$, we have

$$\begin{aligned} & Pr(-\beta/4 < x \pm a < \beta/4) \\ &= Pr(a - \beta/4 < |x| < a + \beta/4)/2 \\ &= (e^{-a/\beta+1/4} - e^{-a/\beta-1/4})/2 \end{aligned}$$

For different a values, this probability can be directly estimated; see Table 1. The larger a is, the fewer jitter values will fall in the bin. Using observed jitter values, the drop in frequency is even larger, since originally the majority of the components fall in $[-\beta/4, \beta/4]$.

The attack algorithm is very simple: Scan through the packet arrival times with a moving time window and compute the percentage of jitter components in the range $[-\beta/4, \beta/4]$ within the window. If it is lower than a threshold θ , tag the packets within the window as “marked”.

a =	0	$\beta/4$	$\beta/2$	β	2β
Frac.	0.221	0.197	0.153	0.093	0.034

Table 1: Expected fraction of watermark jitter in $[-\beta/4, \beta/4]$.

Algorithm 1 *RAINBOW-Embed*^a

Input: v, w, n
for $j = 1 \rightarrow n$ **do**
 if $v[j] + w[j] \geq 0$ **then**
 $v[j] = v[j] + w[j]$
 end if
end for
return v

^aIt is slightly different from the original algorithm described in [8]. However, this is the actual algorithm adapted in RAINBOW's source code and is in accordance with the experiment notes in [10].

Attack Evaluation. We simulated network flows by cutting randomly a subsequence of $n + 1$ packets from a randomly drawn flow and we simulate the encoder with $n = 200$ to 1000 and $a = 2$ to 20. For each set of parameters, we simulated 1000 marked flows and 1000 unmarked ones. Also we simulated natural network delay on each flow, with the RTT dataset by Houmansadr and Borisov in their work of SWIRL [7]. The resulting flow is later ran against by the attack algorithm. The Laplacian model of jitter, estimated from the RTT dataset, indicates $\beta = 10ms$.

With a moving window that takes $m = 200$ consecutive packets, the attacker obtains a histogram of jitter values within the window and focuses on the number of small values within $([-2.5ms, 2.5ms])$, with $\beta = 10ms$. As indicated in Table 1, in the idealized situation we should see the percentage of such small jitter values drop from 20% to 9.3% or lower when an added watermark amplitude $a > \beta$ is expected. We set the threshold θ to be 10%. To illustrate the detection performance, we evaluate the single-flow detection attack by three criteria: True Positive Rate, False Positive Rate and Average Recall.

From Table 2, as expected the detection algorithm successfully recovers almost the entire watermark when $a \geq 10ms$, regardless of the length of the watermark. And it also does a very good job even when $a = 5ms$ and the watermark is sufficiently long and the performance drops when $a = 2ms$. It is evident that the known flow attack is effective against RAINBOW, without any knowledge of the watermark key.

4.2.2 Output-only Detection Attack

Now we consider an attacker who can only see the flow after it passes the encoder: with only access to the potentially watermarked flow, we can still probe irregularities by sampling, with the assumption that the mark is not constantly present in the flow. In other words, we assume the flow contains marked segments and unmarked segments.

Algorithm 2 *RAINBOW-Detect*

Input: $v, DB = \{v_1, v_2, \dots, v_m\}, w, \zeta$
 $isDetected = FALSE$
for $i = 1 \rightarrow m$ **do**
 $d_i = v_i - v$
 $r = \cos(d_i, w)$
 if $r > \zeta$ **then**
 $isDetected = TRUE$
 end if
end for
return $isDetected$

a (ms)	length	TPR	FPR	Avg. Recall
2	200	0.004	0	0.001
	500	0.011	0	0.011
	1000	0.024	0	0.001
5	200	0.996	0	0.692
	500	1.000	0	0.846
	1000	1.000	0	0.921
10	200	1.000	0	0.984
	500	1.000	0	0.994
	1000	1.000	0	0.997
15	200	1.000	0	1.000
	500	1.000	0	1.000
	1000	1.000	0	1.000
20	200	1.000	0	1.000
	500	1.000	0	1.000
	1000	1.000	0	1.000

Table 2: Detection performance of single-flow chosen flow attack on RAINBOW

An important observation is made: The distribution of IPDs, $\{v_i\}$, is highly skewed. One example of the IPD distribution before and after RAINBOW watermarking is shown in Figure 5. In RAINBOW, a marked IPD v_i will be translated to $v_i + a$ or $v_i - a$. The skewness of these marked IPDs will be very different from the original.

Similar to the single flow detection algorithm, we utilize the change in histogram skewness as a predictor. Using a sliding window of w IPDs, histogram samples are generated. Viewed as vectors, these samples is naturally assumed to form a cluster in the linear space. Realizing the IPD distribution varies over time, we limit the histogram sampling to a flow segment of L packets. Since there is no single standardized model of IPD histogram we can refer to, we resort to using the centroid of the histogram samples to describe the cluster. The centroid is called the reference histogram, H_r . Further, each histogram sample is compared with the reference H_r . In case RAINBOW watermark is inserted, there exist samples that are much different from H_r . With a similar-

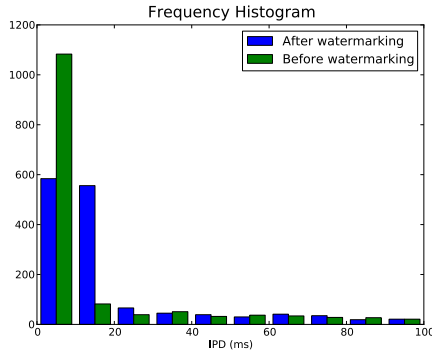


Figure 5: The histogram of first 2000 IPDs of one flow before and after watermarking ($n=2000$, $a = 10$ ms). The bin size is 5ms.

ity function $\Psi(\cdot, \cdot)$ and a threshold τ_S , we judge a flow to be marked if there exists a histogram H_i such that $\Psi(H_i, H_r) \leq \tau_S$.

The watermarking effect on histograms can be viewed as a linear transformation on them. Suppose two histograms H and H' represent distributions of IPDs before and after the watermarking, respectively. Then there exists a matrix M such that $H' = H \cdot M$, leading us to use the cosine function as the similarity measure.

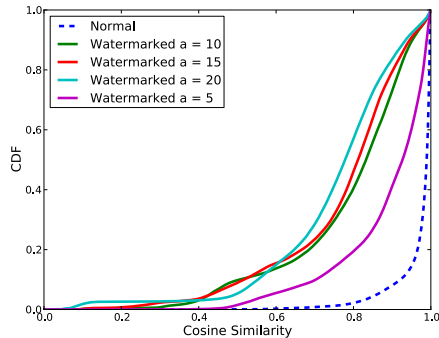


Figure 6: CDF with cosine similarity

A preliminary experiment confirms the effectiveness of cosine similarity on histograms. After selecting 100 random subsequences of $L = 4000$ packets from randomly drawn flows, we simulate RAINBOW watermarks (with $n = 1000$ and various a values) on each sequence. We take a sliding window of $w = 200$ IPDs. For the histogram sampling, we set the bin width to be 5ms, aiming to capture the watermark amplitudes $a > 5$ ms. We also “clip” IPD histograms at $v_i \leq 200$ ms as the majority of IPDs are in that range. The cumulative distribution functions of similarity scores over watermarked and non-marked regions respectively, are shown in Fig 6. The distance between the solid (watermarked) and dashed (un-

watermarked) distributions suggests that we should be able to distinguish between the cases.

Evaluation of Output-only Detection Attack. We simulated 1000 flows by selecting a random subsequence of 4000 packets from a randomly drawn flow. We then simulate RAINBOW watermarks on the flow sample, with multiple parameter combinations. For each flow, normal network delay is simulated by imposing one sample jitter sequence from the RTT dataset.

We experimented with τ_S values between 0 and 1, and calculate ROC curves for each parameter combination. We selected two sets of them to show in Figure 7. To see how the outlier detection is affected by smaller segment, we also repeated the evaluation with 2000-packet flows and found similar results. The area under curve (AUC) values for various parameters for both experiments are summarized in Table 3. The performance of the attack agrees with the intuition: the chances of detecting watermarks improves with increasing amplitude and length. When $a = 20$ ms, the AUC is generally close to or greater than 0.90. When $a \leq 5$ ms, the performance of the attack drops substantially because the histogram with 5ms bin fails to reflect the relatively invisible distribution change.

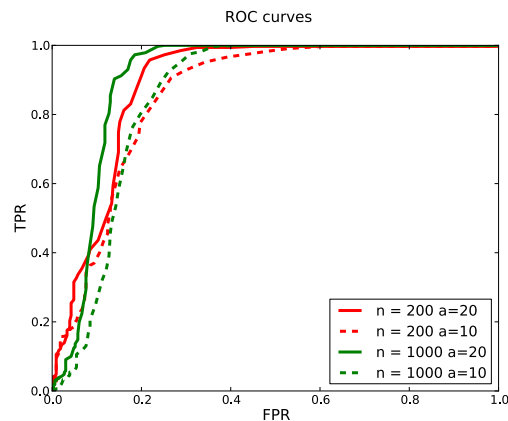


Figure 7: ROC curves of outlier detection

4.3 SWIRL

The design of SWIRL [7] can be briefly described as follows. Time is divided into *intervals* of two basic types: n “base intervals” in which seeds are generated, and n “mark intervals” in which packets are actually manipulated. Both being T seconds long, each base interval corresponds to one mark interval. Each mark interval is equally divided into r *sub-intervals*, and each sub-interval is equally divided into m *slots*. For each pair of base and mark intervals, r secret permutations, $\pi_1, \pi_2, \dots, \pi_r$, are generated, such that each π_i is a permutation of numbers from 1 to m . Each base

a	length	AUC ($L = 2000$)	AUC ($L = 4000$)
5	200	0.417	0.432
	500	0.492	0.455
	1000	0.732	0.489
10	200	0.641	0.858
	500	0.711	0.878
	1000	0.728	0.895
15	200	0.894	0.889
	500	0.924	0.917
	1000	0.927	0.905
20	200	0.875	0.889
	500	0.913	0.917
	1000	0.920	0.910

Table 3: AUC scores of outlier detection

interval will be used to derive a seed $s \in Z_m$ and $\pi_i(s)$ will determine a *designated slot* for the i -th sub-interval of the corresponding mark interval. The encoder and decoder have agreed on the choices of n, T, m, r , and π_1, \dots, π_r .

Within each mark interval, a watermark bit is inserted by re-arranging the packets to the designated slots for each sub-interval. More specifically, packets arriving before each designated slot are delayed into the subsequent one. The choice of seed s , the “quantized centroids” of base interval packets, is determined as follows: First, the centroid of packets, C , is computed as the mean arrival time of packets from the beginning of the interval. And the quantized centroid is

$$s = \lfloor qmC/T \rfloor \pmod{m},$$

where q is the quantization multiplier, introduced to increase randomness. The procedure of delaying packets into slots is illustrated in Figure 8.

For each outgoing flow at the other end, the decoder computes the centroids of every base interval and inspects the corresponding mark interval. A sub-interval is considered marked if the designated slot has packets. For each interval, a watermark bit is found if the number of marked sub-intervals is more than a pre-determined threshold τ . Finally, a watermark is detected if the number of watermark bits exceeds a preset threshold η .

4.3.1 Chosen Flow Attack

To make a SWIRL watermark stand out, the attacker can establish connections to a compromised host with uniformly paced traffic, sending K packets evenly per second. The interval centroid within such flows must lie within $[1/2, 1/2 + 1/K]$, regardless of the initial offset. As long as K is large, the quantized centroid will not

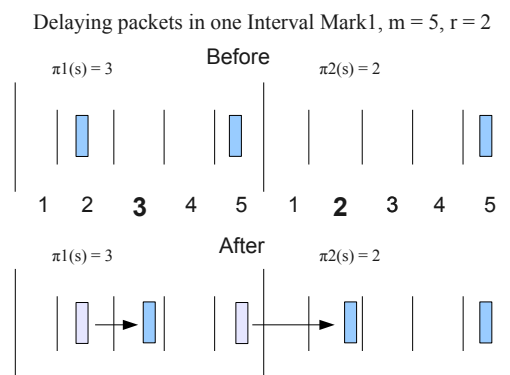


Figure 8: The marking procedure of SWIRL watermarking scheme

change. So each marked flow will have the same pattern of cleared and occupied intervals. To see the pattern across different flows, we introduce the following attack that works like MFA based on packets, rather than time intervals, due to the fact that the pattern is introduced with a random offset on each flow.

1. One attacker injects multiple flows toward the other; the IPDs of all packets in all flows are t ms.
2. The receiving attacker models the jitter as Laplacian and estimates the variance β .
3. The receiver flags packets that arrive too soon ($\leq t - \delta$) or too late ($\geq t + \delta$ ms) from the previous packet, with a threshold δ .
4. The receiver converts flagged packets into bit 1 and others into bit 0, transforming flows into bit strings. It examines the bit strings to see if there are common patterns of jitter.

Here is an example: We simulate an attacker that injects multiple flows with a rate of 20 packets per second ($t = 50$ ms) to the SWIRL encoder and observes the outgoing traffic. We implement the SWIRL encoder with the recommended parameters ($n = 32, T = 2s, r = 20$ and $m = 5$). For each flow, the encoder applies a unique key (including new assignments of base/mark intervals) and a unique random initial offset $o \in [0, T]$. Network jitter is simulated by the observed jitter dataset. The chosen flow attacker first computes the jitter and estimate the jitter model: $Lap(0, 10)$ and set $\delta = 20$ ms. Figure 9 shows three marked flows that are converted into bit sequences and put side by side for comparison.

The *a priori* likelihood of producing highly synchronized bit patterns by normal jitter is very small. The event of seeing a bit-1 (i.e. a jitter component $|j| > \delta$) happens with probability $p = e^{-\delta/\beta}$ by Laplacian distribution. Therefore each bit is the outcome of a Bernoulli

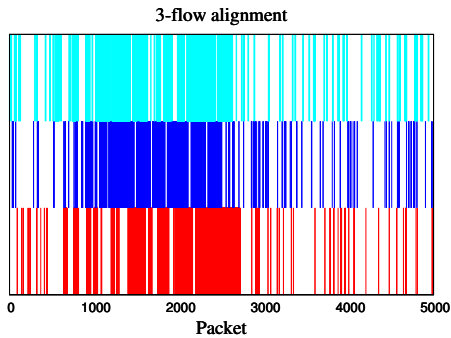


Figure 9: For each flow, an impulse is drawn when a packet is flagged. The synchronization of flags clearly shows the presence of watermark. On the contrast, the non-marked region on the right shows non-synchronization columns.

trial. With an alignment of k bit sequences, each column shows an outcome of k independent Bernoulli trials. The number of 1 bits in each column follows the binomial distribution $X \sim B(k, p)$. Subsequently, the probability of generating an all-1 column is $Pr(X = k) = p^k$. With observation of N columns, the expected total occurrence of all-1 columns is Np^k , with variance of $Np^k(1 - p^k)$.

Returning to the example, we have $\beta = 10$, $\delta = 20ms$, $k = 3$, so $p = 0.135$ and the probability of generating one all-1 column is 0.00248. In Figure 9, the left 2500 columns of the alignment have 350 all-1 columns, much higher than the expected value 6.25 with a Z score of 140. In contrast, the right half has only 8 all-1 columns.

Learning from the example, the attacker uses a moving observation window and simply sets a Z score threshold. She raises the alarm when she sees the number of all-1 columns surpassing the threshold.

Evaluation of Chosen Flow Attack. We simulate the SWIRL watermarking on synthetic flows with two sets of SWIRL parameters: ($n = 32$, $T = 2s$, $r = 10$, $m = 5$) and ($n = 32$, $T = 2s$, $r = 20$, $m = 5$). Each synthetic flow lasts for 300 seconds and contains traffic with a uniform rate of 20 PPS. The resulting flow is further distorted by a random jitter vector from jitter dataset. In each experiment, 3 marked flows are converted to bit sequence as described previously and aligned. The attacker runs through the alignment with a moving window of size 1000. β is estimated as 10. We repeat the detection attack for three times on each alignment with $\delta = 10ms$, $20ms$, and $30ms$ respectively. The Z score threshold is set to 10. 3 non-marked flows with jitter are also fed to the attacker as control. We repeat the experiment 100 times.

There is no surprise to see that the attacker correctly identifies all watermarked flows and non-marked ones without any error, no matter which δ value is chosen. The accuracy of 100% is attributed partly to the fact that

the Laplacian model usually over-estimates the tail distribution of natural jitter. That results in a much lower probability of seeing an all-1 column in the non-marked flow alignment, eliminating false positives.

4.3.2 Output-only Detection Analysis

Two important observations are made from the chosen flow attack on SWIRL:

- Because of the interval choice algorithm, mark intervals are likely allocated toward the end of the entire watermark period. Intuitively, out of 64 intervals, 1st interval must be base interval and the 64th must be mark interval. Very often, the last several intervals are all mark intervals.
- Initial offsets $\in [0, T]$ are not significant enough to break the synchronization of mark intervals.

Inspired by the observations, we examine the distribution of cleared and occupied sub-intervals and their relative positioning. With at least 75% probability, each designated slot is at least one half sub-interval length away from the neighboring designated slot. Intuitively we call this ‘isolation’, i.e. a packet in a marked region is either close to some neighboring packets because they are co-located in the same slot or far away from other packets. We slice the packets into groups by time, such that time gap between groups are at least \mathcal{T} microseconds (\mathcal{T} is chosen as a threshold.) Because of ‘isolation’, packets in marked intervals tend to form groups of short time-span.

To implement this heuristic, we use a naive clustering analysis algorithm that group packets by their arrival times. Two packets are grouped to the same cluster if their IPD in between is $\leq \mathcal{T}$. To capture the intuition of ‘many short clusters’, we use the maximum time-span among all clusters. If all clusters’ time-spans are short, the maximum is guaranteed to be short. For one flow, we first divide it into one-second flow snapshots. Then we apply the analysis to each snapshot and assign the maximum cluster time-span to the snapshot as its heuristic feature. We denote this by F_i for the i -th snapshot. We hypothesize that F_i has a lower expected value if the i -th second is in a marked interval. To see this, we repeat the same procedure on k flows and compute \bar{F}_i as the average F_i across multiple flows. The procedure is illustrated in Figure 10.

Here we give an example of such \bar{F}_i vectors derived from 30 non-marked flows and 30 marked flows, as shown in Figure 11.

The output-only attacker collects marked flows passively and converts them to lists of heuristic values F_i with clustering parameter t . She computes the mean \bar{F}_i

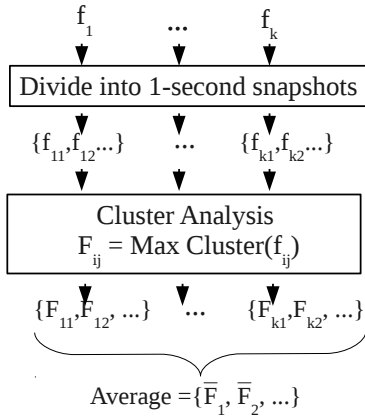


Figure 10: The procedure of computing \bar{F}_i

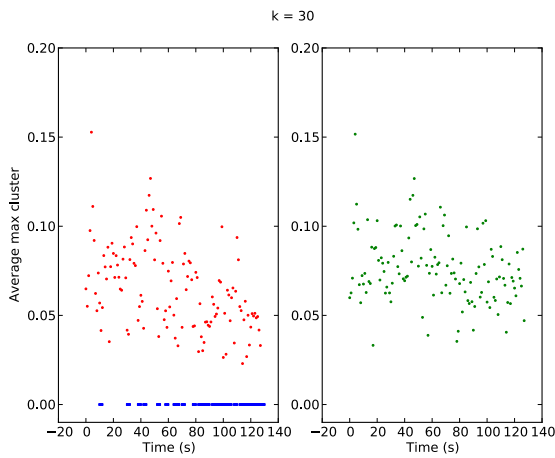


Figure 11: Two figures show $\{\bar{F}_i\}$ values derived from the same set of flows with and without watermark manipulation respectively. Marked flows has lower F_i values at marked intervals. (The thin dash line at $y=0.0$ level in the left figure indicates mark intervals.)

values across multiple flows. Marked regions are expected to score lower \bar{F}_i . Using a threshold τ_F , she determines that a flow is marked if there exists some $\bar{F}_i < \tau_F$. **Evaluation of Output-only Detection Attack.** To test the output-only attack, we first generate flows with packet rate $\lambda \leq 10$ packets per second, the target packet rate for SWIRL. We then repeat for 100 times the simulation of SWIRL watermarking on those flows with the same two set of system parameters in Chosen Flow Attack evaluation.

Assuming no knowledge of the system parameters, the attacker launches the attack with various values of \mathcal{T} . Unfortunately we don't have an analytical model for setting τ_F so we use area under ROC curve (AUC) to evaluate the performance of the attack. The evaluation is shown in Table 4. We have encouraging results when

AUC Score				
t (ms)	k=10	k=20	k=30	k=40
Sub-interval size = 100 ms (r=20)				
50	0.552	0.806	0.920	0.971
100	0.598	0.765	0.882	0.934
150	0.455	0.484	0.498	0.548
200	0.615	0.668	0.706	0.727
Sub-interval size = 200 ms (r=10)				
50	0.410	0.444	0.472	0.513
100	0.580	0.893	0.984	0.998
150	0.749	0.980	0.998	1.000
200	0.934	0.983	1.000	1.000

Table 4: AUC scores with combinations of t and k

the clustering threshold t is smaller than or equal to the subinterval length SI . Obviously, when \mathcal{S} is larger than SI , the heuristic fails as the 'isolation' is bounded by SI and is gone when we look for larger gaps. For the second parameter set, the attack is able to score AUC greater than 0.98 with $t \geq 100ms$ and $k \geq 30$. It is worth mentioning that when $\mathcal{S} = 50ms$, the attack cannot identify watermarked flows, due to the fact that the slot size is 40ms, very close to \mathcal{S} . In that case, given the low traffic rate, F_i with $\mathcal{S} = 50ms$ in the marked regions looks essentially the same as in non-marked regions, producing random numbers between 0 and 50ms. Finally, we are always able to find a threshold that allows the attack to separate marked flows and non-marked flows without error, when enough marked flows are accumulated ($k \geq 30$) and \mathcal{S} is between the slot size and the sub-interval size, e.g. 100ms.

We further argue that blindly setting $\mathcal{S} = 100ms$ will be suitable to detect most SWIRL watermarks. First of all, heuristically we see the sub-interval should not be smaller than 100ms. The reason is that a smaller sub-interval is closely related to larger false negatives. Smaller sub-intervals result in smaller slots, making the scheme vulnerable to natural network jitter. Even worse, many smaller sub-intervals will be unmarkable because there is no packet falling into them, resulting in detection difficulties. On the other hand, we also reason the slot size should not be much larger than 100ms, since larger slots lead to much worse invisibility [7].

5 Active Copy and Ambiguity Attacks

Copy attacks are common in the area of media watermarking. However, to our knowledge the copy attack on network flow watermarks has not been studied extensively. In this section, we focus on copy attacks from a isolated adversary's perspective, resulting in stronger attacks (based on weaker assumptions). For convenience

of discussion, we refer to the original watermarked flow as the *Source Flow* and refer to the to-be-copied-to flow as the *Target Flow*.

Generic Copy Attack The first passive copy attack is the replay attack, which simply replicates the IPD sequence of the source flow with the target flow. In the RAINBOW scheme, the decoder automatically links incoming and out-going flows when a watermark is detected, since the detection only succeeds when outgoing timing and recorded incoming timing are matched correctly. In case of the replay attack, the decoder will have to face two identical watermarked flows and will no longer be able to do exact traffic linking.

Moreover, the replay attack doesn't need to be turned on all the time due to the fact that the watermark decoder will tolerate some errors. For example, suppose the entire watermark is imposed on the first 1000 packets of both source and target flows. When the attack changes the first 500 IPDs of the target flow with the source flow as a template, it creates a "chimera" flow that looks 50% similar to the source and the target, making the decoder more confused.

Copy Attacks against SWIRL. Unlike RAINBOW, SWIRL is interval-based, searching for intervals manipulated by the watermark encoding algorithm. SWIRL also tolerates errors caused by natural delay and jitters. Therefore it is possible to copy SWIRL watermarks imperfectly. Two key insights enable us to design a best-effort keyless copy attack efficiently:

- The interval centroid will not change even when a small fraction of packets are missing.
- The SWIRL decoder only watches for one packet to appear in the right slot and not all designated slots need to be filled.

In order to copy the mark, the attacker simply delays packets with a timing template recorded moments ago. The self-synchronization of the watermarking detection algorithm will automatically shift the interval boundary in place so we don't need to worry about the actual position of marks in the flow. The specific copy attack works as follows (also shown in Figure 12):

1. Between time u and $u + \Delta$, the attacker sets up a buffer that records the arrival times of a source flow as $Q = \{t_1, t_2, \dots, t_n\}$ for packets coming within this period.
2. Between time $u + \Delta$ and $u + 2\Delta$, each packet of the target flow arriving at time v_i will be examined to see if it can be matched to $t_j \in Q$ so that v_i can be delayed to make $t_j - u = v_i - u - \Delta$. Unmatched packets will be handled normally without extra delay. Meanwhile, we use another buffer Q' to record the timing of A in this period.

λ	Parameters
$3 < \lambda < 20$	$n = 32, T = 2s, r = 20, m = 5$
$20 < \lambda < 80$	$n = 32, T = 2s, r = 40, m = 5$
$80 < \lambda$	$n = 32, T = 2s, r = 80, m = 5$

Table 5: Adaptive setting of SWIRL system parameters

3. Replace Q with Q' , u with $u + \Delta$, and repeat step2.

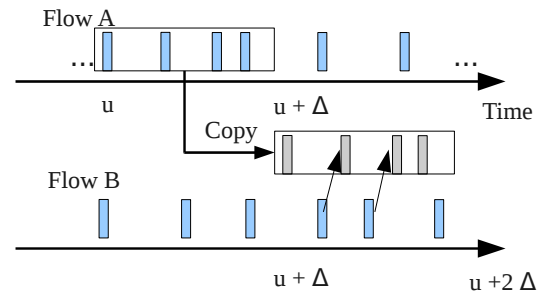


Figure 12: The *best-effort* copy attack on SWIRL scheme

This algorithm can run in real time and only requires two small buffers. Δ can be tuned to a small value, such as 500 milliseconds or 1 second. The "icing on the cake" is that the attacker can also copy the other way around simultaneously while copying from the source to the target.

Evaluation of Copy Attack on SWIRL. We implement the SWIRL encoder and decoder with multiple sets of parameters, adjusted to flows with different packet rates, shown in Table 5.

We simulated network flows by randomly choosing 128-second long intervals from randomly drawn flows. In total we obtained 300 flows for simulation and we simulated SWIRL watermarks on them, each with a unique watermark key. Our implementation of the copy attack chooses $\Delta = 500ms$. The attack is launched on every pair of flows 100 times. Eventually the average performance is calculated. The average number of bits detected on the target flow with the source flow's key is shown in Figure 13. Meanwhile, after the copy attack, the average bits detected by the target flow's key is also shown in Figure 14, organized by both flows' packet rates. The diagonal blocks show that it is easy to achieve copy effect between flows with similar system parameters. It also shows a clear trend that high-packet-rate flows ($\lambda > 40$) can easily copy any marks from a flow with a lower rate, achieving 25.4 bits detected on average. Also because of the high packet rate, they still remain watermarked by their original keys, with 19.5 bits. It is evident that the *best-effort* copy attack is agnostic to the actual settings

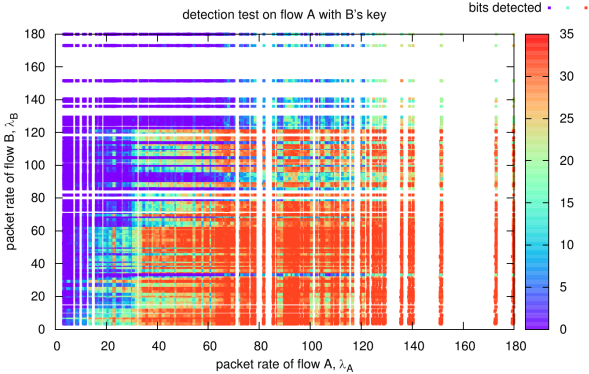


Figure 13: Bits detected after the target flow (A) copies

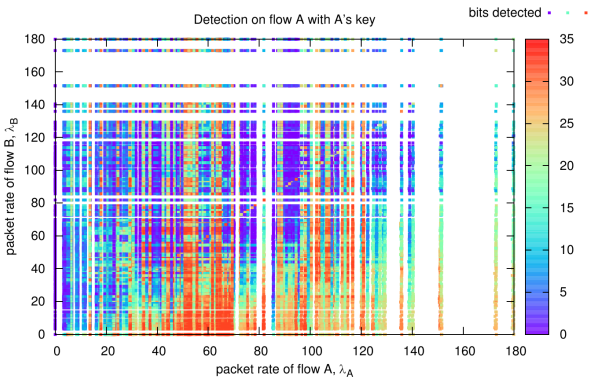


Figure 14: Bits detected after the best-effort copy attack, with the target flow's detection key.

of SWIRL such that two flows carrying completely different SWIRL watermarks can still exchange their marks without knowledge of SWIRL parameters.

6 Discussion

6.1 Defending Chosen Flow Attacks

Generally, defending against chosen flow attackers is hard given that timing manipulation is visible and measurable for such attackers. For jitter-based schemes, chosen flow attacker can directly compare the jitter in question with some theoretical model. One defense technique to borrow from steganography is that the imposed artificial jitter should approximate the real jitter as much as possible. As to interval-based schemes, Kiyavash et al. in their work of MFA already suggested choosing different mark intervals for different flows [9]. However, both PNR attacks and our chosen flow detection algorithms indicate that using base/mark approach doesn't resist against chosen flow adversary. How to choose mark intervals randomly to get pass such adversary becomes an interesting open question.

6.2 Defenses against Isolated Passive Attacks

6.2.1 RAINBOW

There are two ways to mitigate our outlier detection attack on RAINBOW: one is to use a smaller amplitude value a , such as 5ms. The other is to only watermark a subset of IPDs. For example, only one in every k IPDs will be touched for watermarking. Either way we face a trade-off between invisibility and robustness.

6.2.2 SWIRL

To defend against the passive attack we described, an easy fix for SWIRL is to use larger offsets and more random interval assignments. Large offsets breaks the synchronization of marked intervals so multi-flow alignment is less effective. A large offset might, however, present a scalability issue for the self-synchronized watermark decoder because now trying all possible offset values will be a time consuming task. Another defense is to employ a more sophisticated mark interval selection algorithm. For example, we could choose 32 out of 100 intervals to be mark intervals, and have the choice of the 32 mark intervals be determined by the base intervals of individual flows. Properly evaluating these defense strategies will require further research.

6.3 Utilizing Copy Attack for Anonymity

Large autonomous systems (ASes) such as ISPs control the network routing infrastructure in a distributed manner. Studies have pointed out that AS-level adversaries could reduce the link anonymity of Tor substantially by passive traffic analysis [5]. Equipped with high-capacity routers, such adversaries can also employ watermarking to improve the linking results. Blind watermarking schemes such as SWIRL are more suitable than non-blind schemes like RAINBOW because decoders in blind schemes can act along with pre-configured parameters and keys, saving a huge amount of resources.

To defend anonymity against AS-level adversaries that use SWIRL, Tor relays can launch the copy attack against SWIRL to mingle the timing information from multiple concurrent flows and confuse the decoders. The impact of copy attacks against passive and active traffic analysis on low-delay anonymity networks like Tor is an interesting open problem, appealing for future investigation.

7 Conclusion

We have proposed a security evaluation framework for network flow watermarking schemes, based on threat modeling. We started from a strong adversary, who is

capable of *chosen flow attacks*, and proved the effectiveness of chosen flow attacks against the recent state-of-the-art watermarking schemes, RAINBOW and SWIRL. Using insights from these attacks, we were able to devise detection attacks from the perspective of weaker isolated passive adversaries that still defeat the invisibility claims of these schemes. Additionally, we point out the feasibility of keyless copy attacks against flow watermarking and the importance of defending against such attacks in a traffic analysis scenario. In particular we develop an efficient and simple copy attack that works very well against SWIRL. We were able to transfer watermarks from one marked flow to another non-marked flow. Such copy attacks are especially of interests to Tor relays, which have concerns about link privacy with potential watermarking from ASes or compromised relays. In our future work, we would like to investigate the attack effectiveness in the real world and evaluate the subsequent performance impacts.

8 Acknowledgements

We thank Nikita Borisov and Amir Houmansadr for kindly sharing the source code and datasets for RAINBOW and SWIRL. We thank our shepherd Nikita Borisov as well as several anonymous reviewers for their constructive feedback. This work was supported by NSF grant 0917154.

References

- [1] A. Adelsbach, S. Katzenbeisser, and H. Veith. Watermarking schemes provably secure against copy and ambiguity attacks. In *Proceedings of the 3rd ACM workshop on Digital rights management, DRM '03*, pages 111–119, New York, NY, USA, 2003. ACM.
- [2] A. Blum, D. X. Song, and S. Venkataraman. Detection of interactive stepping stones: Algorithms and confidence bounds. In E. Jonsson, A. Valdes, and M. Almgren, editors, *International Symposium on Recent Advances in Intrusion Detection*, volume 3224 of *Lecture Notes in Computer Science*, pages 258–277. Springer, Sept. 2004.
- [3] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33:3–12, July 2003.
- [4] D. L. Donoho, A. G. Flesia, U. Shankar, V. Paxson, J. Coit, and S. Staniford. Multiscale stepping-stone detection: detecting pairs of jittered interactive streams by exploiting maximum tolerable delay. In *Proceedings of the 5th international conference on Recent advances in intrusion detection, RAID'02*, pages 17–35, Berlin, Heidelberg, 2002. Springer-Verlag.
- [5] M. Edman and P. F. Syverson. AS-awareness in tor path selection. In E. Al-Shaer, S. Jha, and A. D. Keromytis, editors, *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 380–389. ACM, 2009.
- [6] T. He and L. Tong. Detecting encrypted stepping-stone connections. *Trans. Sig. Proc.*, 55(5):1612–1623, May 2007.
- [7] A. Houmansadr and N. Borisov. SWIRL: A scalable watermark to detect correlated network flows. In *Network and Distributed System Security Symposium*. Internet Society, Feb 2011.
- [8] A. Houmansadr, N. Kiyavash, and N. Borisov. RAINBOW: A robust and invisible non-blind watermark for network flows. In *Network and Distributed System Security Symposium*. Internet Society, Feb 2009.
- [9] N. Kiyavash, A. Houmansadr, and N. Borisov. Multi-flow attacks against network flow watermarking schemes. In *Proceedings of the 17th conference on Security symposium*, pages 307–320, Berkeley, CA, USA, 2008. USENIX Association.
- [10] X. Luo, P. Zhou, J. Zhang, R. Perdisci, W. Lee, and R. K. C. Chang. Exposing invisible timing-based traffic watermarks with backlit. In *Proceedings of 2011 Annual Computer Security Applications Conference (ACSAC'11), Orlando, FL, USA, December 2011*.
- [11] P. Peng, P. Ning, and D. S. Reeves. On the secrecy of timing-based active watermarking traceback techniques. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 334–349, Washington, DC, USA, 2006. IEEE Computer Society.
- [12] Y. Pyun, Y. Park, X. Wang, D. S. Reeves, and P. Ning. Tracing traffic through intermediate hosts that repacketize flows. In G. Kesidis, E. Modiano, and R. Srikant, editors, *IEEE Conference on Computer Communications (INFOCOM)*, pages 634–642, May 2007.

- [13] C. Walsworth, E. Aben, kc claffy, and D. Andersen. The caida anonymized 2009 internet tracesjanuary., Mar. 2009.
- [14] X. Wang, S. Chen, and S. Jajodia. Tracking anonymous peer-to-peer voip calls on the internet. In *Proceedings of the 12th ACM conference on Computer and communications security, CCS '05*, pages 81–91, New York, NY, USA, 2005. ACM.
- [15] X. Wang, S. Chen, and S. Jajodia. Network flow watermarking attack on low-latency anonymous communication systems. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy, SP '07*, pages 116–130, Washington, DC, USA, 2007. IEEE Computer Society.
- [16] X. Wang and D. S. Reeves. Robust correlation of encrypted attack traffic through stepping stones by manipulation of interpacket delays. In *Proceedings of the 10th ACM conference on Computer and communications security, CCS '03*, pages 20–29, New York, NY, USA, 2003. ACM.
- [17] X. Wang, D. S. Reeves, and S. F. Wu. Inter-packet delay based correlation for tracing encrypted connections through stepping stones. In *Proceedings of the 7th European Symposium on Research in Computer Security, ESORICS '02*, pages 244–263, London, UK, UK, 2002. Springer-Verlag.
- [18] W. Yu, X. Fu, S. Graham, D. Xuan, and W. Zhao. Dsss-based flow marking technique for invisible traceback. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy, SP '07*, pages 18–32, Washington, DC, USA, 2007. IEEE Computer Society.
- [19] Y. Zhang and V. Paxson. Detecting stepping stones. In *Proceedings of the 9th conference on USENIX Security Symposium - Volume 9, SSYM'00*, pages 13–13, Berkeley, CA, USA, 2000. USENIX Association.
- [20] L. Zheng, L. Zhang, and D. Xu. Characteristics of network delay and delay jitter and its effect on voice over ip (voip). In *Communications, 2001. ICC 2001. IEEE International Conference on*, volume 1, pages 122–126 vol.1, jun 2001.

A Appendix

Multi-flow Detection of RAINBOW. First, we consider an attacker that initiates two flows through the watermark and receives them at the compromised host. Now he needs to determine if both flows carry the watermark

by looking at jitter vectors d^0 and d^1 . Similar to detection, we again use cosine similarity between d^0 and d^1 to test it. Therefore, we need to distinguish two hypotheses:

- H_0 : d^0 and d^1 are unwatermarked flows.
- H_1 : d^0 and d^1 are both watermarked flows carrying mark w .

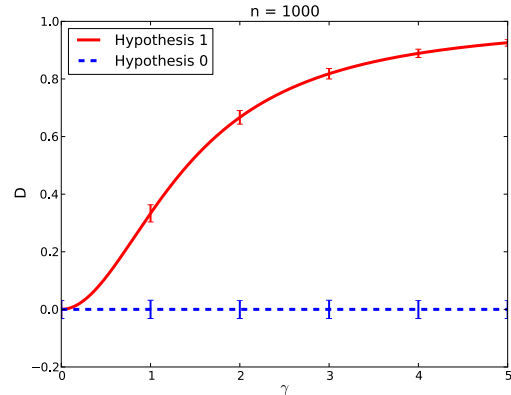


Figure 15: Cosine similarity between two jitter vectors

Define $D = \cos(d^0, d^1)$, Under H_0 , $d^i = \delta^i$ where δ^i are two independent jitter vectors. Under H_1 , $d^i = w + \delta^i$. The decision rule is to use a threshold τ_D , such that if $D \geq \tau_D$, we reject H_0 and the watermark is present, and otherwise absent. To see this working we have the following lemmas (the proofs are shown in the appendix):

Lemma 1. Suppose X_1, X_2, \dots, X_n are i.i.d random variables with Laplacian distribution $Lap(0, \beta)$. Then $E(\sum_{j=1}^n X_j^2) = n \cdot (2\beta^2)$

Proof.

$$\begin{aligned}
 E\left(\sum_{j=1}^n X_j^2\right) &= \sum_{j=1}^n E(X_j^2) \\
 &= n \cdot (\sigma^2(X_j) + E(X_j)^2) \\
 &= n \cdot (2\beta^2)
 \end{aligned} \tag{1}$$

□

Corollary 1. For $\delta = [X_1, X_2, \dots, X_n]$, $E(\|\delta\|) = \sqrt{n \cdot (2\beta^2)}$

Lemma 2. Suppose $X_1, X_2, \dots, X_n, Y_1, Y_2, \dots, Y_n$ are i.i.d random variables with Laplacian distribution $Lap(0, \beta)$. For $Z = \sum_{i=1}^n X_i Y_i$, $\sigma(Z) = \sqrt{n \cdot (2\beta^2)}$

Proof. Due to i.i.d random variables X_i, Y_i :

$$\begin{aligned}
 \sigma^2(Z) &= n\sigma^2(X_0 Y_0) \\
 &= n(2\beta^2)^2
 \end{aligned} \tag{2}$$

□

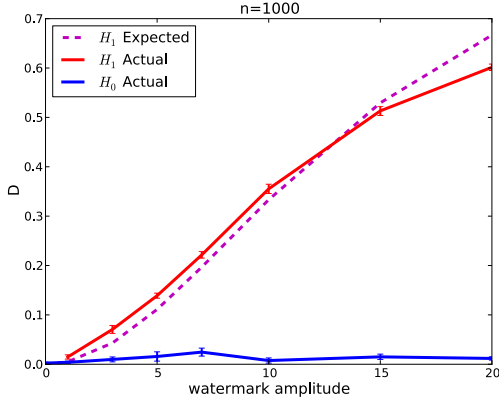


Figure 16: Statistics of RAINBOW multi-flow detection test

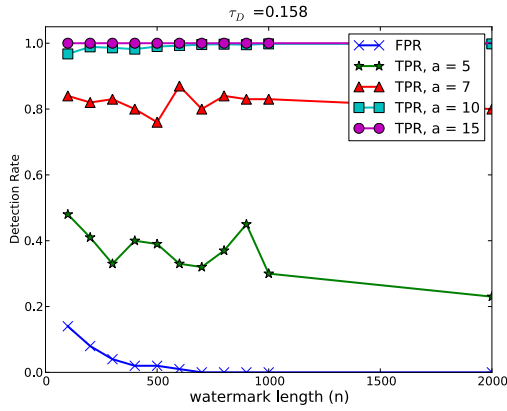


Figure 17: Performance of multi-flow detection attack

Lemma 3. *The inner product of jitter δ and watermark w , has distribution $\langle \delta, w \rangle \sim Lap(0, \sqrt{na}\beta)$. The proof is referred to [8].*

Under H_0 , the distribution of D satisfies the following characteristics:

$$\begin{aligned}
 E(D) &= \frac{E(\sum_{j=1}^n \delta_j^0 \delta_j^1)}{E(\|\delta^0\|)E(\|\delta^1\|)} \\
 &= \sum_{j=1}^n E(\delta_j^0)E(\delta_j^1)/E(\|\delta^0\|)^2 \\
 &= 0
 \end{aligned} \tag{3}$$

$$\begin{aligned}
 \sigma(D) &\approx \sigma(\sum_{j=1}^n \delta_j^0 \delta_j^1)/E(\|\delta^0\|)^2 \\
 &= \sqrt{n(2\beta^2)^2}/(\sqrt{n \cdot (2\beta^2)})^2 \\
 &= 1/\sqrt{n}
 \end{aligned} \tag{4}$$

Under H_1 , we have the following:

$$\begin{aligned}
 E(D) &= \frac{E(\langle w + \delta^0, w + \delta^1 \rangle)}{\|w + \delta^0\|^2 \cdot \|w + \delta^1\|^2} \\
 &\approx \frac{na^2 + E(\langle \delta^0, \delta^1 \rangle)}{\sqrt{(\|w\|^2 + \|\delta^0\|^2)(\|w\|^2 + \|\delta^1\|^2)}} \\
 &= na^2/(na^2 + n2\beta^2) \\
 &= \gamma^2/(2 + \gamma^2)
 \end{aligned} \tag{5}$$

where γ is the ratio of watermark amplitude to the Laplacian parameter β of jitter.

$$\begin{aligned}
 \sigma(D) &\approx \frac{\sqrt{2\sigma^2(\langle w, \delta \rangle) + \sigma^2(\langle \delta^0, \delta^1 \rangle)}}{(na^2 + n2\beta^2)} \\
 &= \frac{\sqrt{4a^2\beta^2 + 4\beta^4}}{\sqrt{n}(a^2 + 2\beta^2)} \\
 &= \frac{2\sqrt{\gamma^2 + 1}}{\sqrt{n}(\gamma^2 + 2)}
 \end{aligned} \tag{6}$$

$$= \frac{2\sqrt{\gamma^2 + 1}}{\sqrt{n}(\gamma^2 + 2)} \tag{7}$$

Figure 15 shows how the two hypotheses lead to different D distributions under varying values of γ with $n = 1000$. Evidently H_1 is significantly different from H_0 when n and γ are sufficiently large. In that case, detection attack can be accomplished by a simple statistic test that reject H_0 with good confidence.

We set a threshold τ_D for detection test. If two jitter vectors score higher than τ_D , the attacker rejects H_0 and deems them as “watermarked”. Otherwise it accepts H_0 . By Chebyshev’s inequality, we expect $FPR \leq \frac{1}{2k^2}$ with $\tau_D = k/\sqrt{n}$. Note now we can choose the threshold value independent of actual a and β values used by RAINBOW.

Attack Evaluation. We simulated network flows by cutting randomly a subsequence of $n + 1$ packets from a randomly drawn flow. For each set of n and a values, We simulated 1000 marked flows and 1000 unmarked ones, and ran the attack algorithm against them.

We first compute cosine similarity D between marked jitter vectors and between normal jitter vectors. Figure 16 shows that the actual detection performance fit nicely with the model. Therefore, we further set $\tau_D = 5/\sqrt{1000} \approx 0.158$, such that the false positive rate is $\leq 2\%$ when $n \geq 1000$. We then launch the detection attack with τ_D . The result, plotted in Figure 17, is satisfactory: Watermarks with $a \geq 10ms$ will be detected for sure while the false positive rate is nearly 0 when $n \geq 500$.

On Breaking SAML: Be Whoever You Want to Be

Juraj Somorovsky¹, Andreas Mayer², Jörg Schwenk¹, Marco Kampmann¹, and Meiko Jensen¹

¹Horst Görtz Institute for IT-Security, Ruhr-University Bochum, Germany

²Adolf Würth GmbH & Co. KG, Künzelsau-Gaisbach, Germany

{*Juraj.Somorovsky, Joerg.Schwenk, Marco.Kampmann, Meiko.Jensen*}@rub.de,
Andreas.Mayer@wuerth.com

Abstract

The Security Assertion Markup Language (*SAML*) is a widely adopted language for making security statements about subjects. It is a critical component for the development of federated identity deployments and Single Sign-On scenarios. In order to protect integrity and authenticity of the exchanged SAML assertions, the XML Signature standard is applied. However, the signature verification algorithm is much more complex than in traditional signature formats like PKCS#7. The integrity protection can thus be successfully circumvented by application of different XML Signature specific attacks, under a weak adversarial model.

In this paper we describe an in-depth analysis of 14 major SAML frameworks and show that 11 of them, including Salesforce, Shibboleth, and IBM XS40, have critical XML Signature wrapping (XSW) vulnerabilities. Based on our analysis, we developed an automated penetration testing tool for XSW in SAML frameworks. Its feasibility was proven by additional discovery of a new XSW variant. We propose the first framework to analyze such attacks, which is based on the information flow between two components of the Relying Party. Surprisingly, this analysis also yields efficient and practical countermeasures.

1 Introduction

The Security Assertion Markup Language (*SAML*) is an XML based language designed for making security statements about subjects. SAML assertions are used as security tokens in WS-Security, and in REST based Single Sign-On (SSO) scenarios. SAML is supported by major software vendors and open source projects, and is widely deployed. Due to its flexibility and broad support, new application scenarios are defined constantly.

SAML ASSERTIONS. Since SAML assertions contain security critical claims about a subject, the validity of these claims must be certified. According to the stan-

dard, this shall be achieved by using XML Signatures, which should either cover the complete SAML assertion, or an XML document containing it (e.g. a SAML Authentication response).

However, roughly 80% of the SAML frameworks that we evaluated could be broken by circumventing integrity protection with novel XML Signature wrapping (XSW) attacks. This surprising result is mainly due to two facts:

- **Complex Signing Algorithm:** Previous digital signature data formats like PKCS#7 and OpenPGP compute a single hash of the whole document, and signatures are simply appended to the document. The XML Signature standard is much more complex. Especially, the position of the signature and the signed content is variable. Therefore, many permutations of the same XML document exist.
- **Unspecified internal interface:** Most SAML frameworks treat the Relying Party (i.e. the Web Service or website consuming SAML assertions) as a single block, assuming a joint common state for all tasks. However, logically this block must be subdivided into the signature verification module (later called RP_{sig}) which performs a cryptographic operation, and the SAML processing module (later called RP_{claims}) which processes the claims contained in the SAML assertion. Both modules have different views on the assertion, and they typically only exchange a Boolean value about the validity of the signature.

CONTRIBUTION. In this paper, we present an in-depth analysis of 14 SAML frameworks and systems. During this analysis, we found critical XSW vulnerabilities in 11 of these frameworks. This result is alarming given the importance of SAML in practice, especially since SSO frameworks may become a single point of attack. It clearly indicates that the security implications behind SAML and XML Signature are not understood yet.

Second, these vulnerabilities are exploitable by an attacker with far fewer resources than the classical network based attacker from cryptography: Our adversary may succeed even if he does not control the network. He does not need realtime eavesdropping capabilities, but can work with SAML assertions whose lifetime has expired. A single signed SAML assertion is sufficient to completely compromise a SAML issuer/Identity Provider. Using SSL/TLS to encrypt SAML assertions, and thus to prevent adversaries from learning assertions by intercepting network traffic, does not help either: The adversary may e.g. register as a regular customer at the SAML issuer, and may use his own assertion to impersonate other customers.

Third, we give the first model for SAML frameworks that takes into account the interface between RP_{sig} and RP_{claims} . This model gives a clear definition of successful attacks on SAML. Besides its theoretical interest, it also enables us to prove several *positive* results. These results are new and help to explain why some of the frameworks were not vulnerable to our attacks, and to give advice on how to improve the security of the other 11 frameworks.

Last, we show that XSW vulnerabilities constitute an important and broad class of attack vectors. There is no easy defense against XSW attacks: Contrary to common belief, even signing the whole document does not necessarily protect against them. To set up working defenses, a better understanding of this versatile attack class is required. A specialized XSW pentesting tool developed during our research will be released as open source to aid this understanding. Its practicability was proven by discovering a new attack vector on Salesforce SAML interface despite the fact that specific countermeasures have been applied.

RESPONSIBLE DISCLOSURE. All vulnerabilities found during our analysis were reported to the responsible security teams. Accordingly, in many cases, we closely collaborated with them in order to patch the found issues.

OUTLINE. The rest of the paper is organized as follows. Section 2 gives a highlevel overview on SAML, and Section 3 adds details. The methodology of the investigation is explained in Section 4, and the detailed results are described in Section 5. In Section 6 we present the first fully automated XSW penetration test tool for SAML. Section 7 gives a formal analysis and derives two countermeasures. In Section 8 we discuss their practical feasibility. Section 9 presents an overview on related work. In the last section we conclude and propose future research directions.

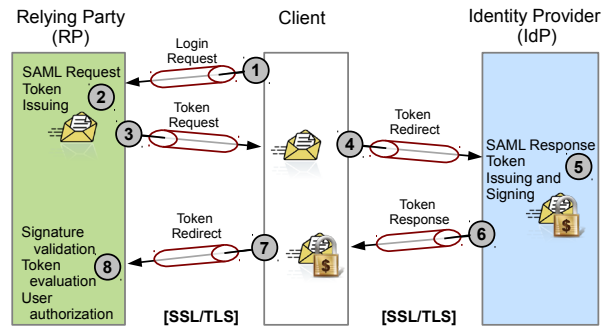


Figure 1: A typical Single Sign-On scenario: The user visits the RP, which generates a request token. He redirects this token to the IdP. The issued token is sent to the user and forwarded to the RP. Even though the channel is secured by SSL/TLS, the user still can see the token.

2 Motivation

In this section we introduce two typical SAML scenarios and some widely used SAML frameworks.

SAML-BASED SINGLE SIGN-ON. Typical Internet users have to manage many identities for different web applications. To overcome this problem, Single Sign-On was developed. In this approach the users authenticate only once to a trustworthy Identity Provider (IdP). After a successful login of a user, the IdP issues security tokens on demand. These tokens are used to authenticate to Relying Parties (RP).

A simplified Single Sign-On scenario is depicted in Figure 1. In this setting, a user logged-in by the IdP first visits the desired RP (1). The RP issues a token request (2). This token is sent to the user (3) who forwards it to the IdP (4). The IdP issues a token response for the user including several claims (e.g. his access rights or expiration time). In order to protect the authenticity and integrity of the claims, the token is signed (5). Subsequently, the token is sent to the user (6), who forwards it to the RP (7). The RP validates the signature and afterwards grants access to the protected service or resource, if the user is authorized (8). This access control decision is based on the claims in the validated token.

SECURING WEB SERVICES WITH SAML. Another typical application scenario is the use of SAML together with WS-Security [29] in SOAP [21] to provide authentication and authorization mechanisms to Web Services. SAML assertions are included as security tokens in the Security header.

SAML PROVIDERS AND FRAMEWORKS. The evaluation presented in this paper was made throughout the last 18 months and includes prominent and well-used SAML frameworks, which are summarized in Table 1. Our analysis included the IBM hardware appliance

Framework/Provider	Type	Language	Reference	Application
Apache Axis 2	WS	Java	http://axis.apache.org	WSO2 Web Services
Guanxi	Web SSO	Java	http://guanxi.sourceforge.net	Sakai Project (www.sakaiproject.org)
Higgins 1.x	Web SSO	Java	www.eclipse.org/higgins	Identity project
IBM XS40	WS	XSLT	www.ibm.com	Enterprise XML Security Gateway
JOSSO	Web SSO	Java	www.josso.org	Motorola, NEC, Redhat
WIF	Web SSO	.NET	http://msdn.microsoft.com	Microsoft Sharepoint 2010
OIOSAML	Web SSO	Java, .NET	http://www.oiosaml.info	Danish eGovernment (e.g. www.virk.dk)
OpenAM	Web SSO	Java	http://forgerock.com/openam.html	Enterprise-Class Open Source SSO
OneLogin	Web SSO	Java, PHP, Ruby, Python	www.onelogin.com	Joomla, Wordpress, SugarCRM, Drupal
OpenAthens	Web SSO	Java, C++	www.openathens.net	UK Federation (www.eduserv.org.uk)
OpenSAML	Web SSO	Java, C++	http://opensaml.org	Shibboleth, SuisseID
Salesforce	Web SSO	—	www.salesforce.com	Cloud Computing and CRM
SimpleSAMLphp	Web SSO	PHP	http://simplesamlphp.org	Danish e-ID Federation (www.wayf.dk)
WSO2	Web SSO	Java	www.wso2.com	eBay, Deutsche Bank, HP

Table 1: Analyzed SAML frameworks and providers: The columns give information about type (Web Service or Browser-based SSO), programming language (if known), web site, and application in concrete products or frameworks.

XS40, which is applied as an XML Security Gateway. Other examples of closed source frameworks are the Windows Identity Foundation (WIF) used in Microsoft Sharepoint and the Salesforce cloud platform. Important open source frameworks include OpenSAML, OpenAM, OIOSAML, OneLogin, and Apache Axis 2. OpenSAML is for example used in Shibboleth and the SDK of the electronic identity card from Switzerland (SuisseID). OpenAM, formerly known as SUN OpenSSO, is an identity and access management middleware, used in major enterprises. The OIOSAML framework is e.g. used in Danish public sector federations (e.g. eGovernment business and citizen portals). The OneLogin Toolkits integrate SAML into various popular open source web applications like Wordpress, Joomla, Drupal, and SugarCRM. Moreover, these Toolkits are used by many OneLogin customers (e.g. Zendesk, Riskconnect, Zoho, KnowledgeTree, and Yammer) to enable SAML-based SSO. Apache Axis2 is the standard framework for generating and deploying Web Service applications.

3 Technical Foundations

In this section we briefly introduce the SAML standard and XML Signature wrapping attacks. Additionally, for readers unfamiliar with the relevant W3C standards, we present XML Signature [14] and XML Schema [36].

3.1 XML Signature

The XML Signature standard [14] defines the syntax and processing rules for creating, representing, and verifying XML-based digital signatures. It is possible to sign a whole XML tree or only specific elements. One XML Signature can cover several local or global resources. A signature placed within the signed content is called an enveloped signature. If the signature surrounds the signed parts, it is an enveloping signature. A detached signature is neither inside nor a parent of the signed data.

```
<Signature ID?>
  <SignedInfo>
    <CanonicalizationMethod/>
    <SignatureMethod/>
    (<Reference URI? >
      (<Transforms>)?
      <DigestMethod>
      <DigestValue>
    </Reference>)+
  </SignedInfo>
  <SignatureValue>
  (<KeyInfo>)?
  (<Object ID??>)*
</Signature>
```

Figure 2: XML Signature data structure (“?”: zero or one occurrence; “+”: one or more occurrences; “*”: zero or more occurrences).

An XML Signature is represented by the Signature element. Figure 2 provides its basic structure. XML Signatures are two-pass signatures: the hash value of the resource (DigestValue) along with the used hash algorithm (DigestMethod) and the URI reference to the resource are stored in a Reference element. Additionally, the Transforms element specifies the processing steps which are applied prior to digesting of the resource. Each signed resource is represented by a Reference element in the SignedInfo element. Therefore, SignedInfo is a collection of hash values and URIs. The SignedInfo itself is protected by the signature. The CanonicalizationMethod and the SignatureMethod element specify the algorithms used for canonicalization and signature creation, and are also embedded in SignedInfo. The Base64-encoded value of the computed signature is deposited in the SignatureValue element. In addition, the KeyInfo element facilitates the transport of signature relevant key management information. The Object is an optional element that may contain any data.

```

<saml:Assertion Version ID IssueInstant>
  <saml:Issuer>
  <ds:Signature?>
  <saml:Subject?>
  <saml:Conditions?>
  <saml:Advice?>
  <saml:AuthnStatement*>
  <saml:AuthzDecisionStatement*>
  <saml:AttributeStatement*>
</saml:Assertion>

```

Figure 3: SAML assertion structure.

3.2 XML Schema

The W3C recommendation XML Schema [36] is a language to describe the layout, semantics, and content of an XML document. A document is deemed to be *valid*, when it conforms to a specific schema. A schema consists of a content model, a vocabulary, and the used data types. The content model describes the document structure and the relationship of the items. The standard provides 19 primitive data types to define the allowed content of the elements and attributes.

Regarding to our evaluation of SAML based XML Signature Wrapping attacks there is one important element definition in XML Schema. The any element allows the usage of any well-formed XML document in a declared content type. When an XML processor validates an element defined by an any element, the `processContents` attribute specifies the level of flexibility. The value `lax` instructs the schema validator to check against the given namespace. If no schema information is available, the content is considered valid. In the case of `processContents="skip"` the XML processor does not validate the element at all.

3.3 SAML

SAML is an XML standard for exchanging authentication and authorization statements about *Subjects* [11]. Several profiles are defined in [10]. The most important profile is the Browser SSO profile, which defines how to use SAML with a web browser.

A SAML assertion has the structure described in Figure 3. The issuing time of the assertion is specified in `saml:IssueInstant`. All attributes are required.

The `saml:Issuer` element specifies the SAML authority (the IdP) that is making the claim(s) in the assertion. The assertion's `saml:Subject` defines the principal about whom all statements within the assertion are made. The `saml:*Statement` elements are used to specify user-defined statements relevant for the context of the SAML assertion.

To protect the integrity of the security claims made by the Issuer, the whole `saml:Assertion` element must be protected with a digital signature following the XML

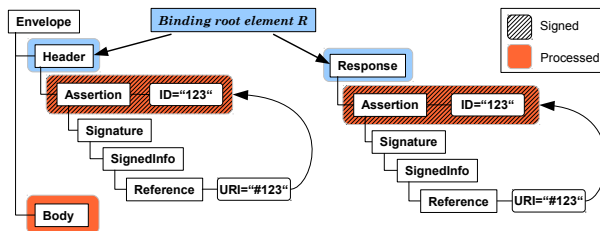


Figure 4: SAML message examples (SOAP and REST): The SAML assertion is put into a root element *R* and signed using an enveloped signature. When signing the SOAP body, an additional detached signature is used.

Signature standard. Therefore, the SAML specification [11] requires that either the `saml:Assertion` element or an ancestor element must be referenced by the `Signature` element, with an *enveloped* XML Signature ([11], Section 5.4.1). Furthermore, Id-based referencing must be used ([11], Section 5.4.2), which opens the way for XSW attacks.

In REST based frameworks, the SAML assertion is typically put into an enveloping `Response` element. Frameworks applying SOAP insert the SAML assertions into the SOAP header (or the `Security` element inside of the SOAP header). For clarification purposes, consider that the SAML assertions are signed using *enveloped* XML Signatures and are put into some binding root element *R* (see Figure 4).

3.4 XML Signature Wrapping Attacks

XML documents containing XML Signatures are typically processed in two independent steps: signature validation and function invocation (business logic). If both modules have different views on the data, a new class of vulnerabilities named *XML Signature Wrapping attacks* (XSW) [27, 23] exists. In these attacks the adversary modifies the message structure by injecting forged elements which do not invalidate the XML Signature. The goal of this alteration is to change the message in such a way that the application logic and the signature verification module use different parts of the message. Consequently, the receiver verifies the XML Signature successfully but the application logic processes the bogus element. The attacker thus circumvents the integrity protection and the origin authentication of the XML Signature and can inject arbitrary content. Figure 5 shows a simple XSW attack on a SOAP message.

XSW attacks resemble other classes of injection attacks like XSS or SQLi: in all cases, the attacker tries to force different views on the data in security modules (e.g. Web Application Firewalls) and data processing modules (HTML parser, SQL engine).

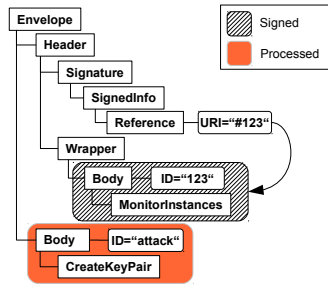


Figure 5: A simple XML Signature wrapping attack: The attacker moves the original signed content to a newly created Wrapper element. Afterwards, he creates an arbitrary content with a different Id, which is invoked by the business logic.

4 XSW Attacks on SAML

In this section we first characterize the assumed threat model. Second, we describe the basic attack principle that underlies our analysis of the 14 frameworks.¹

4.1 Threat Model

As a prerequisite the attacker requires an arbitrary signed SAML message. This could be a single assertion *A* or a whole document *D* with an embedded assertion, and its lifetime can be expired. After obtaining such a message, the attacker modifies it by injecting evil content, e.g. an evil assertion *EA*. In our model we assume two different types of adversaries, which are both weaker than the classical network based attacker:

1. *Adv_{acc}*. To obtain an assertion, this attacker registers as a user of an Identity Provider *IdP*. *Adv_{acc}* then receives, through normal interaction with *IdP*, a valid signed SAML assertion *A* (probably as a part of a larger document *D*) making claims about *Adv_{acc}*. The attacker now adds additional claims *EA* about any other subject *S*, and submits the modified document *D'* (*A'*) to *RP*.
2. *Adv_{intc}*. This adversary retrieves SAML assertions from the Internet, but he does not have the ability to read encrypted network traffic. This can be done either by accessing transmitted data directly from unprotected networks (sniffing), or in an "offline" manner by analyzing proxy or browser caches. Since SAML assertions should be worthless once their lifetime expired, they may even be posted

¹Please note that from now on we distinguish between the document *D* and the root element *R*. This is to make clear the distinction between the element referenced by the XML signature, and the document root: Even if the root element *R* of the original document *D* is signed, we may transform this into a new document *D'* with a new evil root *ER*, without invalidating the signature.

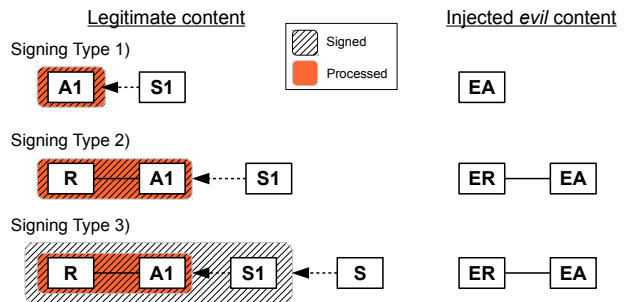


Figure 6: Types of signature applications on SAML assertions on the left. The new malicious content needed to execute the attacks depicted on the right, accordingly.

in technical discussion boards, where *Adv_{intc}* may access them.

4.2 Basic Attack Principle

As described in the previous section, XML Signatures can be applied to SAML assertions in different ways and placed in different elements. The only prerequisite is that the Assertion element or the protocol binding element (ancestor of Assertion) is signed using an enveloped signature with Id-based referencing. In this section we analyze the usage of SAML assertions in different frameworks and the possibilities of inserting malicious content. Generally, SAML assertions and their signatures are implemented as depicted in Figure 6:

1. The first possible usage of signatures in SAML assertions is to insert the XML Signature *S1* as a child of the SAML assertion *A1* and sign only the Assertion element *A1*. This type can be used independently of the underlying protocol (SOAP or REST).
2. The second type of signature application in SAML signs the whole protocol binding element *R*. The XML Signature can be placed into the SAML assertion *A1* or directly into the protocol binding root element *R*. This kind of signature application is used in different SAML HTTP bindings, where the whole Response element is signed.
3. It is also possible to use more than one XML Signature. The third example shows this kind of signature application: the inner signature *S1* protects the SAML assertion and the outer signature *S* additionally secures the whole protocol message. This kind of signature application is e.g. used by the Simple-SAMLphp framework.

In order to apply XSW attacks to SAML assertions, the basic attack idea stays the same: The attacker has

to create new malicious elements and force the assertion logic to process them, whereas the signature verification logic verifies the integrity and authenticity of the original content. In applications of the first signature type, the attacker only has to create a new *evil assertion EA*. In the second and third signing types, he also has to create the whole *evil root ER* element including the *evil assertion*.

4.3 Attack Permutations

The attacker has many different possibilities where to insert the malicious and the original content. To this end, he has to deal with these questions:

- At which level in the XML message tree should the malicious content and the original signed data be included?
- Which Assertion element is processed by the assertion logic?
- Which element is used for signature verification?

By answering these questions we can define different attack patterns, where the original and the malicious elements can be permuted (Figure 7). We thus get a complete list of attack vectors, which served as a guideline for our investigations.

For the following explanations we only consider signing type 1) defined in Figure 6. In this signing type only the Assertion element is referenced.

The attack permutations are depicted in Figure 7. In addition, we analyze their SAML standard conformance and the signature validity:

1. Malicious assertion, original assertion, and signature are left on the same message level: This kind of XML message can have six permutations. None of them is SAML standard compliant, since the XML Signature does not sign its parent element. The digest value over the signed elements in all the messages can be correctly validated. We can use this type of attack messages if the server does not check the SAML conformance.
2. All the three elements are inserted at different message levels, as child elements of each other, which again results in six permutations: Messages 2-a and 2-b show examples of SAML standard conforming and cryptographically valid messages. In both cases the signature element references its parent – the original assertion *A1*. Message 2-c illustrates a message which is not SAML standard conform as the signature signs its child element. Nevertheless, the message is cryptographically valid. Lastly, message 2-d shows an example of an invalid message since the signature would be verified over both assertions. Generally, if the signature is inserted as the

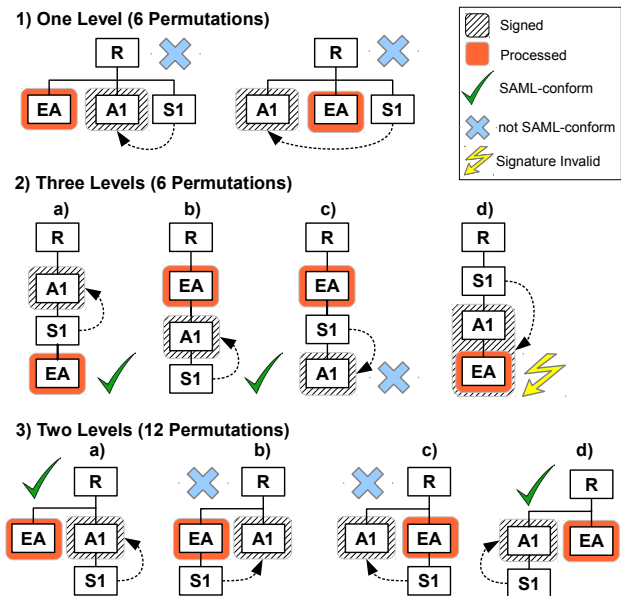


Figure 7: Possible variants for XSW attacks applied on messages with one signed SAML assertion divided according to the insertion depth of the evil assertion *EA*, the original assertion *A1* and the signature *S1*. The various permutations are labeled according to their validity and SAML-conformance.

child of the root element, the message would also be either invalid or not SAML standard compliant.

3. For the insertion of these three elements we use two message levels: Message 3-a shows an example of a valid and SAML compliant document. By constructing message 3-b, the signature element was moved to the new malicious assertion. Since it references the original element, it is still valid, but does not conform to the SAML standard.

The analysis shown above can similarly be applied to messages with different signing types (see Figure 6).

5 Practical Evaluation

We evaluated the above defined attacks on real-world systems and frameworks introduced in Section 2. In this section we present the results.

5.1 Signature Exclusion Attacks

We start the presentation of our results with the simplest attack type called *Signature exclusion attack*. This attack relies on poor implementation of a server's security logic, which checks the signature validity only if the signature is included. If the security logic does not find the Signature element, it simply skips the validation step.

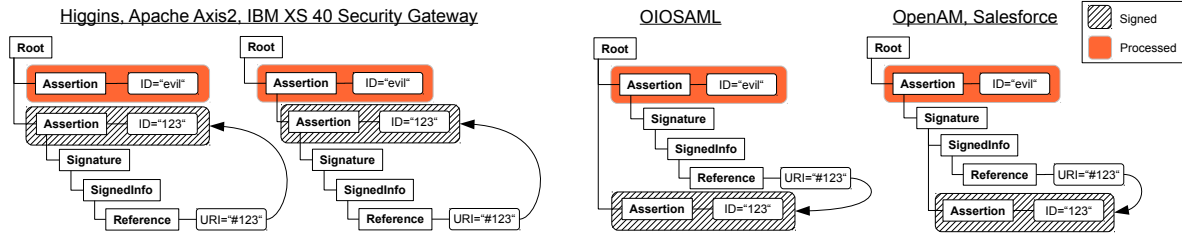


Figure 8: XML tree-based illustration of refined XSW attacks found in Type 1 signature applications.

The evaluation showed that three SAML-based frameworks were vulnerable to these attacks: Apache Axis2 Web Services Framework, JOSSO, and the Java-based implementation of SAML 2.0 in Eduserv (other versions of SAML and the C-implementation in Eduserv were not affected).

By applying this attack on JOSSO and Eduserv the attacker had to remove the `Signature` element from the message, since if it was found, the framework tried to validate it. On the other hand, the Apache Axis2 framework did not validate the `Signature` element over the SAML assertion at all, even if it was included in the message. Apache Axis2 validated only the signature over the SOAP body and the `Timestamp` element. The signature protecting the SAML assertion, which is included separately in the `Assertion` element, was completely ignored.

5.2 Refined Signature Wrapping

Ten out of 14 systems were prone to refined XSW attacks.

Classified on the three different signature application types given in Figure 6, five SAML-based systems failed in validating Type 1 messages, where only the assertion is protected by an XML Signature. Figure 8 depicts the XML tree-based illustration of the found XSW variants. Starting from left to right, Higgins, Apache Axis2, and the IBM XS 40 Security Gateway were outfoxed by the two depicted permutations. In the first variant it was sufficient to inject an evil assertion with a different `Id` attribute in front of the original assertion. As the SAML standard allows to have multiple assertions in one protocol element, the XML Schema validation still succeeded. The second attack type embedded the original assertion as a child element into the evil assertion *EA*. In both cases the XML Signature was still standard conform, as enveloped signatures were applied. This was broken in the case of OIOSAML by using detached signatures. In this variant the original `Signature` element was moved into the *EA*, which was inserted before the legitimate assertion. The last shown permutation was applicable to the cloud services of Salesforce

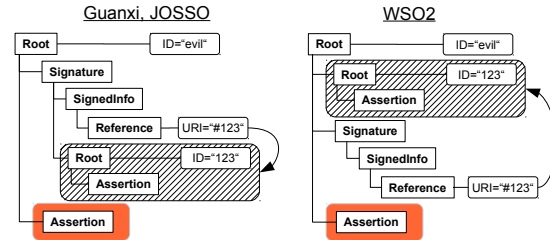


Figure 9: XML tree-based illustration of refined XSW attacks found in Type 2 signature applications.

and the OpenAM framework. At this, the genuine assertion was placed into the original `Signature` element. As both implementations apply XML Schema for validating the schema conformance of a SAML message, this was done by injecting them into the `Object` element, which allows arbitrary content. Again, this is not compliant to the SAML standard because this mutation transforms the enveloped to an enveloping signature. Finally, the OneLogin Toolkits were prone to all shown attack variants as they did not apply XML Schema, validated the XML Signature independent of its semantic occurrence and used a fixed reference to the processed SAML claims (`/samlp:Response/saml:Assertion[1]`).

We found three susceptible implementations, which applied Type 2 messages, where the whole message is protected by an XML Signature. We depict the attacks on these implementations in Figure 9. In the Guanxi and JOSSO implementations the legitimate root element was inserted into the `Object` element in the original `Signature`. The `Signature` node was moved into the *ER* element which also included the new evil assertion. In the case of WSO2, it was sufficient to place the original root element into the *ER* object. Naturally, someone would expect that enforcing full document signing would eliminate XSW completely. The both given examples demonstrate that this does not hold in practice. Again, this highlights the vigilance required when implementing complex standards such as SAML.

Finally, we did not find vulnerable frameworks that applied Type 3 messages, where both the root and the as-

sertion are protected by different signatures. Indeed, one legitimate reason is, that most SAML implementations do not use Type 3 messages. In our practical evaluation, only SimpleSAMLphp applied them by default. Nevertheless, this does not mean that XSW is not applicable to this message type in practice.

5.3 OpenSAML Vulnerability

The attack vectors described above did not work against the prevalently deployed OpenSAML library. The reason was that OpenSAML compared the Id used by the signature validation with the Id of the processed assertion. If these identifiers were different (based on a string comparison), the signature validation failed. Additionally, XML messages including more than one element with the same Id were also rejected. Both mechanisms are handled in OpenSAML by using the Apache Xerces library and its XML Schema validation method [34]. Nevertheless, it was possible to overcome these countermeasures with a more sophisticated XSW attack.

As mentioned before, in OpenSAML the Apache Xerces library performs a schema validation of every incoming XML message. Therefore, the Id of each element can be defined by using the appropriate XML Schema file. This allows the Xerces library to identify all included Ids and to reject messages with Id values which are not unique (e.g. duplicated). However, a bug in this library caused that XML elements defined with `<xsd:any processContents="lax">` were not checked using the defined XML Schema. Therefore, it was possible to insert elements with arbitrary – also duplicated – Ids inside an XML message. This created a good position for our wrapped content.

It is still the question which of the extensible elements could be used for the execution of our attacks. This depends on two processing properties:

1. Which element is used for assertion processing?
2. Which element is validated by the security module, if there are two elements with the same Id?

Interestingly, the two existing implementations of Apache Xerces (Java and C++) handled element dereferencing *differently*.

For C++, the attacker had to ensure that the original signed assertion was copied before the evil assertion. In the Java case, the legitimate assertion had to be placed within or after the evil assertion. In summary, if two elements with the same Id values occurred in an XML message, the XML security library detected only the first (for C++) or the last (for Java) element in the message. This property gave the attacker an opportunity to use

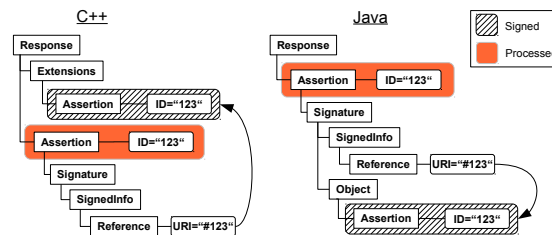


Figure 10: XSW attack on OpenSAML library.

```
<element name="Extensions" type="saml:ExtensionsType"/>
<complexType name="ExtensionsType">
  <sequence>
    <any namespace="##other" processContents="lax"
      maxOccurs="unbounded"/>
  </sequence>
</complexType>
```

Figure 11: XML Schema definition of the Extensions element.

e.g. the Extensions element for the C++ library, whose XML Schema is defined in Figure 11. However, the Extensions element is not the only possible position for our wrapped content. The schemas of SAML and XML Signature allow more locations (e.g. the Object element of the Signature, or the SubjectConfirmationData and Advice elements of the Assertion).

The previously described behavior of the XML schema validation forced OpenSAML to use the wrapped original assertion for signature validation. In contrast, the application logic processed the claims of the evil assertion. In Figure 10, we present the concrete attack messages of this novel XSW variant.

The successful attack on OpenSAML shows that countering the XSW attack can become more complicated than expected. Even when applying several countermeasures, the developer should still consider vulnerabilities in the underlying libraries. Namely, one vulnerability in the XML Schema validating library can lead to the execution of a successful XSW attack.

5.4 Various Implementation Flaws

While reviewing the OneLogin Toolkit, we discovered another interesting flaw: the implementation did not care about what data was actually signed. Therefore, any content signed by the IdP was sufficient to launch a XSW attack. In our case we used the metadata of the IdP² and created our own self-made response message to successfully attack OneLogin.

²The SAML Metadata [12] describes properties of SAML entities in XML to allow the easy establishment of federations. Typically, the metadata is signed by the issuer and publicly available.

Besides the fact that a SAML system has to check what data is signed, it is also essential to verify by whom the signature was created. In an early version of SimpleSAMLphp, which applied Type 3 messages, we observed that an attacker could forge the outer signature of the response message with any arbitrary key. In short, the SimpleSAMLphp RP did not verify if the included certificate in the KeyInfo element is trustworthy at all. The key evaluation for the signed assertion was correctly handled.

5.5 Secure Frameworks

In our evaluation of real-world SAML implementations we observed that Microsoft Sharepoint 2010 and SimpleSAMLphp were resistant to all applied test cases. Based on these findings the following questions arise: How do these systems implement signature validation? In which way do signature validation and assertion processing work together? Due to the fact that the source code of Sharepoint 2010 is not publicly available, we were only able to analyze SimpleSAMLphp.

According to this investigation the main signature validation and claims processing algorithm of SimpleSAMLphp performs the following five steps to counteract XSW attacks:

1. **XML Schema validation:** First, the whole response message is validated against the applied SAML schemas.
2. **Extract assertions:** All included assertions are extracted. Each assertion is saved as a DOM tree in a separate variable. The following steps are only applied on these segregated assertions.
3. **Verify what is signed:** SimpleSAMLphp checks, if each assertion is protected by an enveloped signature. In short, the XML node addressed by the URI attribute of the Reference element is compared to the root element of the same assertion. The XML Signature in the assertion is an enveloped signature if and only if both objects are identical.
4. **Validate signature:** The verification of every enveloped signature is exclusively done on the DOM tree of each corresponding assertion.
5. **Assertion processing:** The subsequent assertion processing is solely done with the extracted and successfully validated assertions.

When not considering the signature exclusion bug found in the OpenAthens implementation and its Java-based assertions' processing, this framework was also resistant to all the described attacks. The analysis of its implementation showed that it processes SAML assertions similarly to the above described SimpleSAMLphp framework.

Frameworks / Providers	Signing type	Signature exclusion	Refined XSW	Sophisticated XSW	Not vulnerable
Apache Axis 2	1)	X	X		
Guanxi	2)		X		
Higgins 1.x	1)		X		
IBM XS40	1)		X		
JOSSO	2)	X	X		
WIF	1)				X
OIOSAML	1)		X		
OpenAM	1)		X		
OneLogin	1)		X		
OpenAthens	1)	X			
OpenSAML	1)			X	
Salesforce	1)		X		
SimpleSAMLphp	3)				X
WSO2	2)		X		

Table 2: Results of our practical evaluation show that a majority of the analyzed frameworks were vulnerable to the refined wrapping techniques.

5.6 Summary

We evaluated 14 different SAML-based systems. We found 11 of them susceptible to XSW attacks, while the majority were prone to refined XSW. One prevalently used framework (OpenSAML) was receptive to a new, more subtle, variant of this attack vector. In addition, three out of the tested frameworks were vulnerable to Signature Exclusion attacks. We found two implementations, which were resistant against all test cases. The results obtained from our analysis are summarized in Table 2.

6 XSW Penetration Test Tool for SAML

Motivated on our crucial findings from the extensive frameworks' analysis and the vast amount of possible attack permutations, we implemented the first fully automated penetration test tool for XSW attacks in SAML-based frameworks. In this section we briefly describe the basic design decisions for our tool. Afterwards, we motivate its usage by revisiting the Salesforce SAML interface. This interface yielded a new possibility for an interesting XSW attack even after a deep investigation with different handcrafted messages.

Our tool will be integrated into the WS-Attacker framework³ and offered as open source to support the huge Web Services and SSO developers' community.

6.1 Penetration Test Tool

According to the theoretical and practical analysis of different SAML frameworks (see Section 4, 5), we gained the following general knowledge about XSW attacks:

³<http://ws-attacker.sourceforge.net>

- XML Schema validation:** Some of the SAML frameworks check message conformance to the underlying XML schema. Therefore, it is necessary to use XML schema extension points for placing the wrapped content. If the extension elements are not provided in the message, they have to be explicitly included.
- Order and position:** The order and position of signed and executed elements in the message tree can force the different processing modules to have inconsistent data views.
- Processing of the Ids:** Several SAML frameworks explicitly check, if the Id in the handled assertion is also used in the Reference of the XML Signature. Application of this countermeasure alone does not work, as there is still the option to use more elements with equal Ids.
- Placement of the Signature element:** The Signature element can be placed in the newly created evil assertion or stay in the original assertion (cf. the attacks on Higgins, Apache Axis2 and IBM XS40 in Figure 8). Both cases must be considered.
- Signature exclusion:** In three out of 14 frameworks implementation bugs caused that the signature validation step was omitted.
- Untrusted signatures:** It is essential to check that the signature was created with a trustworthy key. Otherwise, the attacker can forge a signature with any arbitrary key and embed the corresponding certificate in the KeyInfo element.

Based on this knowledge, we developed a library, which allows the systematic creation of a vast amount of different SAML attack vectors. Its processing can be summarized in the following steps. First, the library takes a signed XML document containing a SAML assertion and analyzes the usage of XML Signature. The element referenced by the signature is stored as a string. Subsequently, it creates a new malicious message including an evil assertion with modified content (e.g. the NameID and/or Timestamp element). Then, it searches dynamically for extension points in the XML Schema documents (e.g. XML Schemas for SAML, HTTP binding, XML Signature, or SOAP). It places the extension elements into the malicious message (e.g. a new Object element is created and placed into the given Signature element). Afterwards, the library embeds the stored original referenced element into each of the possible malicious message elements. For each position, a combination of different attack vectors – considering changes in the Ids of the newly created elements and the positions of the Signature elements – are created. For completeness, test cases for signature exclusion and untrusted signatures are provided. With these attack vectors, develop-

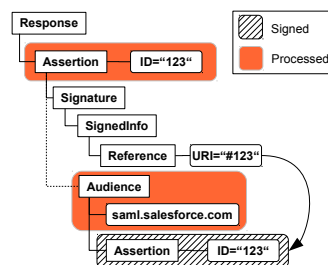


Figure 12: A successful XSW attack performed against the patched Salesforce SAML interface.

ers can systematically test the security of their (newly) developed SAML libraries.

6.2 Salesforce SAML Interface Revisited

After reporting the XSW vulnerability to Salesforce, the security response team developed a simple and promising countermeasure: the SAML interface solely accepted messages containing *one* Assertion element⁴. On request of the Salesforce security team, we investigated the fixed SAML interface with handcrafted messages containing wrapped contents in different elements. Our manual analysis did not reveal any new attack vectors. Every message containing more than one Assertion element was automatically rejected. Therefore, we first considered this interface to be secure.

A few months later, after finishing the development of our penetration test tool, we decided to retest the Salesforce SAML interface and prove the feasibility of our approach. Surprisingly, the automated penetration test tool revealed a new successful attack variant by inserting the wrapped content into the Audience element – a descendant of the Conditions element. This element typically contains a URI constraining the parties that can consume the issued assertion. The wrapped message is depicted in Figure 12. As can be seen in the figure, both Assertion elements needed to contain the same Id attribute.

This scientifically interesting attack vector stayed unanalyzed as the Salesforce security team did not expose any concrete information about their SAML interface. However, this finding shows how complex the development of secure signature wrapping countermeasures is. This motivates for further development of automatic penetration test tools for XSW.

Salesforce security team afterwards implemented a countermeasure, which could successfully mitigate all our attack types. Its details were not revealed.

⁴This countermeasure is not standard-conform as one message can generally contain several assertions. Therefore, we do not consider this remedy in our countermeasure analysis in Section 7.

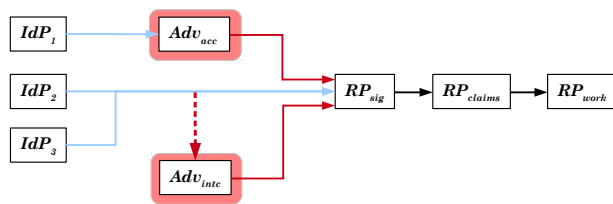


Figure 13: Overview of the components in our formal model.

7 Analysis and Countermeasures

In order to define what a successful attack on a SAML implementation is, we have to define the possibilities of the adversary, and the event that characterizes a successful attack. We do this in form of a game played between the adversary on one side, and *IdP* and *RP* on the other side. Additionally, we derive two different countermeasures. Their practical application is described in Section 8.

7.1 Data Model

A SAML assertion *A* can be sent to a Relying Party *RP* either as a stand-alone XML document, or as part of a larger document *D*. (*D* may be a complete SOAP message, or a SAML Authentication response.) To process the SAML assertion(s), the Relying Party (more specifically, *RP_{claims}*) searches for the `Assertion` element and parses it. We assume that *A* is signed, either stand-alone, or as part of *D*.

7.2 Identity Provider Model

We define an Identity Provider *IdP* to be an entity that issues signed SAML assertions, and that has control over a single private key for signing. Thus, companies like Salesforce may operate several *IdPs*, one for each domain of customers.

An Identity Provider *IdP* operates a customer database *db_{IdP}* and is able to perform a secure authentication protocol with any customer contained in this database. Furthermore, he has control over a private signing key, where the corresponding public key is trusted by a set of Relying Parties $\mathcal{R.P.} := \{RP_1, \dots, RP_n\}$, either directly, or through means of a Public Key Infrastructure. After receiving a request from one of the customers registered in *db_{IdP}*, and after successful authentication, he may issue a signed XML document *D*, where the signed part contains the requested SAML assertion *A*.

7.3 Relying Party Model

We assume that processing of documents containing SAML assertions is split into two parts: (1) XML Signature verification *RP_{sig}*, and (2) SAML security claims processing *RP_{claims}* (see Figure 13). This assumption is justified since both parts differ in their algorithmic base, and because this separation was found in all frameworks. If *RP_{claims}* accepts, then the application logic *RP_{work}* of the Relying Party will deliver the requested resource to the requestor.

The XML Signature verification module *RP_{sig}* is configured to trust several Identity Provider public keys $\{pk_1, \dots, pk_r\}$. Each public key defines a *trusted domain* within *RP*. After receiving a signed XML document *D*, *RP_{sig}* searches for a `Signature` element. It applies the referencing method described in `Reference` to retrieve the signed parts of the document, applies the transforms described in `Transforms` to these parts, and compares the computed hash values with the values stored in `DigestValue`. If all these values match, signature verification is performed over the whole `SignedInfo` element, with one of the trusted keys from $\{pk_1, \dots, pk_r\}$. *RP_{sig}* then communicates the result of the signature verification (eventually alongside *D*) to *RP_{claims}*.

The SAML security claims processing module *RP_{claims}* may operate a customer database *db_{RP}*, and may validate SAML assertions against this database. In this case if the claimed identity is contained in *db_{RP}*, the associated rights are granted to the requestor. As an alternative, *RP_{claims}* may rely on authorization data contained in *db_{IdP}*. In this case, the associated rights will be contained in the SAML assertion, and *RP_{claims}* will grant these.

Please note that the definition of the winning event given below does not depend on the output of the signature verification part *RP_{sig}*, but on the SAML assertion processing *RP_{claims}*. This is necessary since in all cases described in this paper, signature verification was done correctly (as is *always* the case with XML Signature wrapping). Therefore, to be able to formulate meaningful statements about the security of a SAML framework, we must make some assumptions on the behavior of *RP_{claims}*.

There are many possible strategies for *RP_{claims}* to process SAML assertions: E.g. use the claims from the first assertion which is opened during parsing, from the first that is closed during parsing (analogously for the last assertion opened or closed), or issue an error message if more than one `Assertion` element is read.

7.4 Adversarial Model

Please recall the two different types of adversaries we have mentioned in our threat model in Section 4. *Adv_{intc}* is the stronger of the two: He has the ability to

partially intercept network traffic, e.g. by sniffing HTTP traffic on an unprotected WLAN, by reading past messages from an unprotected log file, or by a chosen ciphertext attack on TLS 1.0 along the lines of [5]. Please note that already this adversary is strictly weaker than the classical network based attacker known from cryptography. Adv_{acc} , our weaker adversary, only has access to the IdP and RP , i.e. he may register as a customer with IdP and receive SAML assertions issued about himself, and he may send requests to RP .

We define preconditions and success conditions of an attacker in the form of a game G . If Adv mounts a successful attack under these conditions, we say that Adv wins the game. This facilitates some definitions.

During the game G , the adversary has access to a validly signed document D containing a SAML assertion A issued by IdP . He then generates his own (evil) assertion EA , and combines it arbitrarily with D into an XML document D' . This document is then sent to RP .

Definition 1. We say that the adversary (either Adv_{intc} or Adv_{acc}) wins game G if RP , after receiving document D' , with non-negligible probability $Pr(Win_{Adv})$ bases its authentication and authorization decisions on the security claims contained in EA .

Remark: For all researched frameworks, the winning probability was either negligible or equal to 1. Within the term "negligible" we include the possibility that Adv issues a forged cryptographic signature, which we assume to be impossible in practice. If an adversary wins the game against a specific Relying Party RP , he takes over the trust domain for a specific public key pk within RP . Adv_{acc} may do this for all pk where he is allowed to register as a customer with the corresponding IdP who controls (sk, pk) . Adv_{intc} can achieve this for all pk where he is able to find single signed SAML assertion A where the signature can (could in the past) be verified with pk .

7.5 Countermeasure 1: Only-process-what-is-hashed

We can derive the first countermeasure if we assume that RP_{sig} acts as a filter and only forwards the hashed parts of an XML document to RP_{claims} . The hashed parts of an XML document are those parts that are serialized as an input to a hash function, and where the hash value is stored in a `Reference` element. This excludes all parts of the document that are removed before hash calculation by applying a transformation, especially the enveloped signature transform.

Claim 1. If RP_{sig} only forwards the hashed parts of D to RP_{claims} , then $Pr(Win_{Adv})$ is negligible.

It is straightforward to see that EA is only forwarded to RP_{claims} if a valid signature for EA is available.

Please note that although this approach is simple and effective, it is rarely used in practice due to a number of subtle implementation problems. A variant of this approach is implemented by SimpleSAMLphp, where the RP imposes special requirements on the SAML authentication response, thus limiting interoperability. We discuss these problems in Section 8.

7.6 Countermeasure 2: Mark signed elements

In practice, RP_{sig} only returns a Boolean value, and the whole document D is forwarded to RP_{claims} . Since IdP has to serve many different Relying Parties, we assume knowledge about the strategy of RP_{claims} only for RP_{sig} . One possibility to mark signed elements is to hand over the complete document D from RP_{sig} to RP_{claims} , plus a description where the validly signed assertions can be found.

A second possibility that is more appropriate for SAML is that RP_{sig} chooses a random value r , marks the validly signed elements with an attribute containing r , and forwards r together with the marked document. RP_{claims} can then check if the assertion processed contains r .

Let us therefore consider the second approach in more detail. For sake of simplicity we assume that only one complete element (i.e. a complete subtree of the XML document tree) is signed.

Claim 2. Let D_{sig} be the signed subtree of D , and let $r \in \{0, 1\}^l$ be the random value chosen by RP_{sig} and attached to D_{sig} . Then $Pr(Win_{Adv})$ is bounded by $\max\{break_{sig}, 2^{-l}\}$.

RP_{claims} (regardless of its strategy to choose an assertion) will only process EA if r is attached to this element. An adversary can achieve this by either generating a valid signature for EA (then r will be attached by RP_{sig}), or by guessing r and attaching it to EA .

8 Practical Countermeasures

In Section 5.5 we analyzed message processing of SimpleSAMLphp. This framework was resistant against all XSW attacks. One could therefore ask a legitimate question: Why do we need further countermeasures and why is it not appropriate to apply the security algorithm of SimpleSAMLphp in every system?

We want to make clear that SimpleSAMLphp offers both critical functionalities in one framework: signature validation (RP_{sig}) and SAML assertion evaluation

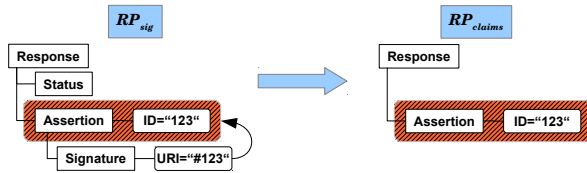


Figure 14: The see-what-is-signed approach applied in HTTP POST binding: After successful signature validation the security module RP_{sig} excludes all the unsigned elements and forwards the message to the module processing security claims RP_{claims} and the business logic.

(RP_{claims}). These two methods are implemented using the same libraries and processing modules. After parsing a document, the elements are stored within a document tree and can be accessed directly. This allows the security developers to conveniently access the same elements used in signature validation and assertion evaluation steps. However, especially in SOA environments there exist scenarios, which force the developers to separate these two steps into different modules or even different systems, e.g.:

- **Using a signature validation library:** Before evaluating the incoming assertion elements, the developer uses a DOM-based signature library, which returns true or false according to the message validity. However, the developer does not exactly know which elements were validated. If the assertion evaluation uses a different parsing approach (e.g. streaming-based SAX or StAX approach) or another DOM-library, the message processing could become error-prone.
- **XML Security gateways:** XML Security gateways can validate XML Signatures and are configured to forward only validated XML documents. If the developer evaluates a validated document in his application, he again has no explicit information about the position of the signed element. Synchronization of signature and assertion processing components in this scenario becomes even more complicated, if the developer has no information about the implementation of the security gateway (e.g. IBM XS40).

These two examples show that a convenient access to the same XML elements is not always given. Subsequently, we present two practical feasible countermeasures, which can be applied in complex and distributed real-world implementations. Both countermeasures result from our formal analysis in Section 7.

8.1 See-what-is-signed

The core idea of this countermeasure is to forward only those elements to the business logic module (RP_{claims})

that were validated by the signature verification module (RP_{sig}). This is not trivial as extracting the unsigned elements from the message context could make the further message processing in some scenarios impossible. Therefore, we propose a solution that excludes only the unsigned elements which do not contain any signed descendants. We give an example of such a message processing in Figure 14. This way, the claims and message processing logic would get the whole message context: in case of SOAP it would see the whole `Envelope` element, by application of HTTP POST binding it would be able to process the entire `Response` element. The main advantage of this approach is that the message processing logic does not have to search for validated elements because all forwarded elements are validated.

We want to stress the fact that by application of this approach *all* unsigned character nodes have to be extracted. Otherwise, the attacker could create an evil assertion EA and insert the signed original assertion into each element of EA . If RP_{sig} would not extract the character contents from EA , RP_{claims} could process its claims. However, by extracting the unsigned character nodes, the attacker has no possibility to insert his *evil* content, since it was excluded in RP_{sig} . Nevertheless, the subsequent XML modules can still access the whole XML tree.

This idea has already been discussed by Gajek et al. [17]. However, until now no XML Signature framework implements this countermeasure. It could be applied especially in the context of SAML HTTP POST bindings because the unsigned elements within the SAML response do not contain any data needed in RP_{claims} . We consider this countermeasure in these scenarios as appropriate because the SAML standard only allows the usage of Id-based referencing, exclusive canonicalization, and enveloped transformation. The authors explicitly state that this countermeasure would not work if XML Signature uses specific XSLT or XPath transformations.

8.2 Unique Identification (Tainting) of Signed Data

The second countermeasure represents another form of the *see-what-is-signed* approach. The basic idea is to uniquely identify the signed data in the RP_{sig} module and forward this information to the following modules. As described in our formal analysis, this could be done by generating a random value r , sending it to the next processing module (or as an attribute in the document root element), and attaching it to all the signed elements. We give an example of this countermeasure applied to a SOAP message in Figure 15.

The main drawback of this countermeasure is that the SAML XML Schema does not allow the inclusion of

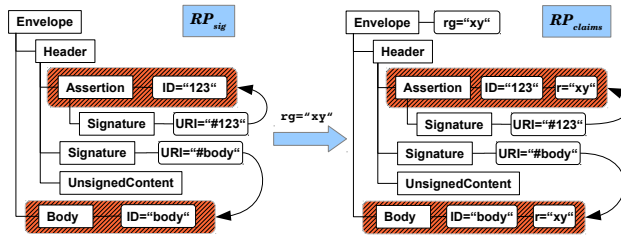


Figure 15: Unique identification of signed data applied on a SOAP message including two signed elements: The RP_{sig} module uniquely identifies the signed elements with a random value r and forwards this information along with the whole XML message.

new attributes: neither directly into the Assertion element nor the Response binding element. Therefore, the XML Schema validation of the assertion processing module would fail. For general application of this idea the SAML XML Schema needs to be extended.

Another possibility to implement this countermeasure is to use XML node types, which do not violate XML Schema, but are visible to the XML processors. For example, processing instructions, which are intended to carry instructions to the application belong to this group. They can be placed anywhere in the document without invalidating the XML Schema. Additionally, they can be conveniently found by processing XML trees with streaming and DOM-based parsers. Therefore, the presence of these XML nodes would help to find the validated data and thus allows to mitigate XSW attacks. We propose this technique for further discussion in the W3C XML Security Working Group and the OASIS consortium.

9 Related Work

XML Signature Wrapping (XSW). XSW attacks have first been described in [28] and [7]. Several countermeasures have been proposed over time.

McIntosh and Austel [28] have presented several XSW attacks and discussed (informal) receiver-sided security policies in order to prevent such exploits. They have however not given a definitive solution for this problem.

Bhargavan, Fournet and Gordon [7] have analyzed a formal approach in order to verify Web Services specifications. They have proposed a policy advisor [8], a tool that generates appropriate security policies for Web Services protocols. This approach is however not directly applicable to SAML.

Rahaman, Schaad and Rits [32, 30, 31] have refrained from policy-driven approaches and have introduced an inline solution. The authors have proposed to embed an Account element into the SOAP header. This element

contains partial information about the structure of the SOAP message and the neighborhood of the signed element(s). The information preserves the structure of the data to be signed. However, Gajek et al. have shown that this approach does not prevent XSW attacks [16]. Benameur, Kadir, and Fenet [6] have extended the inline approach, but suffer from the same vulnerabilities.

Jensen et al. [24] have analyzed the effectiveness of XML Schema validation in terms of fending XSW attacks in Web Services. Thereby, they have used manually hardened XML Schemas. The authors have concluded that XML Schema validation is capable of fending XSW attacks, at the expense of two important disadvantages: for each application a specific hardened XML Schema without extension points must be created carefully. Moreover, validating of a hardened XML Schema entails severe performance penalties.

XPath and XPath Filter 2 are specified as referencing mechanisms in the XML Signature standard. However, the WS-Security standard proposes not to use these mechanisms, and the SAML standard mandates to use Id-based referencing instead. This is due to the fact that both standards are very complex. Gajek et al. [15] have evaluated the effectiveness of these mechanisms to mitigate XSW attacks in the SOAP context, and have proposed a lightweight variant FastXPath, which has lead to the same performance in a PoC implementation as by adapting the Id-based referencing.

Jensen et al. [23] have however shown that this approach does not completely eliminate XSW attacks: by clever manipulations of XML namespace declarations within a signed document, which take into account the processing rules for canonicalization algorithms in XML Signature, XSW attacks could successfully be mounted even against XPath referenced resources.

The impacts of practical XSW attacks have also been analyzed in [20, 33]. In these works new types of XSW attack have been applied on SOAP Web Service interfaces of Amazon and Eucalyptus clouds. The attacks have exploited different XML processing in distinct modules.

In summary, previous work has mostly concentrated on SOAP, and the results do not directly apply to all SAML use cases.

SAML and Single Sign-On Since SAML offers very flexible mechanisms to make claims about identities, there is a large body of research on how SAML can be used to improve identity management (e.g. [22, 39]) and other identity-related processes like payment or SIP on the Internet [25, 35]. In all these applications, the security of all SAML standards is assumed.

In an overview paper on SAML, Maler and Reed [26] have proposed *mutually authenticated* TLS as the basic

security mechanism. Please note that even if mutually authenticated TLS would be employed, it would not prevent our attacks because we only need a single signed SAML assertion from an IdP, which we can get through different means. Moreover, there exist specific sidechannels, which could be exploited by an adversary. Let us e.g. mention chosen-plaintext attacks against SSL/TLS predicted by [37] and refined by [5], or the Million Question attack by Bleichenbacher [9]. Other complications arise with the everlasting problems with SSL PKIs.

In 2003, T. Groß has initiated the security analysis of SAML [18] from a Dolev-Yao point of view, which has been formalized in [4]. He has found, together with B. Pfizmann [19], deficiencies in the information flow between the SAML entities. Their work has influenced a revision of the standard.

In 2008, Armando et al. [3] have built a formal model of the SAML 2.0 Web Browser SSO protocol and have analyzed it with the model checker SATMC. By introducing a malicious RP they have found a practical attack on the SAML implementation of Google Apps. Another attack on the SAML-based SSO of Google Apps has been found in 2011 [2]. Again, a malicious RP has been used to force a user's web browser to access a resource without approval. Thereby, the malicious RP has injected malicious content in the initial unintended request to the attacked RP. After successful authentication on the IdP this content has been executed in the context of the user's authenticated session.

The fact that SAML protocols consist of multiple layers has been pointed out in [13]. In this paper, the Weakest Link Attack has enabled adversaries to succeed at all levels of authentication by breaking only at the weakest one.

Very recently, another work pointing out the importance of SSO protocols has been published by Wang et al. [38]. This work has analyzed the security quality of commercially deployed SSO solutions. It has shown eight serious logic flaws in high-profile IdPs and RPs (such as OpenID, Facebook, or JanRain), which have allowed an attacker to sign in as the victim user. The SAML-based SSO has not been analyzed.

10 Conclusion

In this paper we systematically analyzed the application of XSW attacks on SAML frameworks and systems. We showed that the large majority of systems exhibit critical security insufficiencies in their interfaces. Additionally, we revealed new classes of XSW attacks, which worked even if specific countermeasures were applied. We showed that the application of XML Security heavily depends on the underlying XML processing system (i.e. different XML libraries and parsing types). The pro-

cessing modules involved can have inconsistent views on the same secured XML document, which may result in successful XSW attacks. Generally, these heterogeneous views can exist in all data formats beyond XML.

We proposed a formal model by analyzing the information flow inside the Relying Party and presented two countermeasures. The effectiveness of these countermeasures depends on the *real* information flow and the data processing inside RP_{claims} . Our research is a first step towards understanding the implications of the information flow between cryptographic and non-cryptographic components in complex software environments. Research in this direction could enhance the results, and provide easy-to-apply solutions for practical frameworks.

As another future research direction, we propose development of an enhanced penetration testing tool for XSW in arbitrary XML documents and all types of XML Signatures. This kind of tool presents a huge challenge as it should e.g. consider more difficult transformations like XPath or XSLT.

Acknowledgements

The authors would like to thank all the security teams and their developers for their cooperation, and would like to note that throughout the collaboration all the teams effected a productive and highly professional communication.

Moreover, we would like to thank Scott Cantor, David Jorm, Florian Kohlar, Christian Mainka, Christopher Meyer, Thomas Roessler, and the anonymous reviewers (of the USENIX Security Symposium and the IEEE Symposium on Security and Privacy) for their valuable remarks on the developed attacks and the paper content. Finally, we thank Alexander Bieber for the Sharepoint 2010 test bed.

This work was partially funded by the Sec2 project of the German Federal Ministry of Education and Research (BMBF, FKZ: 01BY1030).

References

- [1] *IEEE International Conference on Web Services, ICWS 2009, Los Angeles, CA, USA, 6-10 July 2009* (2009), IEEE.
- [2] ARMANDO, A., CARBONE, R., COMPAGNA, L., CUÉLLAR, J., PELLEGRINO, G., AND SORNIOTTI, A. From Multiple Credentials to Browser-Based Single Sign-On: Are We More Secure? In *Future Challenges in Security and Privacy for Academia and Industry*, J. Camenisch, S. Fischer-Hbner, Y. Murayama, A. Portmann, and C. Rieder, Eds., vol. 354 of *IFIP Advances in Information and Communication Technology*. Springer Boston, 2011.
- [3] ARMANDO, A., CARBONE, R., COMPAGNA, L., CUÉLLAR, J., AND TOBARRA, M. L. Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps. In *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, FMSE 2008*, V. Shmatikov, Ed. ACM, Alexandria and VA and USA, 2008.

- [4] BACKES, M., AND GROSS, T. Tailoring the dolev-yao abstraction to web services realities. In *SWS (2005)*, E. Damiani and H. Maruyama, Eds., ACM, pp. 65–74.
- [5] BARD, G. V. A Challenging but Feasible Blockwise-Adaptive Chosen-Plaintext Attack on SSL. In *SECURITY (2006)*, M. Malek, E. Fernández-Medina, and J. Hernando, Eds., INSTICC Press, pp. 99–109.
- [6] BENAMEUR, A., KADIR, F. A., AND FENET, S. XML Rewriting Attacks: Existing Solutions and their Limitations. In *IADIS Applied Computing 2008* (Apr. 2008), IADIS Press.
- [7] BHARGAVAN, K., FOURNET, C., AND GORDON, A. D. Verifying policy-based security for web services. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security* (2004), pp. 268–277.
- [8] BHARGAVAN, K., FOURNET, C., GORDON, A. D., AND O'SHEA, G. An advisor for web services security policies. In *SWS '05: Proceedings of the 2005 workshop on Secure web services* (New York, NY, USA, 2005), ACM, pp. 1–9.
- [9] BLEICHENBACHER, D. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs #1. In *CRYPTO (1998)*, pp. 1–12.
- [10] CANTOR, S., KEMP, J., MALER, E., AND PHILPOTT, R. Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS Standard, 15.03.2005, 2005. <http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf>.
- [11] CANTOR, S., KEMP, J., PHILPOTT, R., AND MALER, E. Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS Standard, 15.03.2005, 2005. <http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf>.
- [12] CANTOR, S., MOREH, J., PHILPOTT, R., AND MALER, E. Metadata for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS Standard, 15.03.2005, 2005. <http://docs.oasis-open.org/security/saml/v2.0/saml-metadata-2.0-os.pdf>.
- [13] CHAN, Y.-Y. Weakest link attack on single sign-on and its case in saml v2.0 web sso. In *Computational Science and Its Applications - ICCSA 2006*, M. Gavrilova, O. Gervasi, V. Kumar, C. Tan, D. Taniar, A. Lagan, Y. Mun, and H. Choo, Eds., vol. 3982 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006, pp. 507–516. 10.1007/11751595_54.
- [14] EASTLAKE, D., REAGLE, J., SOLO, D., HIRSCH, F., AND ROESSLER, T. XML Signature Syntax and Processing (Second Edition), 2008. <http://www.w3.org/TR/xmlsig-core/>.
- [15] GAJEK, S., JENSEN, M., LIAO, L., AND SCHWENK, J. Analysis of signature wrapping attacks and countermeasures. In *ICWS [1]*, pp. 575–582.
- [16] GAJEK, S., LIAO, L., AND SCHWENK, J. Breaking and fixing the inline approach. In *SWS '07: Proceedings of the 2007 ACM workshop on Secure web services* (New York, NY, USA, 2007), ACM, pp. 37–43.
- [17] GAJEK, S., LIAO, L., AND SCHWENK, J. Towards a formal semantic of xml signature. W3C Workshop Next Steps for XML Signature and XML Encryption, 2007.
- [18] GROSS, T. Security Analysis of the SAML SSO Browser/Artifact Profile. In *ACSAC (2003)*, IEEE Computer Society, pp. 298–307.
- [19] GROSS, T., AND PFITZMANN, B. SAML artifact information flow revisited. In *In IEEE Workshop on Web Services Security (WSSS) (Berkeley, May 2006)*, IEEE, pp. 84–100.
- [20] GRUSCHKA, N., AND IACONO, L. L. Vulnerable cloud: Soap message security validation revisited. In *ICWS [1]*, pp. 625–631.
- [21] GUDGIN, M., HADLEY, M., MENDELSON, N., MOREAU, J.-J., AND NIELSEN, H. F. SOAP Version 1.2 Part 1: Messaging Framework. *W3C Recommendation* (2003).
- [22] HARDING, P., JOHANSSON, L., AND KLINGENSTEIN, N. Dynamic security assertion markup language: Simplifying single sign-on. *Security Privacy, IEEE 6*, 2 (march-april 2008), 83 – 85.
- [23] JENSEN, M., LIAO, L., AND SCHWENK, J. The curse of namespaces in the domain of xml signature. In *SWS (2009)*, E. Damiani, S. Proctor, and A. Singhal, Eds., ACM, pp. 29–36.
- [24] JENSEN, M., MEYER, C., SOMOROVSKY, J., AND SCHWENK, J. On the effectiveness of xml schema validation for countering xml signature wrapping attacks. In *Securing Services on the Cloud (IWSSC), 2011 1st International Workshop on* (sept. 2011), pp. 7–13.
- [25] LUTZ, D., AND STILLER, B. Combining identity federation with payment: The saml-based payment protocol. In *Network Operations and Management Symposium (NOMS), 2010 IEEE* (april 2010), pp. 495–502.
- [26] MALER, E., AND REED, D. The venn of identity: Options and issues in federated identity management. *Security Privacy, IEEE 6*, 2 (march-april 2008), 16–23.
- [27] MCINTOSH, M., AND AUSTEL, P. XML Signature Element Wrapping Attacks and Countermeasures. In *SWS '05: Proceedings of the 2005 workshop on Secure web services* (New York, NY, USA, 2005), ACM Press, pp. 20–27.
- [28] MCINTOSH, M., AND AUSTEL, P. XML signature element wrapping attacks and countermeasures. In *Workshop on Secure Web Services* (2005).
- [29] NADALIN, A., KALER, C., MONZILLO, R., AND HALLAM-BAKER, P. Web Services Security: SOAP Message Security 1.1 (WS-Security 2004). *OASIS Standard* (2006).
- [30] RAHAMAN, M. A., MARTEN, R., AND SCHAAD, A. An inline approach for secure soap requests and early validation. OWASP AppSec Europe, 2006.
- [31] RAHAMAN, M. A., AND SCHAAD, A. Soap-based secure conversation and collaboration. In *ICWS (2007)*, pp. 471–480.
- [32] RAHAMAN, M. A., SCHAAD, A., AND RITS, M. Towards secure soap message exchange in a soa. In *Workshop on Secure Web Services* (2006).
- [33] SOMOROVSKY, J., HEIDERICH, M., JENSEN, M., SCHWENK, J., GRUSCHKA, N., AND IACONO, L. L. All Your Clouds are Belong to us – Security Analysis of Cloud Management Interfaces. In *The ACM Cloud Computing Security Workshop (CCSW) (Oct. 2011)*.
- [34] THE APACHE SOFTWARE FOUNDATION. Apache Xerces.
- [35] TSCHOFENIG, H., FALK, R., PETERSON, J., HODGES, J., SICKER, D., AND POLK, J. Using saml to protect the session initiation protocol (sip). *Network, IEEE 20*, 5 (sept.-oct. 2006), 14–17.
- [36] VAN DER VLIST, E. *XML Schema*. O'Reilly, 2002.
- [37] WAGNER, D., AND SCHNEIER, B. Analysis of the SSL 3.0 protocol. In *In Proceedings of the Second USENIX Workshop on Electronic Commerce* (1996), USENIX Association, pp. 29–40.
- [38] WANG, R., CHEN, S., AND WANG, X. Signing Me onto Your Accounts through Facebook and Google: a Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In *IEEE Symposium on Security and Privacy (Oakland), IEEE Computer Society* (May 2012).
- [39] YONG-SHENG, Z., AND JING, Y. Research of dynamic authentication mechanism crossing domains for web services based on saml. In *Future Computer and Communication (ICFCC), 2010 2nd International Conference on* (may 2010), vol. 2, pp. V2–395–V2–398.

Clickjacking: Attacks and Defenses

Lin-Shung Huang
Carnegie Mellon University
linshung.huang@sv.cmu.edu

Alex Moshchuk
Microsoft Research
alexmos@microsoft.com

Helen J. Wang
Microsoft Research
helenw@microsoft.com

Stuart Schechter
Microsoft Research
stuart.schechter@microsoft.com

Collin Jackson
Carnegie Mellon University
collin.jackson@sv.cmu.edu

Abstract

Clickjacking attacks are an emerging threat on the web. In this paper, we design new clickjacking attack variants using existing techniques and demonstrate that existing clickjacking defenses are insufficient. Our attacks show that clickjacking can cause severe damages, including compromising a user’s private webcam, email or other private data, and web surfing anonymity.

We observe the root cause of clickjacking is that an attacker application presents a sensitive UI element of a target application *out of context* to a user (such as hiding the sensitive UI by making it transparent), and hence the user is tricked to act out of context. To address this root cause, we propose a new defense, *InContext*, in which web sites (or applications) mark UI elements that are sensitive, and browsers (or OSes) enforce *context integrity* of user actions on these sensitive UI elements, ensuring that a user sees everything she should see before her action and that the timing of the action corresponds to her intent.

We have conducted user studies on Amazon Mechanical Turk with 2064 participants to evaluate the effectiveness of our attacks and our defense. We show that our attacks have success rates ranging from 43% to 98%, and our *InContext* defense can be very effective against the clickjacking attacks in which the use of clickjacking is more effective than social engineering.

1 Introduction

When multiple applications or web sites (or OS principals [44] in general) share a graphical display, they are subject to *clickjacking* [13] (also known as *UI redressing* [28, 49]) attacks: one principal may trick the user into interacting with (e.g., clicking, touching, or voice controlling) UI elements of another principal, triggering actions not intended by the user. For example, in *Likejacking* attacks [46], an attacker web page tricks users into clicking on a Facebook “Like” button by transparently overlaying it on top of an innocuous UI element,

such as a “claim your free iPad” button. Hence, when the user “claims” a free iPad, a story appears in the user’s Facebook friends’ news feed stating that she “likes” the attacker web site. For ease of exposition, our description will be in the context of web browsers. Nevertheless, the concepts and techniques described are generally applicable to all client operating systems where display is shared by mutually distrusting principals.

Several clickjacking defenses have been proposed and deployed for web browsers, but all have shortcomings. Today’s most widely deployed defenses rely on *framebusting* [21, 37], which disallows a sensitive page from being framed (i.e., embedded within another web page). Unfortunately, framebusting is fundamentally incompatible with embeddable third-party widgets, such as Facebook Like buttons. Other existing defenses (discussed in Section 3.2) suffer from poor usability, incompatibility with existing web sites, or failure to defend against significant attack vectors.

To demonstrate the insufficiency of state-of-the-art defenses, we construct three new attack variants using existing clickjacking techniques. We designed the new attack scenarios to be more damaging than the existing clickjacking attacks in the face of current defenses. In one scenario, the often-assumed web-surfing-anonymity can be compromised. In another, a user’s private data and emails can be stolen. Lastly, an attacker can spy on a user through her webcam. We have conducted the first clickjacking effectiveness study through Amazon Mechanical Turk and find that the aforementioned attacks have success rates of 98%, 47%, and 43%, respectively.

Learning from the lessons of existing defenses, we set the following design goals for our clickjacking defense:

- **Widget compatibility:** clickjacking protection should support third-party widgets.
- **Usability:** users should not be prompted for their actions.
- **Backward compatibility:** the defense should not break existing web sites (e.g., by disallowing exist-

ing functionality).

- Resiliency: our defense should address the root cause of clickjacking and be resilient to new attack vectors.

The root cause of clickjacking is that an attacker application presents a sensitive UI element of a target application *out of context* to a user and hence the user gets tricked to act out of context. For example, in the aforementioned Likejacking attack scenario, an attacker web page presents a false visual context to the user by hiding the sensitive “Like” button transparently on top of the “claim your free iPad” button.

To address the root cause and achieve the above goals, our defense, called *InContext*, lets web sites mark their sensitive UI elements and then lets browsers enforce the *context integrity* of user actions on the sensitive UI elements. The context of a user’s action consists of its visual context and temporal context.

- *Visual context* is what a user should see right before her sensitive action. To ensure visual context integrity, both the sensitive UI element and the pointer feedback (such as cursors, touch feedback, or NUI input feedback) need to be fully visible to a user. We refer to the former as *target display integrity* and to the latter as *pointer integrity*.
- *Temporal context* refers to the timing of a user action. Temporal integrity ensures that a user action at a particular time is intended by the user. For example, an attack page can compromise temporal integrity by launching a *bait-and-switch* attack by first baiting the user with a “claim your free iPad” button and then switching in a sensitive UI element right before the anticipated time of user click.

We implemented a prototype of *InContext* on Internet Explorer 9 and found that it is practical to use, adding at most 30ms of delay for verifying a click. We evaluated *InContext* through Amazon Mechanical Turk user studies, and our results show that *InContext* is very effective against attacks in which the use of clickjacking is vital to attack effectiveness.

2 Threat Model

The primary attacker against which *InContext* defends is a *clickjacking attacker*. A clickjacking attacker has all the capabilities of a web attacker [17]: (1) they own a domain name and control content served from their web servers, and (2) they can make a victim visit their site, thereby rendering attacker’s content in the victim’s browser. When a victim user visits the attacker’s page, the page hides a sensitive UI element either visually or temporally (see Section 3.1 for various techniques to achieve this) and lure the user into performing unintended UI actions on the sensitive element out of context.

We make no attempt to protect against social engineering attackers who can succeed in their attacks even when the system is perfectly designed and built. For example, a social engineering attacker can fool users into clicking on a Facebook Like button by drawing misleading content, such as images from a charity site, *around* it. Even though the Like button is not manipulated in any way, a victim may misinterpret the button as “liking” charity work rather “liking” the attacker web site, and the victim may have every intention to click on the button. In contrast, a clickjacking attacker exploits a system’s inability to maintain context integrity for users’ actions and thereby can manipulate the sensitive element visually or temporally to trick users.

3 Background and Related Work

In this section, we discuss known attacks and defenses for clickjacking, and compare them to our contributions. Below, we assume a victim user is visiting a clickjacking attacker’s page, which embeds and manipulates the *target element* residing on a different domain, such as Facebook’s Like button or PayPal’s checkout dialog.

3.1 Existing clickjacking attacks

We classify existing attacks according to three ways of forcing the user into issuing input commands out of context: (1) compromising *target display integrity*, the guarantee that users can fully see and recognize the target element before an input action; (2) compromising *pointer integrity*, the guarantee that users can rely on cursor feedback to select locations for their input events; and (3) compromising *temporal integrity*, the guarantee that users have enough time to comprehend where they are clicking.

3.1.1 Compromising target display integrity

Hiding the target element. Modern browsers support HTML/CSS styling features that allow attackers to visually hide the target element but still route mouse events to it. For example, an attacker can make the target element transparent by wrapping it in a `div` container with a CSS `opacity` value of zero; to entice a victim to click on it, the attacker can draw a decoy *under* the target element by using a lower CSS `z-index` [13]. Alternatively, the attacker may completely cover the target element with an opaque decoy, but make the decoy unclickable by setting the CSS property `pointer-events:none` [4]. A victim’s click would then fall through the decoy and land on the (invisible) target element.

Partial overlays. Sometimes, it is possible to visually confuse a victim by obscuring only a part of the target element [12, 41]. For example, attackers could overlay their own information on top of a PayPal checkout `iframe` to cover the recipient and amount fields while leaving the “Pay” button intact; the victim will thus have

incorrect context when clicking on “Pay”. This overlaying can be done using CSS `z-index` or using Flash Player objects that are made topmost with Window Mode property [2] set to `wmode=direct`. Furthermore, a target element could be partially overlaid by an attacker’s popup window [53].

Cropping. Alternatively, the attacker may crop the target element to only show a piece of the target element, such as the “Pay” button, by wrapping the target element in a new `iframe` that uses carefully chosen negative CSS position offsets and the Pay button’s width and height [41]. An extreme variant of cropping is to create multiple 1x1 pixel containers of the target element and using single pixels to draw arbitrary clickable art.

3.1.2 Compromising pointer integrity

Proper visual context requires not only the target element, but also all pointer feedback to be fully visible and authentic. Unfortunately, an attacker may violate pointer integrity by displaying a fake cursor icon away from the pointer, known as *cursorjacking*. This leads victims to misinterpret a click’s target, since they will have the wrong perception about the current cursor location. Using the CSS `cursor` property, an attacker can easily hide the default cursor and programmatically draw a fake cursor elsewhere [20], or alternatively set a custom mouse cursor icon to a deceptive image that has a cursor icon shifted several pixels off the original position [7].

Another variant of cursor manipulation involves the blinking cursor which indicates keyboard focus (e.g., when typing text into an input field). Vulnerabilities in major browsers have allowed attackers to manipulate keyboard focus using *strokejacking* attacks [50, 52]. For example, an attacker can embed the target element in a hidden frame, while asking users to type some text into a fake attacker-controlled input field. As the victim is typing, the attacker can momentarily switch keyboard focus to the target element. The blinking cursor confuses victims into thinking that they are typing text into the attacker’s input field, whereas they are actually interacting with the target element.

3.1.3 Compromising temporal integrity

Attacks in the previous two sections manipulated visual context to trick the user into sending input to the wrong UI element. An orthogonal way of achieving the same goal is to manipulate UI elements after the user has decided to click, but before the actual click occurs. Humans typically require a few hundred milliseconds to react to visual changes [34, 54], and attackers can take advantage of our slow reaction to launch timing attacks.

For example, an attacker could move the target element (via CSS position properties) on top of a decoy button shortly after the victim hovers the cursor over the

decoy, in anticipation of the click. To predict clicks more effectively, the attacker could ask the victim to repetitively click objects in a malicious game [1, 3, 54, 55] or to double-click on a decoy button, moving the target element over the decoy immediately after the first click [16, 33].

3.1.4 Consequences

To date, there have been two kinds of widespread clickjacking attacks in the wild: Tweetbomb [22] and Likejacking [46]. In both attacks, an attacker tricks victims to click on Twitter Tweet or Facebook Like buttons using hiding techniques described in Section 3.1.1, causing a link to the attacker’s site to be reposted to the victim’s friends and thus propagating the link virally. These attacks increase traffic to the attacker’s site and harvest a large number of unwitting friends or followers.

Many proof-of-concept clickjacking techniques have also been published. Although the impact of these attacks in the wild is unclear, they do demonstrate more serious damages and motivate effective defenses. In one case [38], attackers steal user’s private data by hijacking a button on the approval pages of the OAuth [10] protocol, which lets users share private resources such as photos or contacts across web sites without handing out credentials. Several attacks target the Flash Player webcam settings dialogs (shown in Figure 1), allowing rogue sites to access the victim’s webcam and spy on the user [1, 3, 9]. Other POCs have forged votes in online polls, committed click fraud [11], uploaded private files via the HTML5 File API [19], stolen victims’ location information [54], and injected content across domains (in an XSS spirit) by tricking the victim to perform a drag-and-drop action [18, 40].

3.2 Existing anti-clickjacking defenses

Although the same-origin policy [35] is supposed to protect distrusting web sites from interfering with one another, it fails to stop any of the clickjacking attacks we described above. As a result, several anti-clickjacking defenses have been proposed (many of such ideas were suggested by Zalewski [51]), and some have been deployed by browsers.

3.2.1 Protecting visual context

User Confirmation. One straightforward mitigation for preventing out-of-context clicks is to present a confirmation prompt to users when the target element has been clicked. Facebook currently deploys this approach for the Like button, asking for confirmation whenever requests come from blacklisted domains [47]. Unfortunately, this approach degrades user experience, especially on single-click buttons, and it is also vulnerable to double-click timing attacks of Section 3.1.3, which could trick the victim to click through both the target element

and a confirmation popup.

UI Randomization. Another technique to protect the target element is to randomize its UI layout [14]. For example, PayPal could randomize the position of the Pay button on its express checkout dialog to make it harder for the attacker to cover it with a decoy button. This is not robust, since the attacker may ask the victim to keep clicking until successfully guessing the Pay button's location.

Opaque Overlay Policy. The Gazelle web browser [45] forces all cross-origin frames to be rendered opaquely. However, this approach removes all transparency from *all* cross-origin elements, breaking benign sites.

Framebusting. A more effective defense is *framebusting*, or disallowing the target element from being rendered in *iframes*. This can be done either with JavaScript code in the target element which makes sure it is the top-level document [37], or with newly added browser support, using features called `X-Frame-Options` [21] and CSP's `frame-ancestors` [39]. A fundamental limitation of framebusting is its incompatibility with target elements that are intended to be framed by arbitrary third-party sites, such as Facebook Like buttons.¹ In addition, previous research found JavaScript framebusting unreliable [37], and in Section 4.2, we will show attacks that bypass framebusting protection on OAuth dialogs using popup windows. Zalewski has also demonstrated how to bypass framebusting by navigating browsing history with JavaScript [55].

Visibility Detection on Click. Instead of completely disallowing framing, an alternative is to allow rendering transparent frames, but block mouse clicks if the browser detects that the clicked cross-origin frame is not fully visible. Adobe has added such protection to Flash Player's webcam access dialog in response to webcam clickjacking attacks; however, their defense only protects that dialog and is not available for other web content.

The ClearClick module of the Firefox extension NoScript also uses this technique [23], comparing the bitmap of the clicked object on a given web page to the bitmap of that object rendered in isolation (e.g., without transparency inherited from a malicious parent element). Although ClearClick is reasonably effective at detecting visual context compromises, its on-by-default nature must assume that all cross-origin frames need clickjacking protection, which results in false positives on some sites. Due to these false positives, ClearClick prompts users to confirm their actions on suspected clickjacking attacks, posing a usability burden. An extension called

¹`X-Frame-Options` and `frame-ancestors` both allow specifying a whitelist of sites that may embed the target element, but doing so is often impractical: Facebook would have to whitelist much of the web for the Like button!

ClickIDS [5] was proposed to reduce the false positives of ClearClick by alerting users only when the clicked element overlaps with other clickable elements. Unfortunately, ClickIDS cannot detect attacks based on partial overlays or cropping, and it still yields false positives.

Finally, a fundamental limitation of techniques that verify browser-rendered bitmaps is that cursor icons are not captured; thus, pointer integrity is not guaranteed. To address this caveat, ClearClick checks the computed cursor style of the clicked element (or its ancestors) to detect cursor hiding. Unfortunately, cursor spoofing attacks can still be effective against some users even if the default cursor is visible over the target element, as discussed in Section 7.2.

3.2.2 Protecting temporal context

Although we're not aware of any timing attacks used in the wild, browser vendors have started to tackle these issues, particularly to protect browser security dialogs (e.g., for file downloads and extension installations) [34]. One common way to give users enough time to comprehend any UI changes is to impose a delay after displaying a dialog, so that users cannot interact with the dialog until the delay expires. This approach has been deployed in Flash Player's webcam access dialog, suggested in Zalewski's proposal [51], and also proposed in the Gazelle web browser [45]. In response to our vulnerability report, ClearClick has added a UI delay for cross-origin window interactions [24].

Unresponsive buttons during the UI delay have reportedly annoyed many users. The length of the UI delay is clearly a tradeoff between the user experience penalty and protection from timing attacks. Regardless, UI delay is not a complete answer to protecting temporal integrity, and we construct an attack that successfully defeats a UI delay defense in Section 4.3.

3.2.3 Access Control Gadgets

Access control gadgets (ACG) [30] were recently introduced as a new model for modern OSes to grant applications permissions to access user-owned resources such as camera or GPS. An ACG is a privileged UI which can be embedded by applications that need access to the resource represented by the ACG; authentic user actions on an ACG grant its embedding application permission to access the corresponding resource. The notion of ACGs is further generalized to application-specific ACGs, allowing applications to require authentic user actions for application-specific functionality. Application-specific ACGs precisely capture today's web widgets that demand a clickjacking defense.

ACGs require clickjacking resilience. While Roesner et al's design [30] considered maintaining both visual and temporal integrity, they did not consider pointer in-



Figure 1: **Cursor spoofing attack page.** The target Flash Player webcam settings dialog is at the bottom right of the page, with a “skip this ad” bait link remotely above it. Note there are two cursors displayed on the page: a fake cursor is drawn over the “skip this ad” link while the actual pointer hovers over the webcam access “Allow” button.

tegrity and did not evaluate various design parameters. In this work, we comprehensively address these issues, and we also establish the taxonomy of context integrity explicitly.

3.2.4 Discussion

We can conclude that all existing clickjacking defenses fall short in some way, with robustness and site compatibility being the main issues. Moreover, a glaring omission in all existing defenses is the pointer integrity attacks described in Section 3.1.2. In Section 5, we will introduce a browser defense that (1) does not require user prompts, unlike ClearClick and Facebook’s Likejacking defense, (2) provides pointer integrity, (3) supports target elements that require arbitrary third-party embedding, unlike framebusting, (4) lets sites opt in by indicating target elements, avoiding false positives that exist in ClearClick, and (5) is more robust against timing attacks than the existing UI delay techniques.

3.3 Our contributions

The major contributions of this paper are in evaluating the effectiveness of existing attack techniques as well as designing and evaluating a new defense. Our evaluation uses several new attack variants (described in Section 4) which build on existing techniques described in Section 3.1, including cursor manipulation, fast-paced object clicking, and double-click timing. Whereas most existing proof-of-concepts have focused on compromising target display integrity, we focus our analysis on pointer integrity and temporal integrity, as well as on combining several known techniques in novel ways to increase effectiveness and bypass all known defenses.

4 New Attack Variants

To demonstrate the insufficiency of state-of-the-art defenses described above, we construct and evaluate three

attack variants using known clickjacking techniques. We have designed the new attack scenarios to be potentially more damaging than existing clickjacking attacks in the face of current defenses. We describe each in turn.

4.1 Cursor spoofing attack to steal webcam access

We first crafted a cursor spoofing attack (Section 3.1.2) to steal access to a private resource of a user: the webcam. In this attack, the user is presented with an attack page shown in Figure 1. A fake cursor is programmatically rendered to provide false feedback of pointer location to the user, in which the fake cursor gradually shifts away from the hidden real cursor while the pointer is moving. A loud video ad plays automatically, leading the user to click on a “skip this ad” link. If the user moves the fake cursor to click on the skip link, the real click actually lands on the Adobe Flash Player webcam settings dialog that grants the site permission to access the user’s webcam. The cursor hiding is achieved by setting the CSS cursor property to none, or a custom cursor icon that is completely transparent, depending on browser support.

4.2 Double-click attack to steal user private data

Today’s browsers do not protect temporal integrity for web sites. We show in our second attack that even if a security-critical web page (such as an OAuth dialog page) successfully employs framebusting (refusing to be embedded by other sites), our attack can still successfully clickjack such a page by compromising temporal integrity for popup windows.

We devised a bait-and-switch double-click attack (Section 3.1.3) against the OAuth dialog for Google accounts, which is protected with X-Frame-Options. The attack is shown in Figure 2. First, the attack page baits the user to perform a double-click on a decoy button. After the first click, the attacker switches in the Google OAuth pop-up window under the cursor right before the

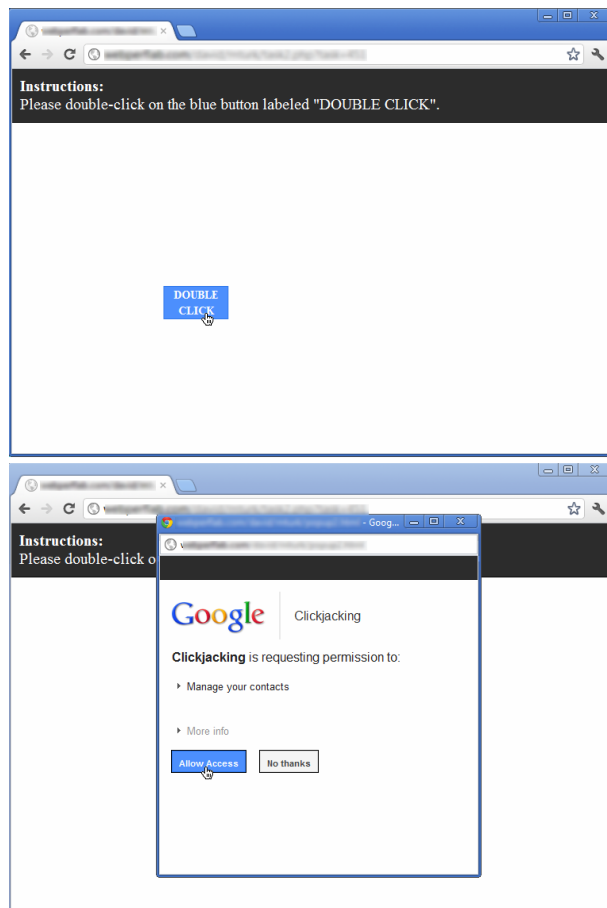


Figure 2: **Double-click attack page.** *The target OAuth dialog popup window appears underneath the pointer immediately after the first click on the decoy button.*

second click (the second half of the double-click). This attack can steal a user’s emails and other private data from the user’s Google account.

The double-click attack technique was previously discussed in the context of extension installation dialogs by Ruderman [33].

4.3 Whack-a-mole attack to compromise web surfing anonymity

In our third attack, we combine the approaches from the previous two attacks, cursor spoofing and bait-and-switch, to launch a more sophisticated whack-a-mole attack that combines clickjacking with social plugins (e.g., Facebook Like button) to compromise web surfing anonymity.

In this attack, we ask the user to play a whack-a-mole game and encourage her to score high and earn rewards by clicking on buttons shown at random screen locations as fast as possible. Throughout the game, we use a fake cursor to control where the user’s attention should be. At a later point in the game, we switch in a Facebook Like button at the real cursor’s location, tricking the user to

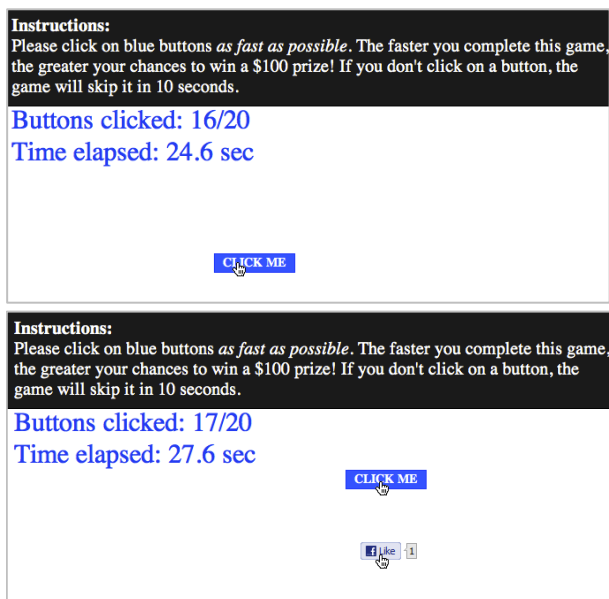


Figure 3: **Whack-a-mole attack page.** *This is a cursor spoofing variant of the whack-a-mole attack. On the 18th trial, the attacker displays the target Like button underneath the actual pointer.*

click on it.

In 2010, Wondracek et al. [48] showed that it is feasible for a malicious web site to uniquely identify 42% of social network users that use groups by exploiting browsing history leaks. Fortunately, the history sniffing technique required in their attack is no longer feasible in major browsers due to Baron’s patch [6]. However, we find that our whack-a-mole attack above, and Like-jacking attacks in general, can still easily reveal the victim’s real identity at the time of visit and compromise user anonymity in web surfing as follows.

Consider an attacker who is an admin for a Facebook page; the attacker crafts a separate malicious page which tricks users to click on his Like button. That page is notified when a victim clicks on the Like button via `FB.Event.subscribe()`, triggering the attacker’s server to pull his fan list from Facebook and instantly identify the newly added fan. The attacker’s server could then query the victim’s profile via Facebook Graph API (and remove the victim fan to avoid suspicion). While we implemented this logic as a proof-of-concept and verified its effectiveness, we did not test it on real users.

In Section 7, we show our results on the effectiveness of all these attacks on Mechanical Turk users.

5 InContext Defense

As described in Section 1, the root cause of clickjacking is that an attacker application presents a sensitive UI element of a target application *out of context* to the user, and hence the user gets tricked to act out of context.

Enforcing context integrity for an application is essentially one aspect of application isolation, in addition to memory and other resource access. Namely, the context for a user’s action in the application should be protected from manipulation by other applications. We believe it is an OS’s (or a browser’s) role to provide such cross-application (or cross-web-site) protection.

Section 1 introduced two dimensions of context integrity: visual and temporal. Enforcing visual integrity ensures that the user is presented with what she should see before an input action. Enforcing temporal integrity ensures that the user has enough time to comprehend what UI element they are interacting with.

We describe our design for each in turn.

5.1 Ensuring Visual Integrity

To ensure visual integrity at the time of a sensitive user action, the system needs to make the display of both the sensitive UI elements and the pointer feedback (such as cursors, touch feedback, or NUI input feedback) fully visible to the user. Only when both the former (*target display integrity*) and the latter (*pointer integrity*) are satisfied, the system *activates* sensitive UI elements and delivers user input to them.

5.1.1 Guaranteeing Target Display Integrity

Although it is possible to enforce the display integrity of all the UI elements of an application, doing so would make all the UI elements inactivated if any part of the UI is invisible. This would burden users to make the entire application UI unobstructed to carry out any interactions with the application. Such whole-application display integrity is often not necessary. For example, not all web pages of a web site contain sensitive operations and are susceptible to clickjacking. Since only applications know which UI elements require protection, we let web sites indicate which UI elements or web pages are *sensitive*. This is analogous to how HTML5 [43] and some browsers [32] (as well as earlier research on MashupOS [44]) allow web sites to label certain content as “sandboxed”. The sandboxed content is isolated so that it cannot attack the embedding page. In contrast, the *sensitive* content is protected with context integrity for user actions, so that the embedding page cannot click-jack the sensitive content.

We considered several design alternatives for providing target display integrity, as follows.

Strawman 1: CSS Checking. A naïve approach is to let the browser check the computed CSS styles of elements, such as the position, size, opacity and z-index, and make sure the sensitive element is not overlaid by cross-origin elements. However, various techniques exist to bypass CSS and steal topmost display, such as using IE’s `createPopUp()` method [25] or Flash Player’s Window

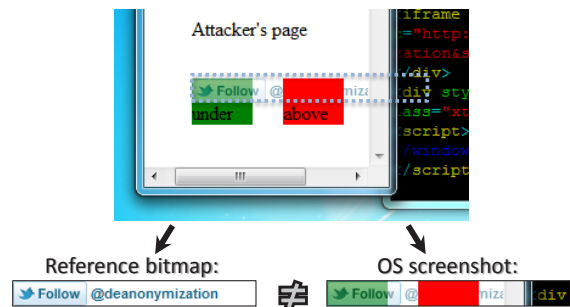


Figure 4: **Ensuring target element display integrity.** Here, the attacker violates visual context of the Twitter Follow button by changing its opacity and obstructing it with two DIVs. InContext detects this during its bitmap comparison. Obstructions from other windows are also detected (e.g., the non-browser Vi window on the right).

Mode [2]. Solely relying on CSS checking is not reliable and thus insufficient.

Strawman 2: Static reference bitmap. Another approach is to let a web site provide a static bitmap of its sensitive element as a reference, and let the browser make sure the rendered sensitive element matches the reference. Flash Player uses this approach for protecting its webcam access dialog (Section 3.2.1). However, different browsers may produce slightly differently rendered bitmaps from the same HTML code, and it would be too burdensome for developers to serve different reference bitmaps for different browsers. Furthermore, this approach fails when sensitive elements contain animated content, such as button mouseover effects, or dynamically generated content, such as the amount to pay in a checkout UI.

Our design. InContext enforces target display integrity by comparing the OS-level screenshot of the area that contains the sensitive element (what the user sees), and the bitmap of the sensitive element rendered in isolation at the time of user action. If these two bitmaps are not the same, then the user action is canceled and not delivered to the sensitive element. Figure 4 illustrates this process.

In the Likejacking attack example in Section 1, when a user clicks on the “claim your iPad” button, the transparent Facebook Like button is actually clicked, as the browser unconditionally delivered the click event to the Facebook Like button. With our defense, Facebook can label its Like button web page as “sensitive” in the corresponding HTTP response. The browser will then perform the following tasks before delivering each click event to the Like button. The browser first determines what the user sees at the position of the Like button on the screen by taking a screenshot of the browser window and cropping the sensitive element from the screenshot based on the element’s position and dimensions known by the browser. The browser then determines what the

sensitive element should look like if rendered in isolation and uses this as a reference bitmap. To this end, the browser draws the sensitive element on a blank surface and extracts its bitmap. The browser then compares the cropped screenshot with the reference bitmap. A mismatch here means that the user does not fully see the Like button but her click targets the Like button. In this case, the browser detects a potential clickjacking offense and cancels the delivery of the click event. Instead, it triggers a new `oninvalidclick` event to give the application an opportunity to deal with such occasions.

This design is resilient to new visual spoofing attack vectors because it uses only the position and dimension information from the browser layout engine to determine what the user sees. This is much easier to get right than relying on other sophisticated logic (such as CSS) from the layout engine to determine what the user sees. By obtaining the reference bitmap at the time of the user action on a sensitive UI element, this design works well with dynamic aspects (such as animations or movies) in a sensitive UI element, unlike Strawman 2 above.

We also enforce that a host page cannot apply any CSS transforms [42] (such as zooming, rotating, etc.) that affect embedded sensitive elements; any such transformations will be ignored by InContext-capable browsers. This will prevent malicious zooming attacks [36], which change visual context via zoom. We also disallow any transparency inside the sensitive element itself. Although doing so may have a compatibility cost in terms of preventing legitimate blending effects of the sensitive element with the host page, we believe this is a necessary restriction, since otherwise attackers could violate visual context by inserting decoys that could show through the sensitive element.

Our bitmap comparison is similar to ClearClick (Section 3.2.1), with two crucial differences: (1) We use OS APIs to take a screenshot of the browser window, rather than relying on the browser to generate screenshots, making it more robust to rendering performed by Flash Player and other plug-ins, and (2) our approach is opt-in, eliminating false positives and obviating user prompts.

5.1.2 Guaranteeing Pointer Integrity

Without pointer integrity support, an attacker could spoof the real pointer. For example, an attack page may show a fake cursor to shift the user's attention from the real cursor and cause the user to act out of context by not looking at the destination of her action. To mitigate this, we must ensure that users see system-provided (rather than attacker-simulated) cursors and pay attention to the right place before interacting with a sensitive element.

For our design, we consider the following techniques, individually and in various combinations, to under-

stand the tradeoff between their effectiveness of stopping pointer-spoofing attacks and intrusiveness to users. Some of the techniques limit the attackers' ability to carry out pointer-spoofing attacks; others draw attention to a particular place on the screen.

No cursor customization. Current browsers disallow cross-origin cursor customization. We further restrict this policy: when a sensitive element is present, InContext disables cursor customization on the host page (which embeds the sensitive element) and on all of the host's ancestors, so that a user will always see the system cursor in the areas surrounding the sensitive element.

Our opt-in design is better than completely disallowing cursor customization, because a web site may want to customize the pointer for its own UIs (i.e., same-origin customization). For example, a text editor may want to show different cursors depending on whether the user is editing text or selecting a menu item.

Screen freezing around sensitive element. Since humans typically pay more attention to animated objects than static ones [15], attackers could try to distract a user away from her actions with animations. To counter this, InContext "freezes" the screen (i.e., ignores rendering updates) around a sensitive UI element when the cursor enters the element.

Muting. Sound could also draw a user's attention away from her actions. For example, a voice may instruct the user to perform certain tasks, and loud noise could trigger a user to quickly look for a way to stop the noise. To stop sound distractions, we mute the speakers when a user interacts with sensitive elements.

Lightbox around sensitive element. Greyout (also called Lightbox) effects are commonly used for focusing the user's attention on a particular part of the screen (such as a popup dialog). In our system, we apply this effect by overlaying a dark mask on all rendered content *around* the sensitive UI element whenever the cursor is within that element's area. This causes the sensitive element to stand out visually.

The mask cannot be a static one. Otherwise, an attacker could use the same static mask in its application to dilute the attention-drawing effect of the mask. Instead, we use a randomly generated mask which consists of a random gray value at each pixel.

No programmatic cross-origin keyboard focus changes. To stop strokejacking attacks that steal keyboard focus (see Section 3.1.2), once the sensitive UI element acquires keyboard focus (e.g., for typing text in an input field), we disallow programmatic changes of keyboard focus by other origins.

Discussion. This list of techniques is by no means exhaustive. For example, sensitive elements could also draw the user's attention with splash animation effects on the cursor or the element [15].

Our goal was to come up with a representative set of techniques with different security and usability tradeoffs, and conduct user studies to evaluate their effectiveness as a design guide. We hope that this methodology can be adopted by browser vendors to evaluate a wider range of techniques with a larger-scale user study for production implementations.

5.2 Ensuring Temporal Integrity

Even with visual integrity, an attacker can still take a user’s action out of context by compromising its temporal integrity, as described in Section 3.1.3. For example, a timing attack could bait the user with a “claim your free iPad” button and then switch in a sensitive UI element (with visual integrity) at the expected time of user click. The bait-and-switch attack is similar to time-of-check-to-time-of-use (TOCTTOU) race conditions in software programs. The only difference is that the race condition happens to a human rather than a program. To mitigate such TOCTTOU race conditions on users, we impose the following constraints for a user action on a sensitive UI element:

UI delay. We apply this existing technique (discussed in Section 3.2.2) to only deliver user actions to the sensitive element if the visual context has been the same for a minimal time period. For example, in the earlier bait-and-switch attack, the click on the sensitive UI element will not be delivered unless the sensitive element (together with the pointer integrity protection such as grey-out mask around the sensitive element) has been fully visible and stationary long enough. We evaluate trade-offs of a few delays in Section 7.3.

UI delay after pointer entry. The UI delay technique above is vulnerable to the whack-a-mole attack (Section 4.3) that combines pointer spoofing with rapid object clicking. A stronger variant on the UI delay is to impose the delay not after changes to visual context, but each time the pointer *enters* the sensitive element. Note that the plain UI delay may still be necessary, e.g., on touch devices which have no pointer.

Pointer re-entry on a newly visible sensitive element.

In this novel technique, when a sensitive UI element first appears or is moved to a location where it will overlap with the current location of the pointer, an InContext-capable browser invalidates input events until the user explicitly moves the pointer from the outside of the sensitive element to the inside. Note that an alternate design of automatically moving the pointer outside the sensitive element could be misused by attackers to programmatically move the pointer, and thus we do not use it. Obviously, this defense only applies to devices and OSes that provide pointer feedback.

Padding area around sensitive element. The sensitive UI element’s padding area (i.e., extra whitespace separat-

Sensitive Element	Dimensions	Click Delay	Memory Overhead
Facebook Like	90x20 px	12.04 ms	5.11 MB
Twitter Follow	200x20 px	13.54 ms	8.60 MB
Animated GIF (1.5 fps)	468x60 px	14.92 ms	7.90 MB
Google OAuth	450x275 px	24.78 ms	12.95 MB
PayPal Checkout	385x550 px	30.88 ms	15.74 MB

Table 1: **Performance of InContext.** For each sensitive element, this table shows extra latency imposed on each click, as well as extra memory used.

ing the host page from the embedded sensitive element) needs to be thick enough so that a user can clearly decide whether the pointer is on the sensitive element or on its embedding page. As well, this ensures that during rapid cursor movements, such as those in the whack-a-mole attack (Section 4.3), our pointer integrity defenses such as screen freezing are activated early enough. Sections 7.2 and 7.4 give a preliminary evaluation on some padding thickness values. The padding could be either enforced by the browser or implemented by the developer of the sensitive element; we have decided the latter is more appropriate to keep developers in control of their page layout.

5.3 Opt-in API

In our design, web sites must express which elements are sensitive to the browser. There are two options for the opt-in API: a JavaScript API and an HTTP response header. The JavaScript API’s advantages include ability to detect client support for our defense as well as to handle `oninvalidclick` events raised when clickjacking is detected. On the other hand, the header approach is simpler as it doesn’t require script modifications, and it does not need to deal with attacks that disable scripting on the sensitive element [37]. We note that bitmap comparison functions should not be directly exposed in JavaScript (and can only be triggered by user-initiated actions). Otherwise, they might be misused to probe pixels across origins using a transparent frame.

6 Prototype Implementation

We built a prototype of InContext using Internet Explorer 9’s public COM interfaces. We implemented the pixel comparison between an OS screenshot and a sensitive element rendered on a blank surface to detect element visibility as described in Section 5.1.1, using the GDI `BitBlt` function to take desktop screenshots and using the MSHTML `IHTMLRender` interface to generate reference bitmaps.

To implement the UI delays, we reset the UI delay timer whenever the top-level window is focused, and whenever the computed position or size of the sensitive element has changed. We check these conditions whenever the sensitive element is repainted, *before* the actual

paint event; we detect paint events using IE binary behaviors [27] with the `IHTMLPainter::Draw` API. We also reset the UI delay timer whenever the sensitive element becomes fully visible (e.g., when an element obscuring it moves away) by using our visibility checking functions above. When the user clicks on the sensitive element, InContext checks the elapsed time since the last event that changed visual context.

Our prototype makes the granularity of sensitive elements to be HTML documents (this includes iframes); alternately, one may consider enabling protection for finer-grained elements such as DIVs. For the opt-in mechanism, we implemented the Javascript API of Section 5.3 using the `window.external` feature of IE.

Although our implementation is IE and Windows-specific, we believe these techniques should be feasible in other browsers and as well. For example, most platforms support a screenshot API, and we found an API similar to IE's `IHTMLRender` in Firefox to render reference bitmaps of an HTML element.

At this time, we did not implement the pointer integrity defenses, although we have evaluated their effects in Section 7.

Performance. To prove that InContext is practical, we evaluated our prototype on five real-world sensitive elements (see Table 1). For each element, we measured the memory usage and click processing time for loading a blank page that embeds each element in a freshly started browser, with and without InContext, averaging over ten runs. Our testing machine was equipped with Intel Xeon CPU W3530 @ 2.80 GHz and 6 GB of RAM.

Without additional effort on code optimization, we find that our average click processing delay is only 30 ms in the worst case. This delay is imposed only on clicks on sensitive elements, and should be imperceptible to most users. We find that the majority (61%) of the click delay is spent in the OS screenshot functions (averaging 11.65 ms). We believe these could be significantly optimized, but this is not our focus in this paper.

7 Experiments

7.1 Experimental design

In February of 2012 we posted a Human Interactive Task (HIT) at Amazon's Mechanical Turk to recruit prospective participants for our experiments. Participants were offered 25 cents to "follow the on-screen instructions and complete an interactive task" by visiting the web site at which we hosted our experiments. Participants were told the task would take roughly 60 seconds. Each task consisted of a unique combination of a simulated attack and, in some cases, a simulated defense. After each attack, we asked a series of follow-up questions. We then disclosed the existence of the attack and explained that since it was simulated, it could only result in clicking on harmless

simulated functionality (e.g., a fake Like button).

We wanted participants to behave as they would if lured to a third-party web site with which they were previously unfamiliar. We hosted our experiments at a web site with a domain name unaffiliated with our research institution so as to ensure that participants' trust (or distrust) in our research institution would not cause them to behave in a more (or less) trusting manner.

For attacks targeting Flash Player and access to video cameras (webcams), we required that participants have Flash Player installed in their browser and have a webcam attached. We used a SWF file to verify that Flash Player was running and that a webcam was present. For attacks loading popup windows, we required that participants were not using IE or Opera browsers since our attack pages were not optimized for them.

We recruited a total of 3521 participants.² Participants were assigned uniformly and at random to one of 27 (between-subjects) treatment groups. There were 10 treatment groups for the cursor-spoofing attacks, 4 for the double-click attacks, and 13 for the whack-a-mole attacks. Recruiting for all treatments in parallel eliminated any possible confounding temporal factors that might result if different groups were recruited or performed tasks at different times. We present results for each of these three sets of attacks separately.

In our analysis, we excluded data from 370 participants who we identified (by worker IDs) have previously participated in this experiment or earlier versions of it. We also discarded data from 1087 participants who were assigned to treatment groups for whack-a-mole attacks that targeted Facebook's Like button but who could not be confirmed as being logged into Facebook (using the technique described in [8]). In Tables 2, 3 and 4, we report data collected from the remaining 2064 participants.

Except when stated otherwise, we use a two-tailed Fisher's Exact Test when testing whether differences between attack rates in different treatment groups are significant enough to indicate a difference in the general population. This test is similar to χ^2 , but more conservative when comparing smaller sample sizes.

7.2 Cursor-spoofing attacks

In our first experiment, we test the efficacy of the cursor-spoofing attack page, described in Section 4.1 and illustrated in Figure 1, and of the pointer integrity defenses we proposed in Section 5.1.2. The results for each treatment group make up the rows of Table 2. The columns show the number of users that clicked on the "Skip ad" link (Skip), quit the task with no pay (Quit), clicked on

²The ages of our participants were as follows: 18-24 years: 46%; 25-34 years: 38%; 35-44 years: 11%; 45-54 years: 3%; 55-64 years: 1%; 65 years and over: 0.5%. A previous study by Ross et al. provides an analysis of the demographics of Mechanical Turk workers [31].

Treatment Group	Total	Timeout	Skip	Quit	Attack Success
1. Base control	68	26	35	3	4 (5%)
2. Persuasion control	73	65	0	2	6 (8%)
3. Attack	72	38	0	3	31 (43%)
4. No cursor styles	72	34	23	3	12 (16%)
5a. Freezing ($M=0$ px)	70	52	0	7	11 (15%)
5b. Freezing ($M=10$ px)	72	60	0	3	9 (12%)
5c. Freezing ($M=20$ px)	72	63	0	6	3 (4%)
6. Muting + 5c	70	66	0	2	2 (2%)
7. Lightbox + 5c	71	66	0	3	2 (2%)
8. Lightbox + 6	71	60	0	8	3 (4%)

Table 2: **Results of the cursor-spoofing attack.** Our attack tricked 43% of participants to click on a button that would grant webcam access. Several of our proposed defenses reduced the rate of clicking to the level expected if no attack had occurred.

webcam “Allow” button (Attack success), and those who watched the ad full video and were forwarded to the end of the task with no clicks (Timeout).

Control. We included a control group, Group 1, which contained an operational skip button, a Flash webcam access dialog, but no attack to trick the user into clicking the webcam access button while attempting to click the skip button. We included this group to determine the click rate that we would hope a defense could achieve in countering an attack. We anticipated that some users might click on the button to grant webcam access simply out of curiosity. In fact, four did. We were surprised that 26 of the 68 participants waited until the full 60 seconds of video completed, even though the “skip ad” button was available and had not been tampered with. In future studies, we may consider using a video that is longer, more annoying, and that does not come from a charity that users may feel guilty clicking through.

We added a second control, Group 2, in which we removed the “skip ad” link and instructed participants to click on the target “Allow” button to skip the video ad. This control represents one attempt to persuade users to grant access to the webcam without tricking them. As with Group 1, we could consider a defense successful if rendered attacks no more successful than using persuasion to convince users to allow access to the webcam.

Whereas 4 of 68 (5%) participants randomly assigned to the persuasion-free control treatment (Group 1) clicked on the “Allow” button, we observed that 6 of 73 (8%) participants assigned to the persuasion control treatment did so. However, the difference in the attack success rates of Group 1 and Group 2 were not significant, with a two-tailed Fisher’s exact test yielding $p=0.7464$.

Attack. Participants in Group 3 were exposed to the simulated cursor spoofing attack, with no defenses to protect them. The attack succeeded against 31 of 72 participants (43%). The difference in the attack success rates between participants assigned to the non-persuasion control treat-

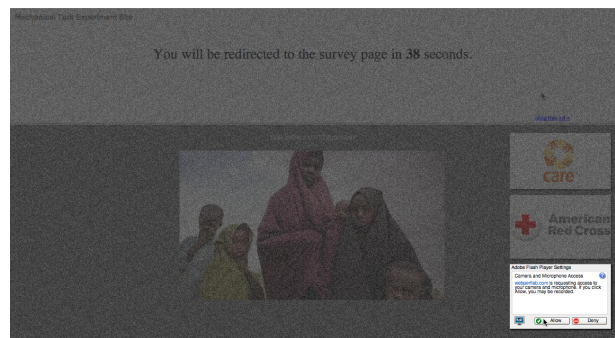


Figure 5: **Cursor-spoofing attack with lightbox defenses.** The intensity of each pixel outside of the target element is darkened and randomized when the actual pointer hovers on the target element.

ment (Group 1) and the attack treatment (Group 3) is statistically significant ($p<0.0001$). The attack might have been even more successful had participants been given a more compelling motivation to skip the “skip this ad” link. Recall that only 51% of participants in the non-persuasion control treatment (Group 1) tried to skip the animation. If we assume the same percent of participants tried to skip the advertisement during the attack, then 84% of those participants who tried to skip the ad fell for the attack (43% of 51%).

Defenses. One straightforward defense against cursor-spoofing attacks is to disallow cursor customization. This would prevent the real cursor from being hidden, though the attack page could still draw a second, fake cursor. Some victims might focus on the wrong cursor and fall for the attack. In Group 4, we disallowed cursor customization and found that 12 of 72 (16%) participants still fell for the attack. This result, along with attacker’s ability to draw multiple fake cursors and emphasize one that is not the real cursor, suggest this defense has limited effectiveness. Nevertheless, the defense does appear to make a dent in the problem, as there is a reduction in attack success rates from Group 3 (43%), without the defense, to Group 4 (16%), with the defense, and the difference between these two treatment groups was statistically significant ($p=0.0009$).

In Groups 5a-c, we deployed the freezing defense described in Section 5.1.2: when this defense triggers, all movement outside the protected region, including the video and fake cursor, is halted. This helps break the illusion of the fake cursor and draws the user’s attention to the part of the screen on which there is still movement—that which contains the real cursor. The freezing effect will not help if users have already initiated a click before noticing it. We thus initiate the freeze when the cursor is within M pixels of the webcam dialog, for M of 0, 10, and 20 pixels. At $M=20$ px (Group 5c), the attack success rate dropped to that of our non-persuasion control group,

Treatment Group	Total	Timeout	Quit	Attack Success
1. Attack	90	46	1	43 (47%)
2a. UI Delay ($T_A=250\text{ms}$)	91	89	0	2 (2%)
2b. UI Delay ($T_A=500\text{ms}$)	89	86	2	1 (1%)
3. Pointer re-entry	88	88	0	0 (0%)

Table 3: **Results of double-click attack.** 43 of 90 participants fell for the attack that would grant access to their personal Google data. Two of our defenses stopped the attack completely.

tricking only 3 of 72 (4%). Fewer participants assigned to the 20px-border-freezing defense of Group 5c fell for the attack (4%) than those in the cursor-customization-defense treatment of Group 4 (16%), and this difference was significant ($p=0.0311$).

Given the efficacy of the large-margin (20px) freezing defense in Group 5c, and the low rate of successful attacks on which to improve, our sample was far too small to detect any further benefits that might result from muting the speaker or freezing portions of the screen with a lightbox might provide. Augmenting the freezing defense to mute the computer’s speaker (Group 6) yielded a similar attack success rate of 2 of 70 (2%) participants. Augmenting that defense again with a lightbox, greying over the frozen region as described in Section 5.1.2, (Groups 7 and 8) also resulted in attack success rates of 2-4%. The lightbox effect is a somewhat jarring user experience, and our experiments do not provide evidence that this user-experience cost is offset by a measurably superior defense. However, larger sample sizes or different attack variants may reveal benefits that our experiment was unable to uncover.

7.3 Double-click attacks

In our second experiment, we tested the efficacy of the double-click timing attack (described in Section 4.2 and shown in Figure 2) and the defenses proposed in Section 5.2. The attack attempts to trick the user into clicking on the “Allow Access” button of a Google OAuth window by moving it underneath the user’s cursor after the first click of a double-click on a decoy button. If the “Allow” button is not clicked within two seconds, the attack times out without success (column Timeout). The results of each treatment group appear as rows of Table 3. **Attack.** Of the 90 participants assigned to the treatment in which they were exposed to the simulated attack without any defense to protect them, the attack was successful against 43 of them (47%). If this had been a real attack, we could have accessed their Gmail to read their personal messages or download their contacts. Furthermore, many of the users who were not successfully attacked escaped because the popup was not shown quickly enough. Indeed, the popup took more than 500ms to be displayed for 31 out of 46 users who timed out on the attack (with 833ms average loading time for those users)—

likely greater than a typical user’s double-click speed. The attack efficacy could likely be improved further by pre-loading the OAuth dialog in a *pop-under* window (by de-focusing the popup window) and refocusing the pop-under window between the two clicks; this would avoid popup creation cost during the attack.

Defenses. Two groups of participants were protected by simulating the UI delay defense described in Section 5.2—we treated clicks on the “Allow” button as invalid until after it has been fully visible for a threshold of T_A ms. We assigned a treatment group for two choices for T_A : 250ms (Group 2a), the mode of double-click intervals of participants in an early mouse experiment [29] in 1984, and 500ms (Group 2b), the default double-click interval in Windows (the time after which the second click would be counted as a second click, rather than the second half of a double-click) [26]. We observed that the delay of 250ms was effective, though it was not long enough for 2 out of 91 (2%) participants in Group 2a, who still fell for the attack. The difference in attack success rates between the attack treatment (Group 1) and the UI delay defense treatment for $T_A=250\text{ms}$ (Group 2a) was significant ($p<0.0001$). Similarly, the 500ms delay stopped the attack for all but 1 of 89 (1%) participants in Group 2b.

We also simulated our *pointer re-entry* defense (Group 3), which invalidated UI events on the OAuth dialog until the cursor has explicitly transitioned from outside of the OAuth dialog to inside. This defense was 100% effective for 88 participants in Group 3. The difference in attack success rates between the attack treatment (Group 1) and the pointer re-entry defense treatment (Group 3) was significant ($p<0.0001$). While the attack success rate reduction from the delay defense (Groups 2a and 2b) to the pointer re-entry defense (Group 3) was not statistically significant, the pointer re-entry defense is preferable for other reasons; it does not constrain the timing with which users can click on buttons, and it cannot be gamed by attacks that might attempt to introduce delays—one can imagine an attack claiming to test the steadiness of a user’s hand by asking him to move the mouse to a position, close his eyes, and press the button after five seconds.

7.4 Whack-a-mole attacks

Next, we tested the efficacy of the fast-paced button clicking attack, the *whack-a-mole* attack, described in Section 4.3 and shown in Figure 3. In attempt to increase the attack success rates, as a real attacker would do, we offered a \$100 performance-based prize to keep users engaged in the game. In this experiment, we used the Facebook’s Like button as the target element (except for Group 1b, where for the purposes of comparison, the Flash Player webcam settings dialog was also

Treatment Group	Total	Timeout	Quit	Attack Success	Attack Success (On 1st Mouseover)	Attack Success (Filter by Survey)
1a. Attack without clickjacking	84	1	0	83 (98%)	N/A	42/43 (97%)
1b. Attack without clickjacking (webcam)	71	1	1	69 (97%)	N/A	13/13 (100%)
2. Attack with timing	84	3	1	80 (95%)	80 (95%)	49/50 (98%)
3. Attack with cursor-spoofing	84	0	1	83 (98%)	78 (92%)	52/52 (100%)
4a. Combined defense ($M=0px$)	77	0	1	76 (98%)	42 (54%)	54/54 (100%)
4b. Combined defense ($M=10px$)	78	10	1	67 (85%)	27 (34%)	45/53 (84%)
4c. Combined defense ($M=20px$)	73	18	4	51 (69%)	12 (16%)	31/45 (68%)
5. Lightbox + 4c	73	21	0	52 (71%)	10 (13%)	24/35 (68%)
6a. Entry delay ($T_E=250ms$) + 4c	77	27	4	46 (59%)	6 (7%)	27/44 (61%)
6b. Entry delay ($T_E=500ms$) + 4c	73	25	3	45 (61%)	3 (4%)	31/45 (68%)
6c. Entry delay ($T_E=1000ms$) + 4c	71	25	1	45 (63%)	1 (1%)	25/38 (65%)
6d. Entry delay ($T_E=500ms$) + 4a	77	6	0	71 (92%)	16 (20%)	46/49 (93%)
7. Lightbox + 6b	73	19	0	54 (73%)	6 (8%)	34/46 (73%)

Table 4: Results of the whack-a-mole attack.

98% of participants were vulnerable to Likejacking de-anonymization under the attack that combined whack-a-mole with cursor-spoofing. Several defenses showed a dramatic drop in attack success rates, reducing them to as low as 1% when filtered by first mouseover events.

tested). We checked whether the participant was logged into Facebook [8] and excluded data from users that were not logged in. The results for each treatment group appear in the rows of Table 4. The “Timeout” column represents those participants who did not click on the target button within 10 seconds, and were thus considered to not have fallen for the attack.

We calculated the attack success rate with three different methods, presented in three separate columns. The first Attack Success column shows the total number of users that clicked on the Like button. However, after analyzing our logs, we realized that this metric is not necessarily accurate: many people appeared to notice the Like button and moved their mouse around it for several seconds before eventually deciding to click on it. For these users, it was not clickjacking that ultimately caused the attack, but rather it was the users’ willingness to knowingly click on the Like button after noticing it (e.g., due to wanting to finish the game faster, or deciding that they did not mind clicking it, perhaps not understanding the consequences). For the purposes of evaluating our defense, we wanted to filter out these users: our defenses are only designed to stop users from clicking on UI elements *unknowingly*.

We used two different filters to try to isolate those victims who clicked on the Like button unknowingly. The first defined an attack to be successful if and only if the victim’s cursor entered the Like button only once before the victim click. This *on first mouseover* filter excludes victims who are moving their mouse around the Like button and deliberating whether or not to click. The second filter uses responses from our post-task survey to exclude participants who stated that they noticed the Like button and clicked on it knowingly, shown in column “Attack Success (Filter by Survey)”. We asked the participants the following questions, one at a time, revealing each question after the previous question was answered:

1. Did you see the Facebook Like button at any point

- in this task? <displayed an image of Like button>
- (If No to 1) Would you approve if your Facebook wall showed that you like this page?
 - (If Yes to 1) Did you click on the Like button?
 - (If Yes to 3) Did you intend to click on the Like button?

We only included participants who either did not approve “liking” (No to 2), were not aware that they “liked” (No to 3) or did not intend to “like” (No to 4). This excludes victims who do not care about “liking” the attacker’s page and who intentionally clicked on the Like button. We expected the two filters to yield similar results; however, as we describe later, the trust in our survey responses was reduced by indications that participants lied in their answers. Therefore, we rely on the *on first mouseover* column for evaluating and comparing our defenses.

Attacks. We assigned two treatment groups to a simulated whack-a-mole attack that did not employ clickjacking. The first (Group 1a) eventually were shown a Like button to click on whereas the second (Group 1b) were eventually shown the “allow” button in the Flash webcam access dialog. In the simulated attack, participants first had to click on a myriad of buttons, many of which were designed to habituate participants into ignoring the possibility that these buttons might have context outside their role in the game. These included buttons that contained the text “great,” “awesome,” and smiley face icons. On the attack iteration, the Like button simply appeared to be the next target object to press in the game. We hypothesized that users could be trained to ignore the semantics usually associated with a user interface element if it appeared within this game.

Though we had designed this attack, its efficacy surprised even us. The Like button version of Group 1a succeeded on 83 of 84 (98%) participants and the “allow” button of Group 1b succeeded on 69 of 71 (97%) participants. The differences between these two groups are not

statistically significant. The attacks were also so effective that, at these sample sizes, they left no room in which to find statistically significant improvements through the use of clickjacking.

In the whack-a-mole attack with timing (Group 2), the Like button is switched to cover one of the game buttons at a time chosen to anticipate the user's click. This attack was also effective, fooling 80 of 84 (95%) participants in Group 2. Next, we combined the timing technique with cursor spoofing that we also used in Section 7.2, so that the game is played with a fake cursor, with the attack (Group 3) succeeding on 83 of 84 (98%) participants.

Defenses. In Groups 4a-c, we combined the proposed defenses that were individually effective against the previous cursor-spoofing and the double-click attacks, including pointer re-entry, appearance delay of $T_A=500$ ms, and display freezing with padding area size $M=0$ px, 10px and 20px. We assumed that the attacker could be aware of our defenses; e.g., our attack compensated for the appearance delay by substituting the Like button roughly 500ms before the anticipated user click.

Using no padding area ($M=0$ px), the attack succeeded on the first mouseover on 42 of 77 (54%) of the participants in Group 4a. The reduction in the first-mouseover-success rate from Group 3 (without defense) to 4a (with the $M=0$ px combined defense) was statistically significant, with $p<0.0001$. So, while all of the participants in Group 4a eventually clicked on the Like button, the defense caused more users to move their mouse away from the Like button before clicking on it. Increasing the padding area to $M=10$ px (Group 4b) further reduced the first-mouseover success rate to 27 of 78 (34%), and the maximum padding area tested ($M=20$ px, Group 4c) resulted in a further reduction to 12 of 73 (16%). The reduction in the first-mouseover attack success rates between Groups 4a and 4b was statistically significant ($p=0.0155$), as was the reduction from Groups 4b to 4c ($p=0.0151$). We also noticed that adding a 10px padding area even reduced the unfiltered attack success rate from 76 of 77 (98%) in Group 4a to 67 of 78 (85%) in Group 4b, and a 20px padding area further reduced the unfiltered attack success rate to 51 of 73 (69%) in Group 4c. The reduction in the unfiltered attack success rates between Groups 4a and 4b was also statistically significant ($p=0.0046$), as was the reduction from Groups 4b to 4c ($p=0.0191$). Thus, larger padding areas provide noticeably better clickjacking protection. Participants assigned to Group 5 received the defense of 4c enhanced with a lightbox, which further decreased the first-mouseover attack effectiveness to 10 of 73 (13%). The difference in first-mouseover success rates between Group 4c and 5 was not statistically significant ($p=0.8176$).

Note that there is a large discrepancy comparing first-mouseover attack success to the survey-filtered attack

success. After analyzing our event logs manually, we realized that many users answered our survey questions inaccurately. For example, some people told us that they didn't click on the Like button, and they wouldn't approve clicking on it, whereas the logs show that while their initial click was blocked by our defense, they continued moving the mouse around for several seconds before finally resolving to click the Like button. While these users' answers suggested that clickjacking protection should have stopped them, our defenses clearly had no chance of stopping these kinds of scenarios.

Participants assigned to Groups 6a-d were protected by the pointer-entry delay defense described in Section 5.2: if the user clicks within a duration of T_E ms of the pointer entering the target region, the click is invalid. In Groups 6a and 6b, we experiment with a pointer entry delay of $T_E=250$ ms and $T_E=500$ ms, respectively. We used an appearance delay of $T_A=500$ ms and a padding area of $M=20$ px as in Group 4c. In both cases, we observed that the addition of pointer entry delay was highly effective. Only 3 of 73 (4%) participants in Group 6b still clicked on the target button. We found a significant difference in attack success rate between Groups 4c and 6b ($p=0.0264$), indicating that the pointer entry delay helps stopping clickjacking attacks, compared to no pointer entry delays. We then test a more extreme pointer entry delay of $T_E=1000$ ms, in which the appearance delay T_A must also be adjusted to no less than 1000ms. This was most successful in preventing clickjacking from succeeding: only 1 of 71 (1%) participants fell for the attack. We also tested the pointer entry delay $T_E=500$ ms without a padding area ($M=0$ px), which allowed 16 of 77 (20%) participants in Group 6d to fall for the attack. Note that the difference in first-mouseover success rates between Groups 6b and 6d was significant ($p=0.0026$). Again, our results suggest that attacks are much more effective when there is no padding area around the target. Finally, in Group 7 we tested the lightbox effect in addition to Group 6b. The attack succeeded on 6 of 73 (8%) participants in Group 7, in which the difference between Groups 6b and 7 was not statistically significant ($p=0.4938$).

Overall, we found that pointer entry delay was crucial in reducing the first-mouseover success rate, the part of the attack's efficacy that could potentially be addressed by a clickjacking defense. Thus, it is an important technique that should be included in a browser's clickjacking protection suite, alongside freezing with a sufficiently large padding area, and the pointer re-entry protection. The pointer entry delay subsumes, and may be used in place of, the appearance delay. The only exception would be for devices that have no pointer feedback; having an appearance delay could still prove useful against a whack-a-mole-like touch-based attack.

7.5 Ethics

The ethical elements of our study were reviewed as per our research institution's requirements. No participants were actually attacked in the course of our experiments; the images they were tricked to click appeared identical to sensitive third-party embedded content elements, but were actually harmless replicas. However, participants may have realized that they had been tricked and this discovery could potentially lead to anxiety. Thus, after the simulated attack we not only disclosed the attack but explained that it was simulated.

8 Conclusion

We have devised new clickjacking attack variants, which bypass existing defenses and cause more severe harm than previously known, such as compromising webcams, user data, and web surfing anonymity.

To defend against clickjacking in a fundamental way, we have proposed InContext, a web browser or OS mechanism to ensure that a user's action on a sensitive UI element is in context, having visual integrity and temporal integrity.

Our user studies on Amazon Mechanical Turk show that our attacks are highly effective with success rates ranging from 43% to 98%. Our InContext defense can be very effective for clickjacking attacks in which the use of clickjacking improves the attack effectiveness.

This paper made the following contributions:

- We provided a survey of existing clickjacking attacks and defenses.
- We conducted the first user study on the effectiveness of clickjacking attacks.
- We introduced the concept of *context integrity* and used it to define and characterize clickjacking attacks and their root causes.
- We designed, implemented, and evaluated InContext, a set of techniques to maintain context integrity and defeat clickjacking.

With all these results, we advocate browser vendors and client OS vendors to consider adopting InContext.

Acknowledgments

We are grateful to Adam Barth, Dan Boneh, Elie Bursztein, Mary Czerwinski, Carl Edlund, Rob Ennals, Jeremiah Grossman, Robert Hansen, Brad Hill, Eric Lawrence, Giorgio Maone, Jesse Ruderman, Sid Stamm, Zhenbin Xu, Michal Zalewski, and the Security and Privacy Research Group at Microsoft Research for reviewing and providing feedback on this work.

References

- [1] F. Aboukhadijeh. HOW TO: Spy on the Webcams of Your Website Visitors. <http://www.feross.org/webcam-spy/>, 2011.
- [2] Adobe. Flash OBJECT and EMBED tag attributes. http://kb2.adobe.com/cps/127/tn_12701.html, 2011.
- [3] G. Aharonovsky. Malicious camera spying using ClickJacking. <http://blog.guya.net/2008/10/07/malicious-camera-spying-using-clickjacking/>, 2008.
- [4] L. C. Aun. Clickjacking with pointer-events. <http://jsbin.com/imuca>.
- [5] M. Balduzzi, M. Egele, E. Kirda, D. Balzarotti, and C. Kruegel. A solution for the automated detection of clickjacking attacks. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 2010.
- [6] D. Baron. Preventing attacks on a user's history through CSS :visited selectors. <http://dbaron.org/mozilla/visited-privacy>, 2010.
- [7] E. Bordi. Proof of Concept - CursorJacking (noScript). <http://static.vulnerability.fr/noscript-cursorjacking.html>.
- [8] M. Cardwell. Abusing HTTP Status Codes to Expose Private Information. https://grepular.com/Abusing_HTTP_Status_Codes_to_Expose_Private_Information, 2011.
- [9] J. Grossman. Clickjacking: Web pages can see and hear you. <http://jeremiahgrossman.blogspot.com/2008/10/clickjacking-web-pages-can-see-and-hear.html>, 2008.
- [10] E. Hammer-Lahav. The OAuth 1.0 Protocol. RFC 5849 (Informational), Apr. 2010.
- [11] R. Hansen. Stealing mouse clicks for banner fraud. <http://ha.ckers.org/blog/20070116/stealing-mouse-clicks-for-banner-fraud/>, 2007.
- [12] R. Hansen. Clickjacking details. <http://ha.ckers.org/blog/20081007/clickjacking-details/>, 2008.
- [13] R. Hansen and J. Grossman. Clickjacking. <http://www.sectheory.com/clickjacking.htm>, 2008.
- [14] B. Hill. Adaptive user interface randomization as an anti-clickjacking strategy. http://www.thesecuritypractice.com/the_security_practice/papers/AdaptiveUserInterfaceRandomization.pdf, May 2012.
- [15] R. Hoffmann, P. Baudisch, and D. S. Weld. Evaluating visual cues for switching windows on large screens. In *Proceedings of the 26th annual SIGCHI conference on Human factors in computing systems*, 2008.
- [16] L.-S. Huang and C. Jackson. Clickjacking attacks unresolved. <http://mayscript.com/blog/david/clickjacking-attacks-unresolved>, 2011.
- [17] C. Jackson. Improving browser security policies. PhD thesis, Stanford University, 2009.
- [18] K. Kotowicz. Exploiting the unexploitable XSS with clickjacking. <http://blog.kotowicz.net/2011/03/exploiting-unexploitable-xss-with.html>, 2011.
- [19] K. Kotowicz. Filejacking: How to make a file server from your browser (with HTML5 of course). <http://blog.kotowicz.net/2011/04/how-to->

- make-file-server-from-your.html, 2011.
- [20] K. Kotowicz. Cursorjacking again. <http://blog.kotowicz.net/2012/01/cursorjacking-again.html>, 2012.
- [21] E. Lawrence. IE8 Security Part VII: ClickJacking Defenses. <http://blogs.msdn.com/b/ie/archive/2009/01/27/ie8-security-part-vii-clickjacking-defenses.aspx>, 2009.
- [22] M. Mahemoff. Explaining the “Don’t Click” Clickjacking Tweetbomb. <http://softwareas.com/explaining-the-dont-click-clickjacking-tweetbomb>, 2009.
- [23] G. Maone. Hello ClearClick, Goodbye Clickjacking! <http://hackademix.net/2008/10/08/hello-clearclick-goodbye-clickjacking/>, 2008.
- [24] G. Maone. Fancy Clickjacking, Tougher NoScript. <http://hackademix.net/2011/07/11/fancy-clickjacking-tougher-noscript/>, 2011.
- [25] Microsoft. createPopup Method. [http://msdn.microsoft.com/en-us/library/ms536392\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ms536392(v=vs.85).aspx).
- [26] Microsoft. SetDoubleClickTime function. [http://msdn.microsoft.com/en-us/library/windows/desktop/ms646263\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms646263(v=vs.85).aspx).
- [27] Microsoft. Implementing Binary DHTML Behaviors. [http://msdn.microsoft.com/en-us/library/ie/aa744100\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ie/aa744100(v=vs.85).aspx), 2012.
- [28] M. Niemietz. UI Redressing: Attacks and Countermeasures Revisited. In *CONFidence*, 2011.
- [29] L. A. Price. Studying the mouse for CAD systems. In *Proceedings of the 21st Design Automation Conference*, 1984.
- [30] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *IEEE Symposium on Security and Privacy*, 2012.
- [31] J. Ross, L. Irani, M. S. Silberman, A. Zaldivar, and B. Tomlinson. Who are the crowdworkers?: shifting demographics in mechanical turk. In *Proceedings of the 28th International Conference On Human Factors In Computing Systems*, 2010.
- [32] J. Rossi. Defense in depth: Locking down mashups with HTML5 Sandbox. <http://blogs.msdn.com/b/ie/archive/2011/07/14/defense-in-depth-locking-down-mash-ups-with-html5-sandbox.aspx?Redirected=true>, 2011.
- [33] J. Ruderman. Bug 162020 - pop up XPInstall/security dialog when user is about to click. https://bugzilla.mozilla.org/show_bug.cgi?id=162020, 2002.
- [34] J. Ruderman. Race conditions in security dialogs. <http://www.squarefree.com/2004/07/01/race-conditions-in-security-dialogs/>, 2004.
- [35] J. Ruderman. The Same Origin Policy. <http://www.mozilla.org/projects/security/components/same-origin.html>, 2011.
- [36] G. Rydstedt, E. Bursztein, and D. Boneh. Framing attacks on smart phones and dumb routers: Tap-jacking and geo-localization. In *USENIX Workshop on Offensive Technologies*, 2010.
- [37] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *Proceedings of the Web 2.0 Security and Privacy*, 2010.
- [38] S. Sclafani. Clickjacking & OAuth. <http://stephensclafani.com/2009/05/04/clickjacking-oauth/>, 2009.
- [39] S. Stamm, B. Sterne, and G. Markham. Reining in the web with content security policy. In *Proceedings of the 19th International Conference on World Wide Web*, 2010.
- [40] P. Stone. Next generation clickjacking. In *Black Hat Europe*, 2010.
- [41] E. Vela. About CSS Attacks. <http://sirdarckcat.blogspot.com/2008/10/about-css-attacks.html>, 2008.
- [42] W3C. CSS 2D Transforms. <http://www.w3.org/TR/css3-2d-transforms/>, 2011.
- [43] W3C. HTML5, 2012. <http://www.w3.org/TR/html5/>.
- [44] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and Communication Abstractions in MashupOS. In *ACM Symposium on Operating System Principles*, 2007.
- [45] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The Multi-Principal OS Construction of the Gazelle Web Browser. In *Proceedings of the 18th Conference on USENIX Security Symposium*, 2009.
- [46] Wikipedia. Likejacking. <http://en.wikipedia.org/wiki/Clickjacking#Likejacking>.
- [47] C. Wisniewski. Facebook adds speed bump to slow down likejackers. <http://nakedsecurity.sophos.com/2011/03/30/facebook-adds-speed-bump-to-slow-down-likejackers/>, 2011.
- [48] G. Wondracek, T. Holz, E. Kirda, and C. Kruegel. A practical attack to de-anonymize social network users. In *Proceedings of the 31th IEEE Symposium on Security and Privacy*, 2010.
- [49] M. Zalewski. Browser security handbook. [http://code.google.com/p/browsersec/wiki/Part2#Arbitrary_page_mashups_\(UI_redressing\)](http://code.google.com/p/browsersec/wiki/Part2#Arbitrary_page_mashups_(UI_redressing)).
- [50] M. Zalewski. Firefox focus stealing vulnerability (possibly other browsers). <http://seclists.org/fulldisclosure/2007/Feb/226>, 2007.
- [51] M. Zalewski. [whatwg] Dealing with UI redress vulnerabilities inherent to the current web. <http://lists.whatwg.org/pipermail/whatwg-whatwg.org/2008-September/016284.html>, 2008.
- [52] M. Zalewski. The curse of inverse strokejacking. <http://lcamtuf.blogspot.com/2010/06/curse-of-inverse-strokejacking.html>, 2010.
- [53] M. Zalewski. Minor browser UI nitpicking. <http://seclists.org/fulldisclosure/2010/Dec/328>, 2010.
- [54] M. Zalewski. On designing UIs for non-robots. <http://lcamtuf.blogspot.com/2010/08/on-designing-uis-for-non-robots.html>, 2010.
- [55] M. Zalewski. X-Frame-Options, or solving the wrong problem. <http://lcamtuf.blogspot.com/2011/12/x-frame-options-or-solving-wrong.html>, 2011.

Privilege Separation in HTML5 Applications

Devdatta Akhawe, Prateek Saxena, Dawn Song
University of California, Berkeley
{devdatta,prateeks,dawnsong}@cs.berkeley.edu

Abstract

The standard approach for privilege separation in web applications is to execute application components in different web origins. This limits the practicality of privilege separation since each web origin has financial and administrative cost. In this paper, we propose a new design for achieving effective privilege separation in HTML5 applications that shows how applications can cheaply create arbitrary number of components. Our approach utilizes standardized abstractions already implemented in modern browsers. We do not advocate any changes to the underlying browser or require learning new high-level languages, which contrasts prior approaches. We empirically show that we can retrofit our design to real-world HTML5 applications (browser extensions and rich client-side applications) and achieve reduction of 6x to 10000x in TCB for our case studies. Our mechanism requires less than 13 lines of application-specific code changes and considerably improves auditability.

1 Introduction

Applications written with JavaScript, HTML5 and CSS constructs (called HTML5 applications) are becoming ubiquitous. Rich web applications and web browser extensions are examples of HTML5 applications that already enjoy massive popularity [1, 2]. The introduction of browser operating systems [3, 4], and support for HTML5 applications in classic operating systems [5, 6] herald the convergence of web and desktop applications. However, web vulnerabilities are still pervasive in emerging web applications and browser extensions [7], despite immense prior research on detection and mitigation techniques [8–12].

Privilege separation is an established security primitive for providing an important second line of defense [13]. Commodity OSes enable privilege separated applications via isolation mechanisms such as LXC [14], seccomp [15], SysTrace [16]. Traditional applications have utilized these for increased assurance and security. Some well-known examples include OpenSSH [17],

QMail [18] and Google Chrome [19]. In contrast, privilege separation in web applications is harder and comes at a cost. If an HTML5 application wishes to separate its functionality into multiple isolated components, the same-origin policy (SOP) mandates that each component execute in a separate web origin.¹ Owning and maintaining multiple web origins has significant practical administrative overheads.² As a result, in practice, the number of origins available to a single web application is limited. Web applications cannot use the same-origin policy to isolate every new component they add into the application. At best, web applications can only utilize sub-domains for isolating components, which does *not* provide proper isolation, due to special powers granted to sub-domains in the cookie and document.domain behaviors.

Recent research [12, 20] and modern HTML5 platforms, such as the Google Chrome extension platform (also used for “packaged web applications”), have recognized the need for better privilege separation in HTML5 applications. These systems advocate re-architecting the underlying browser or OS platform to force HTML5 applications to be divided into a fixed number of components. For instance, the Google Chrome extension framework requires that extensions have three components, each of which executes with different privileges [19]. Similarly, recent research proposes to partition HTML5 applications in “N privilege rings”, similar to the isolation primitives supported by x86 processors [12]. We observe two problems with these approaches. First, the fixed limit on the number of partitions or components creates an artificial and unnecessary limitation. Differ-

¹Browsers isolate applications based on their origins. An origin is defined as the tuple `<scheme, host, port>`. In recent browser extension platforms, such as in Google Chrome, each extension is assigned a unique public key as its web origin. These origins are assigned and fixed at the registration time.

²To create new origins, the application needs to either create new DNS domains or run services at ports different from port 80 and 443. New domains cost money, need to be registered with DNS servers and are long-lived. Creating new ports for web services does not work: first, network firewalls block atypical ports and Internet Explorer doesn’t include the port in determining an application’s origin

ent applications require differing number of components, and a “one-size-fits-all” approach does not work. We show that, as a result, HTML5 applications in such platforms have large amounts of code running with unnecessary privileges, which increases the impact from attacks like cross-site scripting. Second, browser re-design has a built-in deployment and adoption cost and it takes significant time before applications can enjoy the benefits of privilege separation.

In this paper, we rethink how to achieve privilege separation in HTML5 applications. In particular, we propose a solution that does not require any platform changes and is orthogonal to privilege separation architectures enforced by the underlying browsers. Our proposal utilizes standardized primitives available in today’s web browsers, requires no additional web domains and improves the auditability of HTML5 applications. In our proposal, HTML5 applications can create an arbitrary number of “unprivileged components.” Each component executes in its own *temporary origin* isolated from the rest of the components by the SOP. For any privileged call, the unprivileged components communicate with a “privileged” (parent) component, which executes in the main (permanent) origin of the web application. The privileged code is small and we ensure its integrity by enforcing key security invariants, which we define in Section 3. The privileged code mediates all access to the critical resources granted to the web application by the underlying browser platform, and it enforces a fine-grained policy on all accesses that can be easily audited. Our proposal achieves the same security benefits in ensuring application integrity as enjoyed by desktop applications with process isolation and sandboxing primitives available in commodity OSes [14–16].

We show that our approach is practical for existing HTML5 applications. We retrofit two widely used Google Chrome extensions and a popular HTML5 application for SQL database administration to use our design. In our case studies, we show that the amount of trusted code running with full privileges reduces by a factor of 6 to 10000. Our architecture does not sacrifice any performance as compared to alternative approaches that re-design the underlying web browser. Finally, our migration of existing applications requires minimal changes to code. For example, in porting our case studies to this new design we changed no more than 13 lines of code in any application. Developers do not need to learn new languages or type safety primitives to migrate code to our architecture, in contrast to recent proposals [21]. We also demonstrate strong data confinement policies. To encourage adoption, we have released our core infrastructure code as well as the case studies (where permitted) and made it all freely available online [22]. We are currently collaborating with the Google Chrome team to

apply this approach to secure Chrome applications, and our design has influenced the security architecture of upcoming Chrome applications.

In our architecture, HTML5 applications can define more expressive policies than supported by existing HTML5 platforms, namely the Chrome extension platform [19] and the Windows 8 Metro platform [5]. Google Chrome and Windows 8 rely on applications declaring install-time permissions that end users can check [23]. Multiple studies have found permission systems to be inadequate: the bulk of popular applications run with powerful permissions [24, 25] and users rarely check install-time permissions [26]. In our architecture, policy code is explicit and clearly separated, can take into account runtime ordering of privileged accesses, and can be more fine-grained. This design enables expert auditors, such as maintainers of software application galleries, to reason about the security of applications. In our case studies, these policies are typically a small amount of static JavaScript code, which is easily auditable.

2 Problem and Approach Overview

Traditional HTML applications execute with the authority of their “web origin” (protocol, port, and domain). The browser’s same origin policy (SOP) isolates different web origins from one another and from the file system. However, applications rarely rely on domains for isolation, due to the costs associated with creating new domains or origins.

In more recent application platforms, such as the Google Chrome extension platform [23], Chrome packaged web application store [1] and Windows 8 Metro applications [5], applications can execute with enhanced privileges. These privileges, such as access to the geo-location, are provided by the underlying platform through *privileged APIs*. Applications utilizing these privileged API explicitly declare their *permissions* to use privileged APIs at install time via manifest files. These applications are authored using the standard HTML5 features and web languages (like JavaScript) that web applications use; we use the term *HTML5 applications* to collectively refer to web applications and the aforementioned class of emerging applications.

Install-time manifests are a step towards better security. However, these platforms still limit the number of application components to a finite few and rely on separate origins to isolate them. For example, each Google Chrome extension has three components. One component executes in the origin of web sites that the extension interacts with. A second component executes with the extension’s permanent origin (a unique public key assigned to it at creation time). The third component executes in an all-powerful origin having the authority of the web browser. In this section, we show how this limits the

degree of privilege separation for HTML5 applications in practice.

2.1 Issues with the Current Architecture

In this section, we point out two artifacts of today’s HTML5 applications: *bundling* of privileges and *TCB inflation*. We observe that these issues are rooted in the fact that, in these designs, the ability to create new web origins (or security principals) is severely restricted.

Common vulnerabilities (like XSS and mixed content) today actually translate to powerful gains for attackers in current architectures. Recent findings corroborate the need for better privilege separation—for instance, 27 out of 100 Google Chrome extensions (including the top 50) recently studied have been shown to have exploitable vulnerabilities [7]. These attacks grant powerful privileges like code execution in *all* HTTP and HTTPS web sites and access to the user’s browsing history.

As a running example, we introduce a hypothetical extension for Google Chrome called ScreenCap. ScreenCap is an extension for capturing screenshots that also includes a rudimentary image editor to annotate and modify the image before sending to the cloud or saving to a disk.

Bundling. The ScreenCap extension consists of two functionally disjoint components: a screenshot capturing component and an image editor. In the current architecture, both the components run in the same principal (origin), despite requiring disjoint privileges. We call this phenomenon *bundling*. The screenshot component requires the `tabs` and `<all_urls>` permission, while the image editor only requires the `pictureLibrary` permission to save captured images to the user’s picture library on the cloud.

Bundling causes over-privileged components. For example, the image editor component runs with the powerful `tabs` and `<all_urls>` permission. In general, if an application’s components require privilege sets $\alpha_1, \alpha_2, \dots$, *all* components of the application run with the privileges $\bigcup \alpha_i$, leading to over-privileging. As we show in Section 5.4, 19 out of the Top 20 extensions for the Google Chrome platform exhibit bundling. As discussed earlier, this problem manifests on the web too.

TCB inflation. Privileges in HTML5 are ambient—all code in a principal runs with full privileges of the principal. In reality, only a small application core needs access to these privileges and rest of the application does not need to be in the trusted computing base (TCB). For example, the image editor in ScreenCap consists of a number of complex and large UI and image manipulation libraries. All this JavaScript code runs with the ambient privilege to write to the user’s picture library. Note that this is in addition to it running bundled with the privi-

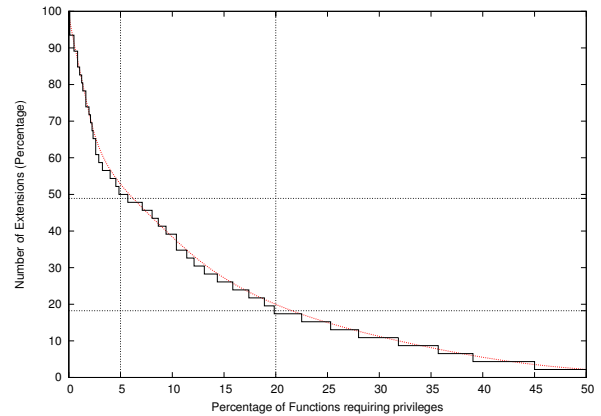


Figure 1: CDF of percentage of functions in an extension that make privileged calls (X axis) vs. the fraction of extensions studied (in percentage) (Y axis). The lines for 50% and 20% of extensions as well as for 5% and 20% of functions are marked.

leges of the screenshot component.

We measured the TCB inflation for the top 50 Chrome extensions. Figure 1 shows the percentage of total functions in an extension requiring privileges as a fraction of the total number of static functions. In half the extensions studied, less than 5% of the functions actually need any privileges. In 80% of the extensions studied, less than 20% of the functions require any privileges.

Summary. It is clear from our data that HTML5 applications, like Chrome extensions, do not sufficiently isolate their sub-components. The same-origin policy equates web origins and security principals, and web origins are fixed at creation time or tied to the web domain of the application. All code from a given provider runs under a single principal, which forces privileges to be ambient. Allowing applications to cheaply create as many security principals as necessary and to confine them with fine-grained, flexible policies can make privilege separation more practical.

Ideally, we would like to isolate the image editor component from the screenshot component, and give each component exactly the privileges it needs. Moving the complex UI and image manipulation code to an unprivileged component can tremendously aid audit and analysis. Our first case study (Section 5.1) discusses un-bundling and TCB reduction on a real world screenshot application. We achieved a 58x TCB reduction.

2.2 Problem Statement

Our goal is to design a new architecture for privilege separation that side-steps the problem of scarce web origins and enables the following properties:

Reduced TCB. Given the pervasive nature of code injection vulnerabilities, we are interested, instead, in reducing the TCB.

Ease of Audit. Dynamic code inclusion and use of complex JS constructs is pervasive. An architecture that eases audits, in spite of these issues, is necessary.

Flexible policies. Current manifest mechanisms provide insufficient contextual data for meaningful security policies. A separate flexible policy mechanism can ease audits and analysis.

Reduce Over-privileging. Bundling of disjoint applications in the same origin results in over-privileging. We want an architecture that can isolate applications agnostic of origin.

Ease of Use. For ease of adoption, we also aim for minimal compatibility costs for developers. Mechanisms that would involve writing applications for a new platform are outside scope.

Scope. We focus on the threat of vulnerabilities in *benign* HTML5 application. We aim to enable a privilege separation architecture that benign applications can utilize with ease to provide a strong second line of defense. We consider malicious applications as out of scope, but our design improves auditability and may be applicable to HTML5 malware in the future.

This paper strictly focuses on mechanisms for achieving privilege separation and on mechanisms for expressive policy-based confinement. Facilitating policy development and checking if policies are reasonable is an important issue, but beyond the scope of this paper.

3 Design

We describe our privilege separation architecture in this section. We describe the key security invariants we maintain in Section 3.2 and the mechanisms we use for enforcing them in Section 3.3.

3.1 Approach Overview

We advocate a design that is independent of any privilege separation scheme enforced by the underlying browser. In our design, HTML5 applications have one *privileged parent* component, and can have an arbitrary number of *unprivileged children*. Each child component is spawned by the parent and it executes in its own *temporary* origin. These temporary origins are created on the fly for each execution and are destroyed after the child exits; we detail how temporary origins can be implemented using modern web browsers primitives in Section 3.3. The privileged parent executes in the main (permanent) origin assigned to the HTML5 application, typically the web origin for traditional web application. The same origin policy isolates unprivileged children from one another and from the privileged parent. Figure 2 shows our proposed HTML5 application architecture. In our design, applications can continue to be authored in existing web languages like JavaScript, HTML and CSS. As a result,

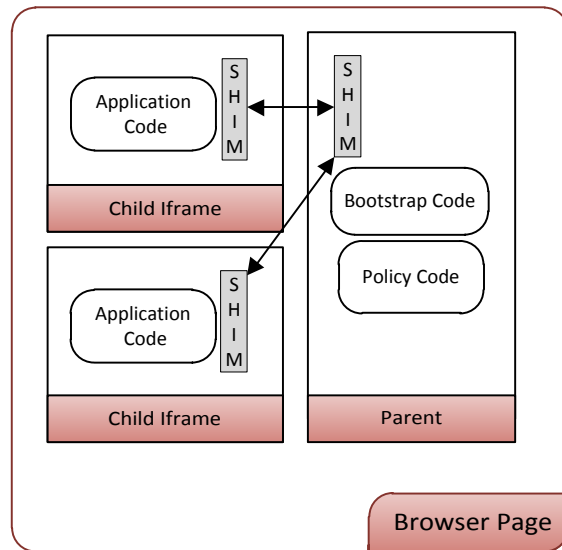


Figure 2: High-level design of our proposed architecture.

our design maintains compatibility and facilitates adoption.

Parent. Our design ensures the integrity of the privileged parent by maintaining a set of key *security invariants* that we define in Section 3.2. The parent guards access to a powerful API provided by the underlying platform, such as the Google Chrome extension API. For making any privileged call or maintaining persistent data, the unprivileged children communicate with the parent over a thin, well-defined *messaging interface*. The parent component has three components:

- *Bootstrap Code.* When a user first navigates to the HTML5 application, a portion of the parent code called the bootstrap code executes. Bootstrap code is the *unique* entry point for the application. The bootstrap code downloads the application source, spawns the unprivileged children in separate temporary origins, and controls the lifetime of their execution. It also includes boilerplate code to initialize the messaging interface in each child before child code starts executing. Privileges in HTML5 applications are tied to origins; thus, a temporary origin runs with no privileges. We explain temporary origins further in Section 3.3.
- *Parent Shim.* During their execution, unprivileged children can make privileged calls to the parent. The parent shim marshals and unmarshals these requests to and from the children. The parent shim also presents a usable interface to the policy code component of the parent.
- *Policy Code.* The policy code enforces an application-specific policy on *all* messages received

from children. Policy code decides whether to allow or disallow access to privileged APIs, such as access to the user's browsing history. This mechanism provides complete mediation on access to privileged APIs and supports fine-grained policies, similar to system call monitors in commodity OSes like Sys-Trace [16]. In addition, as part of the policy code, applications can define additional restrictions on the privileges of the children, such as disabling import of additional code from the web.

Only the policy code is application-specific; the bootstrap and parent shim are the same across all applications. To ease adoption, we have made the application-independent components available online. The application independent components need to be verified once for correctness and can be reused for all application in the future. For new applications using our design, only the application's policy code needs to be audited. In our experimental evaluation, we find that the parent code is typically only a small fraction of the rest of the application and our design invariants make it statically auditable.

Children. Our design moves all functional components of the application to the children. Each child consists of two key components:

- *Application Code.* Application code starts executing in the child after the bootstrap code initializes the messaging interface. All the application logic, including code to handle visual layout of the application, executes in the unprivileged child; the parent controls no visible area on the screen. This implies that all dynamic HTML (and code) rendering operations execute in the child. Children are allowed to include libraries and code from the web and execute them. Vulnerabilities like XSS or mixed content bugs (inclusion of HTTP scripts in HTTPS domains) can arise in child code. In our threat model, we assume that children may be compromised during the application's execution.
- *Child Shim.* The parent includes application independent shim code into the child to seamlessly allow privileged calls to the parent. This is done to keep compatibility with existing code and facilitate porting applications to our design. Shim code in the child defines wrapper functions for privileged APIs (e.g., the Google Chrome extension API [27]). The wrapper functions forward any privileged API calls as messages to the parent. The parent shim unmarshals these messages, checks the integrity of the message and executes the privileged call if allowed by the policy. The return value of the privileged API call is marshaled into messages by the parent shim and returned to the child shim. The child shim

unmarshals the result and returns it to the original caller function in the child. Certain privileged API functions take callbacks or structured data objects; in Section 4.1 we outline how our mechanism proxies these transparently. Together, the parent and child shim hide the existence of the privilege boundary from the application code.

3.2 Security Invariants

Our security invariants ensure the integrity and correctness of code running in the parent with full privileges. We do not restrict code running in the child; our threat model assumes that unprivileged children can be compromised any time during their execution. We enforce four security invariants on the parent code:

1. The parent cannot convert any string to code.
2. The parent cannot include external code from the web.
3. The parent code is the only entry point into the privileged origin.
4. Only primitive types (specifically, strings) cross the privilege boundary.

The first two invariants help increase assurance in the parent code. Together, they disable dynamic code execution and import of code from the web, which eliminates the possibility of XSS and mixed content vulnerabilities in parent code. Furthermore, it makes parent code statically auditable and verifiable. Several analysis techniques can verify JavaScript when dynamic code execution constructs like `eval` and `setTimeout` have been syntactically eliminated [9–11, 28, 29].

Invariant 3 ensures that *only* the trusted parent code executes in the privileged origin; no other application code should execute in the permanent origin. The naive approach of storing the unprivileged (child) code as a HTML file on the server suffers from a subtle but serious vulnerability. An attacker can directly navigate to the unprivileged code. Since it is served from the same origin as the parent, it will execute with full privileges of the parent without going through the parent's bootstrap mechanism. To prevent such escalation, invariant 3 ensures that all entry points into the application are directed only through the bootstrap code in the parent. Similarly, no callbacks to unprivileged code are passed to the privileged API—they are proxied by parent functions to maintain Invariant 3. We detail how we enforce this invariant in Section 3.3.

Privilege separation, in and of itself, is insufficient to improve security. A problem in privilege-separated C applications is the exchange of pointers across the privilege boundary, leading to possible errors [30, 31]. While

JavaScript does not have C-style pointers, it has first-class functions. Exchanging functions and objects across the privilege boundary can introduce security vulnerabilities. Invariant 4 eliminates such attacks by requiring that only primitive strings are exchanged across the privilege boundary.

3.3 Mechanisms

We detail how we implement the design and enforce the above invariants in this section. Whenever possible, we rely on browser's mechanisms to declaratively enforce the outlined invariants, thereby minimizing the need for code audits.

Temporary Origins. To isolate components, we execute unprivileged children in separate `iframes` sourced from temporary origins. A temporary origin can be created by assigning a fresh, globally unique identifier that the browser guarantees will never be used again [32]. A temporary origin does not have any privileges, or in other words, it executes with null authority. The globally unique nature means that the browser isolates every temporary origin from another temporary origin, as well as the parent. The temporary origin only lasts as long as the lifetime of the associated `iframe`.

Several mechanisms for implementing temporary origins are available in today's browsers, but these are rarely found in use on the web. In the HTML5 standard, `iframes` with the `sandbox` directive run in a temporary origin. This primitive is standardized and already supported in shipping versions of Google Chrome/ChromeOS, Safari, Internet Explorer/Windows 8, and a patch for Mozilla Firefox is in the final stages of review [33].

Enforcement of Security Invariants. To enforce security invariants 1 and 2 in the parent, our implementation utilizes the Content Security Policy (CSP) [34]. CSP is a new specification, already supported in Google Chrome and Firefox, that defines browser-enforced restrictions on the resources and execution of application code. In our case studies, it suffices to use the CSP policy directive `default-src 'none'; script-src 'self'`—this disables *all* constructs to convert strings into code (**Invariant 1**) and restricts the source of all scripts included in the page to the origin of the application (**Invariant 2**). We find that application-specific code is typically small (5 KB) and easily auditable in our case studies. On platforms on which CSP is not supported, we point out that disabling code evaluation constructs and external code import is possible by syntactically restricting the application language to a subset of JavaScript [11, 28, 29].

We require that all non-parent code, when requested, is sent back as a text file. Browsers do not execute text files—the code in the text files can only

execute if downloaded and executed by the parent, via the bootstrap mechanism. This ensures **Invariant 3**. In case of pure client-side platforms like Chrome, this involves a simple file renaming from `.html` to `.txt`. In case of classic client-server web applications, this involves returning a `Content-Type` header of `text/plain`. To disable mime-sniffing, we also set the `X-Content-Type-Options` HTTP header to `nosniff`.

Messaging Interface. We utilize standard primitives like `XMLHttpRequest` and the DOM API for downloading the application code and executing it in an `iframe`. We rely on the `postMessage` API for communication across the privilege boundary. `postMessage` is an asynchronous, cross-domain, purely client-side messaging mechanism. By design, `postMessage` only accepts primitive strings. This ensures **Invariant 4**.

Policy. Privilege separation isolates the policy and the application logic. Policies, in our design, are written in JavaScript devoid of any dynamic evaluation constructs and are separated from the rest of the complex application logic. Permissions on existing browser platforms are granted at install-time. In contrast, our design allows for more expressive and fine-grained policies like granting and revoking privileges at run-time. For example, in the case of ScreenCap, a child can get the ability to capture a screenshot only once and only after the user clicks the 'capture' button. Such fine-grained policies require the policy engine to maintain state, reason about event ordering and have the ability to grant/revoke fine-grained privileges. Our attempt at expressive policies is along the line of active research in this space [21], but in contrast to existing proposals, it does not require developers to specify policies in new high-level languages. Our focus is on mechanisms to support expressive policies; determining what these policies should be for applications is beyond the scope of this paper.

Additional Confinement of Child Code. By default, no restrictions are placed on the children beyond those implied by use of temporary origins. Specifically, the child does *not* inherit the parent's CSP policy restrictions. In certain scenarios, the application developer *may* choose to enforce additional restrictions on the child code, via an appropriate CSP policy on the child `iframe` at the time of its creation by the parent code. For example, in the case of ScreenCap, the screenshot component can be run under the `script-src 'self'`. This increases assurance by disabling inline scripts and code included from the web, making XSS and mixed content attacks impossible. The policy code can then grant the powerful privilege of capturing a screenshot of a user's webpage to a high assurance screenshot component.

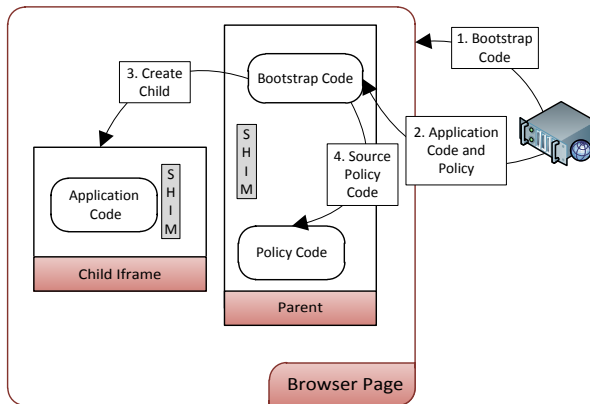


Figure 3: Sequence of events to run application in sandbox. Note that only the bootstrap code is sent to the browser to execute. Application code is sent directly to the parent, which then creates a child with it.

```

var sb_content = "<html><head>";
sb_content += "<meta http-equiv='X-WebKit-CSP'";
//csp_policy is defined in downloaded policy
sb_content += "content='"+csp_policy+"'>";
sb_content += "<script src='"+childShimSrc+"'>";
//the baseUrl is current window uri
//so that relative URIs work
sb_content += "<base href='"+baseUrl+"'>";
//contents of app.txt
sb_content += application_code;
// attribute values are URI-decoded
// by HTML parser
sb_content = encodeURIComponent(sb_content);
var fr = document.createElement("iframe");
fr.src = "data:text/html;charset=utf-8," +
sb_content;
//sandboxed frames run in fresh origin
fr.setAttribute('sandbox', 'allow-scripts');
document.body.appendChild(fr);

```

Listing 1: Bootstrap Code (JavaScript)

4 Implementation

As outlined in Section 3, the parent code executes when the user navigates to the application. The bootstrap code is in charge of creating an unprivileged sandbox and executing the unprivileged application code in it. The shim code and policy also run in the parent, but we focus on the bootstrap and shim code implementation in this section. The unprivileged child code and the security policy vary for each application, and we discuss these in our case studies (Section 5).

Figure 3 outlines the steps involved in creating one unprivileged child. First, the user navigates to the application and the parent's bootstrap code starts executing (**Step 1** in Figure 3). In **Step 2**, the parent's bootstrap code retrieves the application HTML code (as plain text files) as well as the security policy of the application. For client-side platforms like Chrome and Windows 8, this is a local file retrieval.

The parent proceeds to create a temporary origin, unprivileged iframe using the downloaded code as the source (**Step 3**, Figure 3). Listing 1 outlines the code to create the unprivileged temporary origin. The parent builds up the child's HTML in the `sb_content` variable. The parent can optionally include content restrictions on the child via a CSP policy, as explained in Section 3.3. Creating multiple children is a simple repetition of the step 3.

The parent also sources the child shim into the child iframe. The parent concatenates the child's code (HTML) and URI-encodes it all into a variable called `sb_content`. The parent creates an `iframe` with `sb_content` as the data: URI source, sets the `sandbox` attribute and appends the `iframe` to the document. The parent code also inserts a `base` HTML tag that enables relative URIs to work seamlessly.

`data:` is a URI scheme that enables references to inline data as if it were an external reference. For example, an `iframe` with `src` attribute set to `data:text/html;Hi` is similar to an `iframe` pointing to an HTML page containing only the text 'Hi'. Recall our enforcement mechanism for Invariant 3: the application code is a text file. The use of `data:` is necessary to convert text to code that the `iframe src` can point to, without storing unprivileged application code as HTML or JavaScript files.

4.1 API Shims

Recall that the child executes in a temporary origin, without the privileges needed for making privileged calls like `chrome.tabs.captureVisibleTab`. Privileged API calls in the original child code would fail when it executes in a temporary origin; our transformation should, therefore, take additional steps to preserve the original functionality of the application. In our design, we propose API shims to proxy calls to privileged API in the child to the parent code safely and transparently.

The child shim defines wrapper objects in the child that proxy a privileged call to the parent. The aim of the parent and child shim is to make the privilege separation boundary transparent. We have implemented shims for all the privileged API functions needed for our case studies. This implementation of the parent shim is 5.46 KB and that of the child shim is 9.1 KB. Note that only the parent shim is in the TCB.

Figure 4 outlines the typical events involved in proxying a privileged call. First, the child shim defines a stub implementation of the privileged APIs (e.g., `chrome.tabs.captureVisibleTab`) that, when called, forwards the call to the parent. On receiving the message, the parent shim checks with the policy and if the policy allows, the parent shim makes the call on behalf of the child. On completion of the call, the parent shim forwards the callback arguments (given by the run-

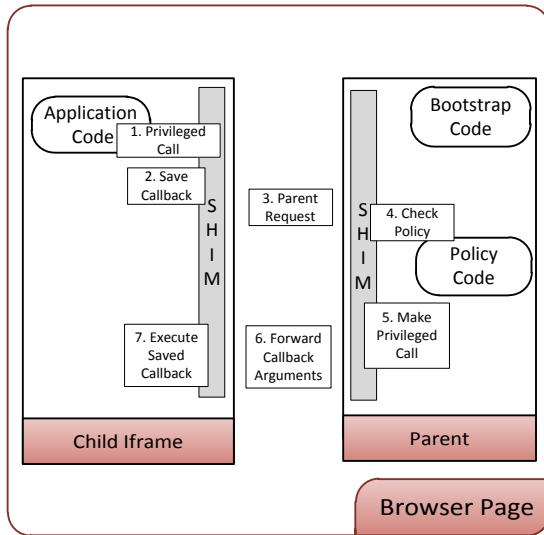


Figure 4: Typical events for proxying a privileged API call. The numbered boxes outline the events. The event boxes span the components involved. For example, event 4 involves the parent shim calling the policy code.

time) to the child shim, and the child shim executes the original callback.

Continuing with our running example, we give concrete code examples of the shims for the `chrome.tabs.captureVisibleTab` function, used to capture a screenshot. `captureVisibleTab` takes three arguments: a windowID, an options object, and a callback parameter. On successfully capturing a screenshot of the given window, the chrome runtime executes the callback with the encoded image data as the only argument. Note that the callback parameter is a first-class function; our invariants do not allow exchange of a function across the privilege boundary.

Child Shim. The child shim creates a stub implementation of the privileged API. In the unprivileged child, a privileged call would fail since the child does not have privileges to execute it. Instead, the stub function defined by the child function is called. This stub function marshals all the arguments and sends it to the parent. Listing 2 is the child shim implementation for the `captureVisibleTab` function.

No code is passed across the privilege boundary. Instead, the child saves the callback (Step 2 in Fig. 4) and forwards the rest of the argument list to the parent (Step 3). The callback is stored in a cache and a unique identifier is sent to the parent. The parent uses this identifier later.

We stress that this process is transparent to the application: the parent code ensures that the child shim is loaded before any application code starts executing. The appli-

```

tabs.captureVisibleTab =
function(windowid, options, callback){
  var id =callbackctr++;
  cached_callbacks[id] = callback;
  sendToParent({
    "type":"tabs.captureVisibleTab",
    "windowid":windowid,
    "options":options,
    "callbackid":id
  });
};

```

Listing 2: Child shim for `captureVisibleTab`

```

//m is the argument given to
// sendToParent in the child shim
if(m.type==='tabs.captureVisibleTab')
{ //fail if policy does not allow
  if(!policy.allowCall(m){ return;}
  tabs.captureVisibleTab(
    m.windowid,
    m.options,
    function(imgData){
      sendToChild({
        type:"cb_tabs.captureVisibleTab",
        id:m.callbackid,
        imgData: imgData
      });
    });
}

```

Listing 3: Parent shim for `captureVisibleTab`

cation can continue calling the privileged API as before.

Parent Shim. On receiving the message, the parent's shim first checks with the policy (Step 4 in Fig. 4 and line 5 in Listing 3) and if the policy allows it, the parent shim makes the requested privileged call.

In case of `ScreenCap`, a simple policy could disallow `captureVisibleTab` call if the request came from the image editor, and allow the call if the request came from the screenshot component. Such a policy unbundles the two components. If a network attacker compromises one of the two components in `ScreenCap`, then it only gains the ability to make request already granted to that component. As another example, the application can enforce a policy to only allow one `captureVisibleTab` call after a user clicks the 'capture' button. All future requests during that execution of the application are denied until the user clicks the 'capture' button again.

Note that the privileged call is syntactically the same as what the child would have made, except for the callback. The modified callback (lines 9-14 in Listing 3) forwards the returned image data to the child (Step 6), the original callback still executes in the child.

Child Callback The message handler on the child receives the forwarded arguments from the parent and executes the saved callback with the arguments provided by the parent. (Step 7 in Figure 4 and line 6 in Listing 4). The saved callback is then deleted from the cache (Line

```

if (
  m.type==='cb_tabs.captureVisibleTab'
){
  var cb_id = m.callbackid;
  var savedCb = cached_callbacks[cb_id
  ];
  savedCb.call(window,m.imgData);
  delete cached_callbacks[cb_id];
}

```

Listing 4: Child shim for captureVisibleTab: Part 2

7).

Persistent State. We take a different approach to data persistence APIs like `window.localStorage` and `document.cookie`. It is necessary that the data stored using these APIs is also stored in the parent since the next time a child is created, it will run in a fresh origin and the previous data will be lost. We point out that enabling persistent storage while maintaining compatibility requires some changes to code. Persistent storage APIs (like `window.localStorage`) in today's platforms are synchronous; our proxy mechanism uses `postMessage` to pass persistent data, but `postMessage` is asynchronous. To facilitate compatibility, we implement a wrapper for these synchronous API calls in the child shim code and asynchronously update the parent via `postMessage` underneath. For example, a part of the `localStorage` child shim is presented in Listing 5. The shim creates a wrapper for the `localStorage` API using an associative array (viz., `data`). On every update, the new associative array is sent to the parent. On receiving the `localStorage_save` message, the parent can save the data or discard it per policy.

We observe that in our transformation, calls to API that access persistent state become asynchronous which contrasts the synchronous API calls in the original code. To preserve the application's intended behavior, in principle, it may be necessary to re-design parts of the code that depend on the synchronous semantics of persistent storage APIs—for example, when more than one unprivileged children are sharing data via persistent state simultaneously. In our case studies so far, however, we find that the application behavior does not depend on such semantics. In future work, we plan to investigate transformation mechanisms that can provide reasonable memory consistency properties in accessing persistent local storage.

5 Case Studies

We retrofit our design onto three HTML5 applications to demonstrate that our architecture can be adopted by applications today:

- As an example of browser extensions, we retrofit our design to Awesome Screenshot, a widely used

```

setItem: function (key, value) {
  data[key] = value+'';
  saveToMainCache(data);
},

saveToMainCache: function(data){
  sendToParent({
    "type": "localStorage_save",
    "value": data
  });
},

```

Listing 5: localStorage Shim in the Child Frame

chrome extension (802,526 users) similar to Screen-Cap.

- As an example of emerging packaged HTML5 web applications, we retrofit our design to SourceKit, a full-fledged text editor available as a Chrome packaged web application. SourceKit's design is similar to editors often bundled with online word processors and web email clients. These editors typically run with the full privileges of the larger application they accompany.
- As an example of traditional HTML5 web applications, we retrofit our design to SQL Buddy, a PHP web application for database administration. Web interfaces for database administration (notably, PHPMyAdmin) are pervasive and run with the full privileges of the web application they administer.

Our goal in this evaluation is to measure (a) the reduction in TCB our architecture achieves, (b) the amount of code changes necessary to retrofit our design, and (c) performance overheads (user latency, CPU overheads and memory footprint impact) compared to platform redesign approaches. Table 1 lists our case studies and summarizes our results. First, we find that the TCB reduction achieved by our redesign ranges from 6x to 1000x. Due to the prevalence of minification, we believe LOC is not a useful metric for JavaScript code and, instead, we report the size of the code in KB. Second, we find that we require minimal changes, ranging from 0 to 13 lines, to port the case studies to our design. This is in addition to the application independent shim and bootstrap code that we added.

We also demonstrate examples of expressive policies that these applications can utilize. The focus of this paper is on mechanisms, not policies, and we do not discuss alternative policies in this work.

Finally, we also quantify the reduction in privileges we would achieve in the 50 most popular Chrome extensions with our architecture. We also find that in half the extensions studied, we can move 80% of the functions out of the TCB. This quantifies the large gap between the privileges granted by Chrome extensions today and

what is necessary. In addition, we also analyze the top 20 Chrome extensions to determine the number of components bundled in each. We find that 19 out of the top 20 extension exhibit bundling, and estimate that we can separate these between 2 to 4 components, in addition to the three components that Chrome enforces.

To facilitate further research and adoption of our techniques, we make all the application independent components of the architecture and the SQL Buddy case study available online [22]. Due to licensing restrictions, we are unable to release the other case studies publicly.

Table 1: Overview of case studies. The TCB sizes are in KB. The lines changed column only counts changes to application code, and not application independent shims and parent code.

Application	Number of users	Initial TCB (KB)	New TCB (KB)	Lines Changed
Awesome Screenshot	802,526	580	16.4	0
SourceKit	14,344	15,000	5.38	13
SQL Buddy	45,419	100	2.67	11

5.1 Awesome Screenshot

The Awesome Screenshot extension allows a user to capture a screenshot of a webpage similar to our running example [35]. A rudimentary image editor, included in the extension, allows the user to annotate and modify the captured image as he sees fit. Awesome Screenshot has over 800,000 users.³

The extension consists of three components: `background.html`, `popup.html`, and `editor.html`. A typical interaction involves the user clicking the Awesome Screenshot button, which opens `popup.html`. The user selects her desired action; `popup.html` forwards the choice to `background.html`, which captures a screenshot and sends it to the image editor (`editor.html`) for post-processing. All components communicate with each other using the `sendRequest` API call.

Privilege Separation. We redesigned Awesome Screenshot following the model laid out in Section 3 (Figure 2). Each component runs in an unprivileged temporary origin. The parent mediates access to privileged APIs, and the policy keeps this access to the minimum required by the component in question.

Code Changes. Apart from the application independent code, we required no changes to the code. The parent and child shims make the redesign seamless. We

³Due to a bug in Chrome, the current Awesome Screenshot extension uses a NPAPI binary to save big (> 2MB) images. We used the HTML5 version (which doesn't allow saving large files) for the purposes of this work. This is just a temporary limitation.

manually tested the application functionality thoroughly and did not observe any incompatibilities.

Unbundling. In the original version of Awesome Screenshot, the image editor (`editor.html`) accepts the image from `background.html` and allows the user to edit it, but runs with the full privileges of the extension—an example of bundling. Similarly, the `popup.html` only needs to forward the user's choice to `background.html` but runs with all of the extension's privileges.

In our privilege-separated implementation of Awesome Screenshot, the editor code, stored in `editor.txt` now, runs within a temporary origin. The policy only gives it access to the `sendRequest` API to send the `exit` and `ready` messages as well as receive the image data message from the background page.

TCB Reduction. The image editor in the original Awesome Screenshot extension uses UI and image manipulation libraries (more than 500KB of complex code), which run within the same origin as the extension. As a result, these libraries run with the ambient privileges to take screenshots of any page, log the user's browsing history, and access the user's data on any website. While some functions in the extension do need these privileges, the complete codebase does not need to run with these privileges.

In our privilege-separated implementation of Awesome Screenshot, the amount of code running with full privileges (TCB) decreased by a factor of 58. We found the UI and image manipulation libraries, specifically jQuery UI, used dynamic constructs like `innerHTML` and `eval`. Our design moves these potentially vulnerable constructs to an unprivileged child.

The code in the child can still request privileged function calls via the interface provided by the parent. However, this interface is thin, well defined and easily auditable. In contrast, in the non-privilege separated design, the UI and image libraries run with ambient privileges. In contrast, in the original extension *all* the code needs to be audited.

Example Policy. In addition to unbundling the image editor from the screenshot component, the parent can enforce stronger, temporal policies on the application. In particular, the parent can require that the `captureVisibleTab` function is only called once after the user clicks the `capture` button. Any subsequent calls have to be preceded by another button click. Such temporal policies are impossible to express and enforce in current permission-based systems.

5.2 SourceKit Text Editor

The SourceKit text editor is an HTML5 text editor for a user's documents stored on the Dropbox cloud service [36]. It uses open source components like the

Ajax.org cloud editor [37] and Dojo toolkit [38], in conjunction with the Dropbox REST APIs [36].

SourceKit is a powerful text editor. It includes a file-browser pane and can open multiple files at the same time. The text editor component supports themes and syntax highlighting. The application consists of 15MB of JavaScript code, all of which runs with full privileges.

Privilege Separation. In our least privilege design, the whole application runs in a single child. Redesigning SourceKit to move code to an unprivileged temporary origin was seamless because of the library shims (Section 4.1). One key change was replacing the included Dojo toolkit with its asynchronous version. The included Dojo toolkit uses synchronous XMLHttpRequest calls, which the asynchronous `postMessage` cannot proxy. The asynchronous version of Dojo is freely available on the Dojo website. We do not include this change in the number of lines modified in Table 1.

Unbundling. Functionally, SourceKit is a single Chrome application, and no bundling has occurred in its design. Popular Web sites (like GitHub [39]), use the text editor module as an online text editor [37]. In such cases, the text editor runs *bundled* with the main application, inheriting the application's privileges and increasing its attack surface. While we focus only on SourceKit for this case study, our redesign directly applies to these online text editors.

TCB Reduction. In our privilege separated SourceKit, the amount of code running with full privileges reduced from 15MB to 5KB. A large part of this reduction is due to moving the Dojo Toolkit, the syntax highlighting code and other UI libraries to an unprivileged principal. Again, we found the included libraries, specifically the Dojo Toolkit, relying on dangerous, dynamic constructs like `eval`, string arguments to `setInterval`, and `innerHTML`. In our redesign, this code executes unprivileged.

Code Change. In addition to the switch to asynchronous APIs, we also had to modify one internal function in SourceKit to use asynchronous APIs. In particular, SourceKit relied on synchronous requests to load files from the `dropbox.com` server. We modified SourceKit to use an asynchronous mechanism instead. The change was minor; only 13 lines of code were changed.

Example Policy. In the original application, all code runs with the `tabs` permission, which allows access to the user's browsing history, and permission to access `dropbox.com`. In our privilege-separated design, the policy only allows the child access to the `tabs.open` and `tabs.close` Chrome APIs for accessing `dropbox.com`. Similarly, it only forwards tab events for `dropbox.com`

URIs. Thus, after the redesign, the child has access to the user's browsing history *only* for `dropbox.com`, and not for all websites. Implementing this policy requires only two lines of code—an if condition that forwards events only for `dropbox.com` domains suffices.

SourceKit accesses Dropbox using the Dropbox OAuth APIs [36]. At first run, SourceKit opens Dropbox in a new tab, where the user can grant SourceKit the requisite OAuth access token [40]. The parent can only allow access to the tabs privileges at first run, and disable it once the child receives the OAuth token. Such temporal policies cannot be expressed by install-time permissions implemented in existing platforms.

We can also enforce stronger policies to provide a form of data separation [41]. By default, the Dropbox JS API [42] stores the OAuth access token in `localStorage`, accessible by all the code in the application. Instead, the policy code can store the OAuth token in the parent and append it to all `dropbox.com` requests. This mitigates data exfiltration attacks where the attacker can steal the OAuth token to bypass the parent's policy.⁴ Such application-specific data-separation policies cannot be expressed in present permission systems.

5.3 SQL Buddy

SQL Buddy is an open source tool to administer the MySQL database using a Web browser. Written in PHP, SQL Buddy is functionally similar to `phpMyAdmin` and supports creating, modifying, or deleting databases, tables, fields, or rows; SQL queries; and user management.

SQL Buddy uses the `MooTools` JS library to create an AJAX front-end for MySQL administration. It uses the MySQL user table for authentication and logged-in users maintain authentication via PHP session cookies.

Privilege Separation. We modified SQL Buddy to execute all its code in an unprivileged child. To ensure that no code is interpreted by the browser, we required all PHP files to return a Content-Type header of `text/plain`, as discussed in Section 3.3. Only two new files: `buddy.html` and `login.html` execute in the browser; these are initialized by the bootstrap code.

Unbundling. A typical SQL Buddy installation runs at `www.example.net/sqlbuddy`, and helps ease database management for the application at `www.example.net`. Classic operating system mechanisms can isolate SQL Buddy and the main application on the server side. But SQL Buddy runs with the full privileges of the application on the client-side. In particular, an XSS vulnerability in SQL Buddy is equivalent to an XSS vulnerability on the main application: it is not isolated from the application at the client-side. SQL Buddy inherits all the

⁴For example, to prevent malware, the parent can require that all files accessed using SourceKit have non-binary file extensions.

privileges of the application, including special client-side privileges such as access to camera, geolocation, and ambient privileges granted to the web origin such as the ability to do cross-origin XMLHttpRequests [43].

In our privilege-separated redesign, a restrictive policy on the child mitigates SQL Buddy bundling. The parent allows the child XMLHttpRequest access to only `/sqlbuddy/<filename>.php` URIs. No other privilege is available to SQL Buddy code, including `document.cookie`, `localStorage`, or XMLHttpRequest to the main application's pages. This policy isolates SQL Buddy from any other application executing on the same domain, a hitherto unavailable option.

Code Change. The key change we made to the SQL Buddy client side code was to convert the login script at the server. The original SQL Buddy system returned a new login page on a failed login. Instead, we changed it to only return an error code over XMLHttpRequest. The client-side code utilized this response to show the user the new login page, thereby preserving the application behavior. This change required modification of only 11 lines of code.

TCB Reduction. SQL Buddy utilizes the MooTools JavaScript library, which runs with the full privileges of the application site (e.g., `www.example.net`). Over 100KB of JavaScript code runs with full privileges of the `www.example.net` origin. This code uses dangerous, dynamic constructs such as `innerHTML` and `eval`. In our design, the total amount of code running in the `www.example.net` origin is 2.5KB, with the JavaScript code utilizing dynamic constructs running in an unprivileged temporary origin

Example Policy. Privilege separation reduces the ambient authority from these libraries. For example, the session cookie for `www.example.net`, is never sent to the child: all HTTP traffic requiring the cookie needs to go through the parent. Note that the cookie for the `www.example.net` principal includes both, the SQL Buddy session cookie as well as the cookie for the main `www.example.net` application. In case of successful code injection, the attacker cannot exfiltrate this cookie. Furthermore, the policy strictly limits privileged API access to those calls required by SQL Buddy. The SQL Buddy code does not have ambient authority to make privileged calls in the `www.example.net` principal. Again, implementing this policy requires two lines of JavaScript code in our architecture.

5.4 Top 50 Google Chrome extensions

Finally, we measure the opportunity available to our technique by quantifying the extent of TCB inflation and bundling in Chrome extensions. To perform this analy-

sis, we developed a syntactic static analysis engine for JavaScript using an existing JavaScript engine called Pynarcissus [44] and performed a manual review for additional confidence. We report our results on 46 out of the top 50 extensions we study.⁵ In our analysis, we (conservatively) identify all calls to privileged APIs (i.e., calls to the `chrome` object) and list them in Figure 1. We believe that our analysis is overly conservative, being syntactic, so these numbers represent only an undercount of the over-privileging in these applications.⁶

TCB Reduction. We show the distribution of the number of functions requiring any privileges as a percentage of the total number of functions. TCB inflation is pervasive in the extensions studied. In half the extensions, less than 5% of the total functions require any ambient privileges. In the current architecture the remaining 95% run with full privileges, inflating the TCB.

Bundling. We manually analyzed the 20 most popular Google Chrome extensions, and found 19 of them exhibited bundling. The most common form of bundling occurred when the options page or popup window of an extension runs with full privileges, in spite of not requiring any privileges at all. While the Google Chrome architecture does enable privilege separation between content scripts and extension code, running all code in an extension with the same privileges is unnecessary.

Another form of over-privileging occurs due to the bundling of privileges in Chrome's permission system. Google Chrome's extension system bundles multiple privileges into one coarse-grained install-time permission. For example, the `tabs` permission in Chrome extension API, required by 42 of the 46 extensions analyzed, bundles together a number of related, powerful privileges. This install-time permission includes the ability to listen to eight events related to `tabs` and `windows`, access users' browsing history, and call 20 other miscellaneous functions. Figure 5 measures the percentage of the `tabs` API actually used by extensions as a percentage of the total API granted by `tabs` for the 42 extensions analyzed. As can be seen, no extension requires the full privileges granted by the `tabs` permission, with one extension requiring 44.83% of the permitted API being the highest. More than half of the extensions require only 6.9% of the API available, which indicates over-privileging. In our design, the policy acts on fine-grained function calls and replaces coarse-grained permissions.

6 Performance Benchmarks

Our approach has two possible overheads: run-time overhead caused by the parent's mediation on privileged APIs

⁵Due to limitations of Pynarcissus, it was unable to completely parse code in 4 out of the top 50 extensions.

⁶More precise analysis can be used in the future.

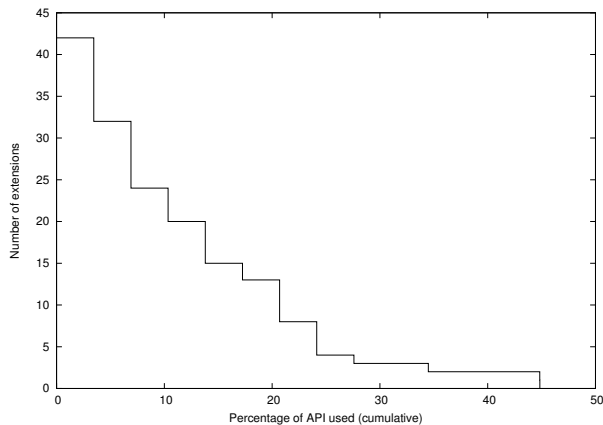


Figure 5: Frequency distribution of event listeners and API calls used by the top 42 extensions requiring the tabs permission.

and the memory consumption of the new DOM and JavaScript heap created for each iframe. We measure the impact of each below.

Performance Overhead. First, as a micro-benchmark, we measured the run-time overhead caused by the parent’s mediation on privileged APIs. We created a function that measures the total time taken to open a tab and then close it. This involves four crossings of the privilege boundary.

We performed the experiment 100 times with and without privilege separation. The median time with and without privilege-separation was 140ms and 80ms respectively. This implies an overhead of 15ms on each call crossing the sandbox.

As a macro-benchmark, we measured the amount of time required to load an image in the Awesome Screenshot image editor. Recall that the image editor receives the image data from the background page. We took a screenshot of `www.google.com` and measured the time taken for the image to load in the image editor, once the background sends it. We repeated the experiment 20 times each for the privilege separated and the original versions. The average (median) amount of time taken for the image load was 72.5ms (77.3ms) for the image load in the original Awesome Screenshot extension, and 78.5ms (80.1ms) for the image load in the privilege separated version—an overhead of 8.2% (3.6%). In our testing, we have not noticed any user-perceivable increase in latency after our redesign.

Memory Consumption. We measured the increase in memory consumption caused by creating a new temporary origin `iframe`, and found no noticeable increase in memory consumption.

On the Google Chrome platform, an alternate mechanism to get additional principals is creating a new ex-

tension. For example, Awesome Screenshot could be broken up into two extensions: a screenshot extension and an image editor extension. In addition to requiring two install decisions from the user, each additional extension runs in its own process on the Chrome platform. We measured the memory consumption of creating two extensions over a single extension and found an increase in memory consumption of 20MB. This demonstrates that our approach has no memory overhead as opposed to the 20MB overhead of creating a new extension.

7 Related Work

The concept of privilege separation was first formalized by Saltzer and Schroeder [13]. Several have used privilege separation for increased security. We discuss the most closely related works in the space.

Privilege Separation in Commodity OS Platforms. Notable examples of user-level applications utilizing privilege separation include QMail [18], OpenSSH [17] and Google Chrome [19]. Brumley and Song investigated automatic privilege separation of programmer annotated C programs and implemented data separation as well [41]. More recently, architectures like Wedge [45] identified subtleties in privilege separating binary applications and enforcing a default-deny model. Our work shows how to achieve privilege separation in emerging HTML5 applications, which are fuelling a convergence between commodity OS applications and web applications, without requiring any changes to the browser platform.

Re-architecting Browser Platforms. Several previous works on compartmentalizing web applications have suggested re-structuring the browser or the underlying execution platform altogether. Some examples include the Google Chrome extension platform [23], Escudo [12], MashupOS [46], Gazelle [47], OP [48], IPC Inspection [49], and CLAMP [50]. Our work advocates that we can achieve strong privilege separation using abstractions provided by modern browsers. This obviates the need for further changes to underlying platforms. We point out that temporary origins is similar to MashupOS’s “null-principal SERVICEINSTANCE” proposal; therefore, the alternative line of research into new browser primitives has indeed been fruitful. Our work demonstrates how we can utilize these advancements by combining deployed primitives (like temporary origins and CSP [34]) to achieve effective privilege separation, without requiring any further changes to the platform.

Carlini et al. [7] study the effectiveness of privilege separation in the Chrome extension architecture and find that in 4 (19) out of 61 cases, insufficient validation of messages exchanged over the privilege boundary allowed for full (partial) privilege escalation. In our design, we

explicitly prohibit the parent from using incoming messages in a way that can lead to code execution. Furthermore, Chrome extensions today tend to have inflated TCB in the privileged component as we show in Section 5.4. This is in contrast to our proposed design.

Mashup & Advertisement Isolation. The problem of isolating code in web applications, especially in mashups [46, 51] and malicious advertisements [52], has received much attention in research. Our work has similarities with these works in that it uses isolation primitives like `iframes`. However, one key difference is that we advocate the use of temporary origins, which are now available in most browsers, as a basis for creating arbitrary number of components.

In concurrent work, Treehouse [53] provides similar properties, but relies on isolated web workers with a virtual DOM implementation for backwards compatibility. A virtual DOM allows Treehouse to interpose on all DOM events, providing stronger security and resource isolation properties, but at a higher performance cost.

Language-based Isolation of web applications. Recent work has focused on language-based analysis of web application code, especially JavaScript, for confinement. IBEX proposed writing extensions in a high-level language (FINE) that can later be analyzed to conform to specific policies [21]. In contrast, our work does not require developers to learn new language, and thus maintains compatibility with existing code. Systems like IBEX are orthogonal to our approach and can be supported on top of our architecture; if necessary, the parent's policy component can be written in a high-level language and subject to automated analysis.

Heavyweight language-based analyses and rewriting systems have been used for isolating untrusted code, such as advertisements [28, 29, 54]. Our approach instead relies on a lighter weight mechanism based on built-in browser primitives like `iframes` and temporary origins.

8 Conclusion

Privilege separation is an important second line of defense. However, achieving privilege separation in web applications has been harder than on the commodity OS platform. We observe that the central reason for this stems in the same origin policy (SOP), which mandates use of separate origins to isolate multiple components, but creating new origins on the fly comes at a cost. As a result, web applications in practice bundle disjoint components and run them in one monolithic authority. We propose a new design that uses standardized primitives already available in modern browsers and enables partitioning web applications into an arbitrary number of temporary origins. This design contrasts with previous approaches that advocate re-designing the browser

or require adoption of new languages. We empirically show that we can apply our new architecture to widely used HTML5 applications right away; achieving drastic reduction in TCB with no more than thirteen lines of change for the applications we studied.

9 Acknowledgements

We thank Erik Kay, David Wagner, Adrienne Felt, Adrian Mettler, the anonymous reviewers, and our shepherd, William Enck for their insightful comments. This material is based upon work partially supported by the NSF under the TRUST grant CCF-0424422, by the Air Force Office of Scientific Research (AFOSR) under MURI awards FA9550-09-1-0539 and FA9550-08-1-0352 and by Intel through the ISTC for Secure Computing. The second author is supported by the Symantec Research Labs Graduate Fellowship.

References

- [1] Google Inc., "Google chrome webstore." <https://chrome.google.com/webstore/>.
- [2] HTTP Archive, "JS Transfer Size and JS Requests." <http://httparchive.org/trends.php#bytesJS&reqJS>.
- [3] Google Inc., "Chromium os." <http://www.chromium.org/chromium-os>.
- [4] "Mozilla boot2gecko." <https://wiki.mozilla.org/B2G>.
- [5] Microsoft, "Metro style app development," 2012. <http://msdn.microsoft.com/en-us/windows/apps/>.
- [6] H. Wang, A. Moshchuk, and A. Bush, "Convergence of desktop and web applications on a multi-service os," in *Proceedings of the 4th USENIX conference on Hot topics in security*, 2009.
- [7] N. Carlini, A. P. Felt, and D. Wagner, "An evaluation of the google chrome extension security architecture," in *Proceedings of the 21st USENIX Conference on Security*, 2012.
- [8] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pp. 513–528.
- [9] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett, "Vex: vetting browser extensions for security vulnerabilities," in *Proceedings of the 19th USENIX conference on Security*, 2010.

- [10] M. Dhawan and V. Ganapathy, "Analyzing information flow in javascript-based browser extensions," in *Proceedings of the Computer Security Applications Conference*, pp. 382–391, IEEE, 2009.
- [11] S. Guarnieri and B. Livshits, "Gatekeeper: mostly static enforcement of security and reliability policies for JavaScript code," in *Usenix Security*, 2009.
- [12] K. Jayaraman, W. Du, B. Rajagopalan, and S. Chapin, "Escudo: A fine-grained protection model for web browsers," in *Proceedings of the 30th International Conference on Distributed Computing Systems*, pp. 231–240, IEEE, 2010.
- [13] J. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.
- [14] "lxc linux containers." <http://lxc.sourceforge.net/>.
- [15] "Google seccomp sandbox for linux." <http://code.google.com/p/seccompsandbox/>.
- [16] N. Provos, "Improving host security with system call policies," in *Proceedings of the 12th USENIX Security Symposium*, 2003.
- [17] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation," in *Proceedings of the 12th USENIX Security Symposium*, 2003.
- [18] D. J. Bernstein, "Some thoughts on security after ten years of qmail 1.0," in *Proceedings of the 2007 ACM workshop on Computer security architecture*.
- [19] A. Barth, C. Jackson, C. Reis, and T. G. C. Team, "The security architecture of the chromium browser," 2008.
- [20] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson, "App isolation: get the security of multiple browsers with just one," in *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 227–238, 2011.
- [21] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy, "Verified security for browser extensions," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 115–130, 2011.
- [22] "Html5 privilege separation: Source code release." <http://github.com/devd/html5privsep>.
- [23] A. Barth, A. Felt, P. Saxena, and A. Boodman, "Protecting browsers from extension vulnerabilities," in *Proceedings of the 17th Network and Distributed System Security Symposium*, 2010.
- [24] A. P. Felt, K. Greenwood, and D. Wagner, "The effectiveness of application permissions," in *Proceedings of the 2nd USENIX conference on Web application development*, 2011.
- [25] K. W. Y. Au, Y. F. Zhou, Z. Huang, P. Gill, and D. Lie, "Short paper: A look at smartphone permission models," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, 2011.
- [26] A. P. Felt, "Advertising and android permissions," Nov 2011. <http://www.adrienreporterfelt.com/blog/?p=357>.
- [27] Google Inc., "Google chrome extensions: chrome.* apis." http://code.google.com/chrome/extensions/api_index.html.
- [28] S. Maffei, J. C. Mitchell, and A. Taly, "Object capabilities and isolation of untrusted web applications," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pp. 125–140.
- [29] Google Inc., "Issues: google-caja: A source-to-source translator for securing Javascript-based web content." <http://code.google.com/p/google-caja>.
- [30] M. Finifter, J. Weinberger, and A. Barth, "Preventing capability leaks in secure JavaScript subsets," in *Proc. of Network and Distributed System Security Symposium*, 2010.
- [31] G. Tan and J. Croft, "An empirical security study of the native code in the jdk," in *Proceedings of the 17th Usenix Conference on Security*, pp. 365–377, 2008.
- [32] A. Barth, "Rfc 6454: The web origin concept." <http://tools.ietf.org/html/rfc6454>.
- [33] Bugzilla@Mozilla, "Bug 341604 - (framesandbox) implement html5 sandbox attribute for iframes." https://bugzilla.mozilla.org/show_bug.cgi?id=341604.
- [34] B. Sterne and A. Barth, "Content security policy: W3c editor's draft," 2012. <https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html>.
- [35] diigo.com, "Awesome screenshot : Capture annotate share." <http://www.awesomescreenshot.com/>.

- [36] Dropbox Inc., “Dropbox developer reference.” <http://www.dropbox.com/developers/reference>.
- [37] “Ace - ajax.org cloud9 editor.” <http://ace.ajax.org/>.
- [38] The Dojo Foundation, “The dojo toolkit.” <http://dojotoolkit.org/>.
- [39] GitHub Inc., “Edit like an ace.” <https://github.com/blog/905-edit-like-an-ace>.
- [40] “Oauth.” <http://oauth.net/>.
- [41] D. Brumley and D. Song, “Privtrans: automatically partitioning programs for privilege separation,” in *Proceedings of the 13th on USENIX Conference on Security*, 2004.
- [42] P. Josling, “dropbox-js: A javascript library for the dropbox api.” <http://code.google.com/p/dropbox-js/>.
- [43] A. van Kesteren (Ed.), “Cross-origin resource sharing.” <http://www.w3.org/TR/cors/>.
- [44] “pynarcissus : The narcissus javascript interpreter ported to python.” <http://code.google.com/p/pynarcissus/>.
- [45] A. Bittau, P. Marchenko, M. Handley, and B. Karp, “Wedge: splitting applications into reduced-privilege compartments,” in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pp. 309–322, 2008.
- [46] H. J. Wang, X. Fan, J. Howell, and C. Jackson, “Protection and communication abstractions for web browsers in mashups,” *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 1–16, Oct. 2007.
- [47] H. Wang, C. Grier, A. Moshchuk, S. King, P. Choudhury, and H. Venter, “The multi-principal os construction of the gazelle web browser,” in *Proceedings of the 18th USENIX security symposium*, pp. 417–432, 2009.
- [48] C. Grier, S. Tang, and S. King, “Designing and implementing the op and op2 web browsers,” *ACM Transactions on the Web (TWEB)*, 2011.
- [49] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, “Permission re-delegation: Attacks and defenses,” in *Proceedings of the 20th USENIX conference on Security*, 2011.
- [50] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig, “Clamp: Practical prevention of large-scale data leaks,” in *Proceedings of the 30th IEEE Symposium on Security and Privacy*, pp. 154–169, 2009.
- [51] A. Barth, C. Jackson, and W. Li, “Attacks on javascript mashup communication,” in *Workshop on Web 2.0 Security and Privacy (W2SP)*, 2009.
- [52] M. T. Louw, K. T. Ganesh, and V. N. Venkatakrishnan, “Adjail: practical enforcement of confidentiality and integrity policies on web advertisements,” in *Proceedings of the 19th USENIX conference on Security*, 2010.
- [53] L. Ingram and M. Walfish, “Treehouse: Javascript sandboxes to help web developers help themselves,” in *Proceedings of the USENIX annual technical conference*, 2012.
- [54] “AdSafe : Making JavaScript Safe for Advertising.” <http://www.adsafe.org/>.

Fuzzing with Code Fragments

Christian Holler
Mozilla Corporation*
choller@mozilla.com

Kim Herzig
Saarland University
herzig@cs.uni-saarland.de

Andreas Zeller
Saarland University
zeller@cs.uni-saarland.de

Abstract

Fuzz testing is an automated technique providing random data as input to a software system in the hope to expose a vulnerability. In order to be effective, the fuzzed input must be *common enough* to pass elementary consistency checks; a JavaScript interpreter, for instance, would only accept a semantically valid program. On the other hand, the fuzzed input must be *uncommon enough* to trigger exceptional behavior, such as a crash of the interpreter. The *LangFuzz* approach resolves this conflict by using a *grammar* to randomly generate valid programs; the code fragments, however, partially stem from *programs known to have caused invalid behavior before*. *LangFuzz* is an effective tool for security testing: Applied on the Mozilla JavaScript interpreter, it discovered a total of 105 new severe vulnerabilities within three months of operation (and thus became one of the top security bug bounty collectors within this period); applied on the PHP interpreter, it discovered 18 new defects causing crashes.

1 Introduction

Software security issues are risky and expensive. In 2008, the annual CSI Computer Crime & Security survey reported an average loss of 289,000 US\$ for a single security incident. Security testing employs a mix of techniques to find vulnerabilities in software. One of these techniques is *fuzz testing*—a process that automatically generates random data input. Crashes or unexpected behavior point to potential software vulnerabilities.

In web browsers, the JavaScript interpreter is particularly prone to security issues; in Mozilla Firefox, for instance, it encompasses the majority of vulnerability fixes [13]. Hence, one could assume the JavaScript interpreter would make a rewarding target for fuzz testing. The problem, however, is that fuzzed input to a

JavaScript interpreter must follow the syntactic rules of JavaScript. Otherwise, the JavaScript interpreter will reject the input as invalid, and effectively restrict the testing to its lexical and syntactic analysis, never reaching areas like code transformation, in-time compilation, or actual execution. To address this issue, fuzzing frameworks include strategies to model the structure of the desired input data; for fuzz testing a JavaScript interpreter, this would require a built-in JavaScript grammar.

Surprisingly, the number of fuzzing frameworks that generate test inputs on grammar basis is very limited [7, 17, 22]. For JavaScript, *jsfunfuzz* [17] is amongst the most popular fuzzing tools, having discovered more than 1,000 defects in the Mozilla JavaScript engine. *jsfunfuzz* is effective because it is hardcoded to target a specific interpreter making use of specific knowledge about past and common vulnerabilities. The question is: Can we devise a *generic* fuzz testing approach that nonetheless can exploit *project-specific* knowledge?

In this paper, we introduce a framework called *LangFuzz* that allows black-box fuzz testing of engines based on a context-free grammar. *LangFuzz* is not bound against a specific test target in the sense that it takes the *grammar* as its input: given a JavaScript grammar, it will generate JavaScript programs; given a PHP grammar, it will generate PHP programs. To adapt to *specific* targets, *LangFuzz* can use its grammar to learn *code fragments* from a given *code base*. Given a suite of previously failing programs, for instance, *LangFuzz* will use and recombine fragments of the provided test suite to generate new programs—assuming that a recombination of previously problematic inputs has a higher chance to cause new problems than random input.

The combination of fuzz testing based on a language grammar and reusing project-specific issue-related code fragments makes *LangFuzz* an effective tool for security testing. Applied on the Mozilla JavaScript engine, it discovered a total of 105 new vulnerabilities within three months of operation. These bugs are serious and

*At the time of this study, Christian Holler was writing his master thesis at Saarland University. He is now employed at Mozilla.

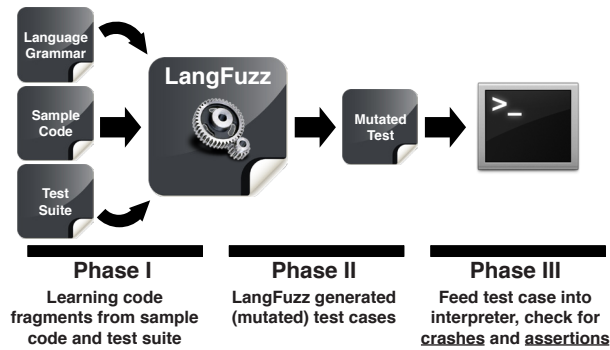


Figure 1: LangFuzz workflow. Using a language grammar, LangFuzz parses code fragments from sample code and test cases from a test suite, and mutates the test cases to incorporate these fragments. The resulting code is then passed to the interpreter for execution.

valuable, as expressed by the 50.000\$ bug bounties they raised. Nearly all the detected bugs are memory safety issues. At the same time, the approach can generically handle arbitrary grammars, as long as they are weakly typed: applied on the PHP interpreter, it discovered 18 new defects. All generated inputs are semantically correct and can be executed by the respective interpreters.

Figure 1 describes the structure of LangFuzz. The framework requires three basic input sources: a *language grammar* to be able to parse and generate code artifacts, *sample code* used to learn language fragments, and a *test suite* used for code mutation. Many test cases contain code fragments that triggered past bugs. The test suite can be used as sample code as well as mutation basis. LangFuzz then generates new test cases using code mutation and code generation strategies before passing the generated test cases to a test driver executing the test case—e.g. passing the generated code to an interpreter.

As an example of a generated test case exposing a security violation, consider Figure 2 that shows a security issue in Mozilla’s JavaScript engine. `RegExp.$1` (Line 8) is a pointer to the first grouped regular expression match. This memory area can be altered by setting a new input (Line 7). An attacker could use the pointer to arbitrarily access memory contents. In this test case, Lines 7 and 8 are newly generated by LangFuzz, whereas Lines 1–6 stem from an existing test case.

The remainder of this paper is organized as follows. Section 2 discusses the state of the art in fuzz testing and provides fundamental definitions. Section 3 presents how LangFuzz works, from code generation to actual test execution; Section 4 details the actual implementation. Section 5 discusses our evaluation setup, where we compare LangFuzz against jsfunfuzz and show that LangFuzz detects several issues which jsfunfuzz misses. Section 6 describes the application of LangFuzz on PHP.

```

1 var haystack = "foo";
2 var re_text = "foo";
3 haystack += "x";
4 re_text += "(x)";
5 var re = new RegExp(re_text);
6 re.test(haystack);
7 RegExp.input = Number();
8 print(RegExp.$1);

```

Figure 2: Test case generated by LangFuzz, crashing the JavaScript interpreter when executing Line 8. The static access of `RegExp` is deprecated but valid. Reported as Mozilla bug 610223 [1].

Section 7 discusses threats to validity, and Section 8 closes with conclusion and future work.

2 Background

2.1 Previous Work

“Fuzz testing” was introduced in 1972 by Purdom [16]. It is one of the first attempts to automatically test a parser using the grammar it is based on. We especially adapted Purdom’s idea of the “Shortest Terminal String Algorithm” for LangFuzz. In 1990, Miller et al. [10] were among the first to apply fuzz testing to real world applications. In their study, the authors used random generated program inputs to test various UNIX utilities. Since then, the technique of fuzz testing has been used in many different areas such as protocol testing [6, 18], file format testing [19, 20], or mutation of valid input [14, 20].

Most relevant for this paper are earlier studies on grammar-based fuzz testing and test generations for compiler and interpreters. In 2005, Lindig [8] generated code to specifically stress the C calling convention and check the results later. In his work, the generator also uses recursion on a small grammar combined with a fixed test generation scheme. Molnar et al. [12] presented a tool called *SmartFuzz* which uses symbolic execution to trigger integer related problems (overflows, wrong conversion, signedness problems, etc.) in x86 binaries. In 2011, Yang et al. [22] presented *CSmith*—a language-specific fuzzer operating on the C programming language grammar. *CSmith* is a pure generator-based fuzzer generating C programs for testing compilers and is based on earlier work of the same authors and on the random C program generator published by Turner [21]. In contrast to LangFuzz, *CSmith* aims to target correctness bugs instead of security bugs. Similar to our work, *CSmith* randomly uses productions from its built-in C grammar to create a program. In contrast to LangFuzz, their grammar has non-uniform probability annotations. Furthermore, they already introduce semantic rules during their

generation process by using *filter functions*, which allow or disallow certain productions depending on the context. This is reasonable when constructing a fuzzer for a specific language, but very difficult for a language independent approach as we are aiming for.

Fuzzing web browsers and their components is a promising field. The most related work in this field is the work by Ruderman and his tool *jsfunfuzz* [17]. *Jsfunfuzz* is a black-box fuzzing tool for the JavaScript engine that had a large impact when written in 2007. *Jsfunfuzz* not only searches for crashes but can also detect certain correctness errors by differential testing. Since the tool was released, it has found over 1,000 defects in the Mozilla JavaScript Engine and was quickly adopted by browser developers. *jsfunfuzz* was the first JavaScript fuzzer that was publicly available (it has since been withdrawn) and thus inspired LangFuzz. In contrast, LangFuzz does not specifically aim at a single language, although this paper uses JavaScript for evaluation and experiments. Instead, our approaches aim to be solely based on grammar and general language assumptions and to combine random input generation with code mutation.

Miller and Peterson [11] evaluated these two approaches—random test generation and modifying existing valid inputs—on PNG image formats showing that mutation testing alone can miss a large amount of code due to missing variety in the original inputs. Still, we believe that mutating code snippets is an important step that adds regression detection capabilities. Code that has been shown to detect defects helps to detect incomplete fixes when changing their context or fragments, especially when combined with a generative approach.

LangFuzz is a pure black-box approach, requiring no source code or other knowledge of the tested interpreter. As shown by Godefroid et al. [7] in 2008, a grammar-based fuzzing framework that produces JavaScript engine input (Internet Explorer 7) can increase coverage when linked to a constraint solver and coverage measurement tools. While we consider coverage to be an insufficient indicator for test quality in interpreters (just-in-time compilation and the execution itself heavily depend on the global engine state), such an extension may also prove valuable for LangFuzz.

In 2011, Zalewski [23] presented the *crossfuzz* tool that is specialized in DOM fuzzing and revealed some problems in many popular browsers. The same author has published even more fuzzers for specific purposes like *ref fuzz*, *mangleme*, *Canvas fuzzer* or *transfuzz*. They all target different functionality in browsers and have found severe vulnerabilities.

2.2 Definitions

Throughout this paper, we will make use of the following terminology and assumptions.

Defect. Within this paper, the term “defect” refers to errors in code that cause abnormal termination only (e.g. crash due to memory violation or an assertion violation). All other software defects (e.g. defect that produce false output without abnormal termination) will be disregarded, although such defects might be detected under certain circumstances. We think that this limitation is reasonable due to the fact that detecting other types of defects using fuzz-testing generally requires strong assumptions about the target software under test.

Grammar. In this paper, the term “grammar” refers to context-free grammars (Type-2 in the Chomsky hierarchy) unless stated otherwise.

Interpreter. An “interpreter” in the sense of this paper is any software system that receives a program in source code form and then executes it. This also includes just-in-time compilers which translate the source to byte code before or during runtime of the program. The main motivation to use grammar-based fuzz testing is the fact that such interpreter systems consist of lexer and parser stages that detect malformed input which causes the system to reject the input without even executing it.

3 How LangFuzz works

In fuzz testing, we can roughly distinguish between two techniques: *Generative* approaches try to create new random input, possibly using certain constraints or rules. *Mutative* approaches try to derive new testing inputs from existing data by randomly modifying it. For example, both *jsfunfuzz* [17] and *CSmith* [22] use generative approaches. LangFuzz makes use of both approaches, but mutation is the primary technique. A purely generative design would likely fail due to certain semantic rules not being respected (e.g. a variable must be defined before it is used). Introducing semantic rules to solve such problems would tie LangFuzz to certain language semantics. Mutation, however, allows us to learn and reuse existing semantic context. This way, LangFuzz stays language-independent without losing the ability to generate powerful semantic context to embed generated or mutated code fragments.

3.1 Code mutation

The mutation process consists of two phases, a *learning phase* in the beginning and the main *mutation phase*.

In the learning phase, we process a certain set of sample input files using a parser for the given language (derived from the language grammar). The parser will allow us to separate the input file into *code fragments* which are essentially examples for non-terminals in the grammar. Of course, these fragments may overlap (e.g. an expression might be contained in an `ifStatement` which is a `statement` according to the grammar). Given a large codebase, we can build up a *fragment pool* consisting of expansions for all kinds of non-terminal symbols. Once we have learned all of our input, the mutation phase starts. For mutation, a single target file is processed again using the parser. This time, we randomly pick some of the fragments we saw during parsing and replace them with other fragments of the same type. These code fragments might of course be semantically invalid or less useful without the context that surrounded them originally, but we accept this trade-off for being independent of the language semantics. In Section 3.3, we discuss one important semantic improvement performed during fragment replacement.

As our primary target is to trigger defects in the target program, it is reasonable to assume that existing test cases (especially regressions) written in the target language should be helpful for this purpose; building and maintaining such test suites is standard practice for developers of interpreters and compilers. Using the mutation process described in the previous section, we can process the whole test suite file by file, first learning fragments from it and then creating executable mutants based on the original tests.

3.2 Code generation

With our mutation approach, we can only use those code fragments as replacements that we have learned from our code base before. Intuitively, it would also be useful if we could generate fragments on our own, possibly yielding constructs that cannot or can only hardly be produced using the pure mutation approach.

Using a language grammar, it is natural to generate fragments by *random walk* over the tree of possible expansion series. But performing a random walk with uniform probabilities is not guaranteed to terminate. However, terminating the walk without completing all expansions might result in a syntactically invalid input.

Usually, this problem can be mitigated by restructuring the grammar, adding non-uniform probabilities to the edges and/or imposing additional semantic restrictions during the production, as in the *CSmith* work [22].

Restructuring or annotating the grammar with probabilities is not straightforward and requires additional work for every single language. It is even reasonable to assume that using fixed probabilities can only yield a

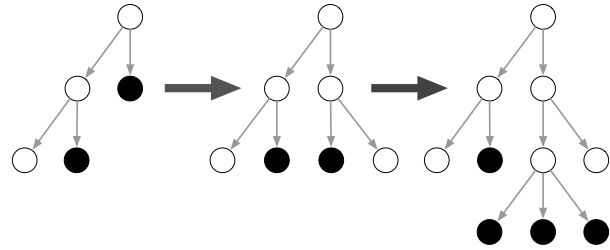


Figure 3: Example of a stepwise expansion on the syntax tree: Dark nodes are unexpanded non-terminals (can be expanded) while the other nodes have already been expanded before.

coarse approximation as the real probabilities are conditional, depending on the surrounding context. To overcome these problems, we will use an algorithm that performs the generation in a *breadth-first* manner:

1. Set current expansion e_{cur} to the start symbol S
2. Loop num iterations:
 - (a) Choose a random non-terminal n in e_{cur} :
 - i. Find the set of productions $P_n \subseteq P$ that can be applied to n .
 - ii. Pick one production p from P_n randomly and apply it to n , yielding $p(n)$.
 - iii. Replace that occurrence of n in e_{cur} by $p(n)$.

Figure 3 gives an example of such a stepwise expansion, considering the code as a syntax tree. Dark nodes are unexpanded non-terminals that can be considered for expansion while the remaining nodes have already been expanded before. This algorithm does not yield a valid expansion after num iterations. We need to replace the remaining non-terminal symbols by sequences of terminal symbols. In the learning phase of the mutation approach we are equipped with many different examples for different types of non-terminals. We randomly select any of these code fragments to replace our remaining non-terminals. In the unlikely situation that there is no example available, we can use the *minimal expansion* of the non-terminal instead. During mutation, we can use learned and generated code fragments.

3.3 Adjusting Fragments to Environment

When a fragment is replaced by a different fragment, the new fragment might not fit with respect to the semantics of the remaining program. As LangFuzz does not aim to semantically understand a specific language, we can only perform corrections based on *generic* semantic assumptions. One example with a large impact are *identifiers*.

Many programming languages use identifiers to refer to variables and functions, and some of them will throw an error if an identifier has not been declared prior to using it (e.g. in JavaScript, using an identifier that is never declared is considered to be a runtime error).

We can reduce the chances to have undeclared identifiers within the new fragment by replacing all identifiers in the fragment with identifiers that occur somewhere in the rest of the program. Note that this can be done purely at the syntactic level. LangFuzz only needs to know which non-terminal in the grammar constitutes an identifier in order to be able to statically extract known identifiers from the program and replace identifiers in the new fragment. Thus, it is still possible that identifiers are unknown at the time of executing a certain statement (e.g. because the identifier is declared afterwards), but the chances of *identifier reuse* are increased.

Some languages contain identifiers that can be used without declaring them (usually *built-in objects/globals*). The adjustment approach can be even more effective if LangFuzz is aware of these global objects in order to ignore them during the replacement process. The only way to identify such global objects within LangFuzz is to require a list of these objects as (optional) argument. Such global object lists are usually found in the specification of the respective language.

4 The LangFuzz Implementation

Based on the methods described so far, we now assemble the different parts to get a proof-of-concept fuzzer implementation that works as described in the overview diagram (Figure 1) in the introduction.

Typically, LangFuzz starts with a *learning phase* where the given sample code is parsed using the supplied language grammar, thereby learning code fragments (Section 4.1). The input of this learning phase can be either a sample code base or the test suite itself. Once the learning step is complete, LangFuzz starts to process the test suite. All tests are parsed and the results are cached for performance reasons.

Then the tool starts the actual working phase:

1. From the next test to be mutated, several fragments (determined by an adjustable parameter, typically 1–3) are randomly selected for replacement.
2. As a single fragment can be considered as multiple types (e.g. `if (true) { ... }` can be seen as an if-statement but also more generally as a statement), we randomly pick one of the possible interpretations for each of those fragments.
3. Finally, the mutated test is executed and its result is checked (Section 4.3).

4.1 Code Parsing

In the learning and mutation phase, we parse the given source code. For this purpose, LangFuzz contains a parser subsystem such that concrete parsers for different languages can be added. We decided to use the ANTLR parser generator framework [15] because it is widespread and several grammars for different languages exist in the community. The parser is first used to learn fragments from the given code base which LangFuzz then memorizes as a token stream. When producing a mutated test, the cached token stream is used to find all fragments in the test that could be replaced and to determine which code can be replaced according to the syntax—we can mutate directly on the cached token stream.

4.2 Code Generation

The code generation step uses the stepwise expansion (Section 3.2) algorithm to generate a code fragment. As this algorithm works on the language grammar, LangFuzz also includes an ANTLR parser for ANTLR grammars. However, because LangFuzz is a proof-of-concept, this subsystem only understands a subset of the ANTLR grammar syntax and certain features that are only required for parsing (e.g. implications) are not supported. It is therefore necessary to simplify the language grammar slightly before feeding it into LangFuzz. LangFuzz uses further simplifications internally to make the algorithm easier: Rules containing quantifiers ('*', '+') and optionals ('?') are de-sugared to remove these operators by introducing additional rules according to the following patterns:

$$\begin{aligned} X^* &\rightsquigarrow (R \rightarrow \epsilon | XR) && \text{(zero or more)} \\ X^+ &\rightsquigarrow (R \rightarrow X | XR) && \text{(one or more)} \\ X? &\rightsquigarrow (R \rightarrow \epsilon | X) && \text{(zero or one)} \end{aligned}$$

where X can be any complex expression. Furthermore, sub-alternatives (e.g. $R \rightarrow ((A|B)C|D)$), are split up into separate rules as well. With these simplifications done, the grammar only consists of rules for which each alternative is only a sequence of terminals and non-terminals. While we can now skip special handling of quantifiers and nested alternatives, these simplifications also introduce a new problem: The additional rules (*synthesized rules*) created for these simplifications have no counterpart in the parser grammar and hence there are no code examples available for them. In case our stepwise expansion contains one or more synthesized rules, we replace those by their minimal expansion as described in Section 3.2. All other remaining non-terminals are replaced by learned code fragments as described earlier. In our implementation, we introduced a size limitation

on these fragments to avoid placing huge code fragments into small generated code fragments.

After code generation, the fragment replacement code adjusts the new fragment to fit its new environment as described in [Section 3.3](#). For this purpose, LangFuzz searches the remaining test for available identifiers and maps the identifiers in the new fragment to existing ones. The mapping is done based on the identifier name, not its occurrence, i.e. when identifier “a” is mapped to “b”, all occurrences of “a” are replaced by “b”. Identifiers that are on the built-in identifier list (e.g. global objects) are not replaced. LangFuzz can also actively map an identifier to a built-in identifier with a certain probability.

4.3 Running Tests

In order to be able to run a mutated test, LangFuzz must be able to run the test with its proper *test harness* which contains definitions required for the test. A good example is the Mozilla test suite: The top level directory contains a file *shell.js* with definitions required for all tests. Every subdirectory may contain an additional *shell.js* with further definitions that might only be required for the tests in that directory. To run a test, the JavaScript engine must execute all shell files in the correct order, followed by the test itself. LangFuzz implements this logic in a test suite class which can be derived and adjusted easily for different test frameworks.

The simplest method to run a mutated test is to start the JavaScript engine binary with the appropriate test harness files and the mutated test. But starting the JavaScript engine is slow and starting it over and over again would cost enormous computation time. To solve this problem, LangFuzz uses a *persistent shell*: A small JavaScript program called the *driver* is started together with the test harness. This way, we reduce the number of required JavaScript engines to be started drastically. The driver runs a set of tests within one single JavaScript engine and signals completion when done. LangFuzz monitors each persistent shell and records all input to it for later reproduction. Of course the shell may not only be terminated because of a crash, but also because of timeouts or after a certain number of tests being run. The test driver is language dependent and needs to be adapted for other languages (see [Section 6](#)); such a test driver would also be required if one implemented a new fuzzer from scratch.

Although the original motivation to use persistent shells was to increase test throughput it has an important side-effect. It increased the number of defects detected. Running multiple tests within a single shell allows individual tests to influence each other. Different tests may use the same variables or functions and cause crashes that would not occur when running the individual tests

alone. In fact, most of the defects found in our experiments required multiple tests to be executed in a row to be triggered. This is especially the case for memory corruptions (e.g. garbage collector problems) that require longer runs and a more complex setup than a single test could provide.

Running multiple tests in one shell has the side effect that it increases the number of source code lines executed within each JavaScript shell. To determine which individual tests are relevant for failure reproduction we use the *delta debugging algorithm* [24] and the *delta* tool [9] to filter out irrelevant test cases. The very same algorithm also reduces the remaining number of executed source code lines. The result is a suitably small test case.

4.4 Parameters

LangFuzz contains a large amount of adjustable parameters, e.g. probabilities and amounts that drive decisions during the fuzzing process. In [Table 3](#) (see Appendix) we provide the most common/important parameters and their default values. Please note that all default values are chosen empirically. Because the evaluation of a certain parameter set is very time consuming (1–3 days per set and repeating each set hundreds of time times to eliminate the variance introduced by random generation), it was not feasible to compare all possible parameter combinations and how they influence the results. We tried to use reasonable values but cannot guarantee that these values deliver the best performance.

5 Evaluation

To evaluate how well LangFuzz discovers undetected errors in the JavaScript engines, we setup three different experimental setups. The external validation compares LangFuzz to the state of the art in JavaScript fuzzing. The internal validation compares the two fragment replacement strategies used within LangFuzz: random code generation and code mutation. Finally, we conducted a field study to check whether LangFuzz is actually up to the task to detect real defects in current state of the art JavaScript engines.

5.1 LangFuzz vs. jsfunfuzz

The state of the art fuzzer for JavaScript is the *jsfunfuzz* tool written by Ruderman [17]. The tool is widely used and has proven to be very successful in discovering defect within various JavaScript engines. *jsfunfuzz* is an active part of Mozilla’s and Google’s quality assurance and regularly used in their development.

The differences between *jsfunfuzz* and LangFuzz are significant and allow only unfair comparisons between

both tools. jsfunfuzz is highly adapted to test JavaScript engines and contains multiple optimizations. jsfunfuzz is designed to test new and previously untested JavaScript features intensively. This of course required detailed knowledge of the software project under test. Additionally, jsfunfuzz has a certain level of semantic knowledge and should be able to construct valid programs easier. However, for every new language feature, the program has to be adapted to incorporate these changes into the testing process. Also, focusing on certain semantics can exclude certain defects from being revealed at all.

In contrast, LangFuzz bases its testing strategy solely on the grammar, existing programs (e.g. test suites) and a very low amount of additional language-dependent information. In practice, this means that

- changes to the language under test do not require any program maintenance apart from possible grammar updates; and
- through the choice of test cases, LangFuzz can be set up to cover a certain application domain.

The use of existing programs like previous regression tests allows LangFuzz to profit from previously detected defects. However, LangFuzz lacks a semantic background on the language which lowers the chances to obtain sane programs and produce test cases that trigger a high amount of interaction between individual parts of the program.

Although both tools have some differences that make a fair comparison difficult, comparing both tools can unveil two very interesting questions:

Q1. *To what extent do defects detected by LangFuzz and jsfunfuzz overlap?*

By overlap, we refer to the number of defects that both tools are able to detect. A low overlap would indicate that LangFuzz is able to detect new defects that were not found and most likely will not be found by jsfunfuzz. Therefore we define the overlap as the fraction of number of defects found by both tools and the number of defects found in total. This gives us a value between zero and one. A value of one would indicate that both tools detected exactly the same defects. If both tools detected totally different defects, the overlap would be zero.

$$overlap = \frac{\text{number of defects found by both tools}}{\text{number of defects found in total}}$$

The second question to be answered by this comparison is targeted towards the effectiveness of LangFuzz.

Q2. *How does LangFuzz's detection rate compare to jsfunfuzz?*

By effectiveness, we mean how many defects each tool is able to locate in a given period of time. Even though the overlap might be large, it might be the case that either tool might detect certain defects much quicker or slower than the respective other tool. To compare the effectiveness of LangFuzz in comparison against jsfunfuzz, we define the effectiveness as:

$$effectiveness = \frac{\text{number of defects found by LangFuzz}}{\text{number of defects found by jsfunfuzz}}$$

This sets the number of defects found by LangFuzz into relation to the number of defects found by jsfunfuzz. Since both tools ran on the same time windows, the same amount of time using identical amounts of resources (e.g. CPU and RAM) we do not have to further normalize this value.

Overall, this comparison answers the question whether LangFuzz is a useful contribution to a quality assurance process, even if a fuzzer such as jsfunfuzz is already used. It is not our intention to show that either tool outperforms the other tool by any means. We believe that such comparisons are non-beneficial since both jsfunfuzz and LangFuzz operate on different assumptions and levels.

5.1.1 Testing windows

We compared jsfunfuzz and LangFuzz using Mozilla's JavaScript engine *TraceMonkey*. There were two main reasons why we decided to choose TraceMonkey as comparison base. First, Mozilla's development process and related artifacts are publicly available—data that required internal permission was kindly provided by the Mozilla development team. The second main reason was that jsfunfuzz is used in Mozilla's daily quality assurance process which ensures that jsfunfuzz is fully functional on TraceMonkey without investing any effort to setup the external tool. But using TraceMonkey as reference JavaScript engine also comes with a downside. Since jsfunfuzz is used daily within Mozilla, jsfunfuzz had already run on every revision of the engine. This fact has two major consequences: First, jsfunfuzz would most likely not find any new defects; and second, the number of potential defects that can be found by LangFuzz is significantly reduced. Consequently, it is not possible to measure effectiveness based on a single revision of TraceMonkey. Instead we make use of the fact that Mozilla maintains a list of defects found by jsfunfuzz. Using this list, we used a test strategy which is based on *time windows* in which no defect fixes were applied that are based on defect reports filed by jsfunfuzz (see [Figure 4](#)). Within these periods, both tools will have equal chances to find defects within the TraceMoney engine.

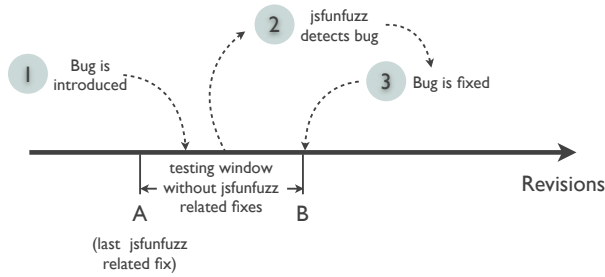


Figure 4: Example of a testing window with the live cycle of a single defect.

	start revision	end revision
W_1	46569:03f3c7efaa5e	47557:3b1c3f0e98d8
W_2	47557:3b1c3f0e98d8	48065:7ff4f93bddaa
W_3	48065:7ff4f93bddaa	48350:d7c7ba27b84e
W_4	48350:d7c7ba27b84e	49731:aaa87f0f1afe
W_5	49731:aaa87f0f1afe	51607:f3e58c264932

Table 1: The five testing windows used for the experiments. Each window is given by Mercurial revisions of the Mozilla version archive. All windows together cover approximately two months of development activity.

Within the remainder of this paper, we call these periods *testing windows*.

In detail, we applied the following test strategy for both tools:

1. Start at some base revision f_0 . Run both tools for a fixed amount of time. Defects detected can solely be used to analyze the overlap, not effectiveness.
2. Set $n = 1$ and repeat several times:
 - (a) Find the next revision f_n starting at f_{n-1} that fixes a defect found in the list of jsfunfuzz defects.
 - (b) Run both tools on $f_n - 1$ for a fixed amount of time. The defects found by both tools can be used for effectiveness measurement if and only if the defect was introduced between f_{n-1} and $f_n - 1$ (the preceding revision of f_n)¹. For overlap measurement, all defects can be used.

Figure 4 illustrates how such a testing window could look like. The window starts at revision A. At some point, a bug is introduced and shortly afterwards, the bug gets reported by jsfunfuzz. Finally, the bug is fixed in revision $B + 1$. At this point, our testing window ends and we can use revision B for experiments and count all

¹ $f_n - 1$ is exactly one revision before f_n and spans the testing window. The testing window starts at f_{n-1} and ends at $f_n - 1$ because f_n is a jsfunfuzz-induced fix.

defects that were introduced between A and B which is the testing window.

For all tests, we used the TraceMonkey development repository. Both the tested implementation and the test cases (approximately 3,000 tests) are taken from the development repository. As base revision, we chose revision 46549 (03f3c7efaa5e) which is the first revision committed in July 2010, right after Mozilla Firefox 4 Beta 1 was released at June 30, 2010. Table 1 shows the five test windows used for our experiments. The end revision of the last testing window dates to the end of August 2010, implying that we covered almost two months of development activity using these five windows. For each testing window, we ran both tools for 24 hours.²

To check whether a defect detected by either tool was introduced in the current testing window, we have to detect the lifetime of the issue. Usually, this can be achieved by using the *bisect* command provided by the Mercurial SCM. This command allows automated testing through the revision history to find the revision that introduced or fixed a certain defect. Additionally, we tried to identify the corresponding issue report to check whether jsfunfuzz found the defect in daily quality assurance routine.

5.1.2 Result of the external comparison

During the experiment, jsfunfuzz identified 23 defects, 15 of which lay within the respective testing windows. In contrast, LangFuzz found a total of 26 defects with only 8 defects in the respective testing windows. The larger proportion of defects outside the testing windows for LangFuzz is not surprising since LangFuzz, unlike jsfunfuzz, was never used on the source base before this experiment. Figure 5 illustrates the number of defects per fuzzer within the testing windows.

To address research question Q1, we identified three defects found by both fuzzers. Using the definition from Section 5.1 the overlap between LangFuzz and jsfunfuzz is 15%. While a high overlap value would indicate that both fuzzers could be replaced by each other, an overlap value of 15% is a strong indication that both fuzzers find different defects and hence supplement each other.

LangFuzz and jsfunfuzz detect different defects (overlap of 15%) and thus should be used complementary to each other.

To answer research question Q2, we computed the effectiveness of LangFuzz for the defects found within the experiment.

²For both tools we used the very same hardware. Each tool ran on 4 CPUs with the same specification. Since jsfunfuzz does not support threading, multiple instances will be used instead. LangFuzz's parameters were set to default values (see Table 3).

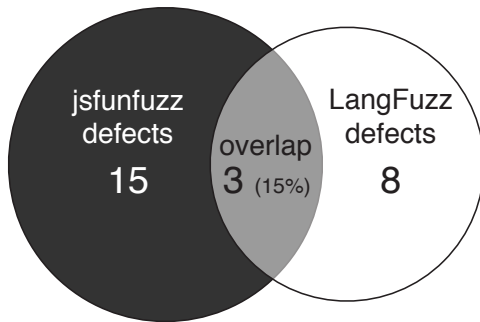


Figure 5: Number of defects found by each fuzzer within the testing windows and their overlap.

Compared to the 15 defects that were exclusively detected by jsfunfuzz LangFuzz with its eight exclusively detected defect has an effectiveness of $15 : 8 = 53\%$. In other words, LangFuzz is half as effective as jsfunfuzz.

A generic grammar-based fuzzer like LangFuzz can be 53% as effective as a language-specific fuzzer like jsfunfuzz.

For us, it was not surprising that a tried-and-proven language-specific fuzzer is more effective than our more general approach. However, effectiveness does not imply capability. The several severe issues newly discovered by LangFuzz show that the tool is capable of finding bugs not detected by jsfunfuzz.

5.1.3 Example for a defect missed by jsfunfuzz

For several defects (especially garbage collector related) we believe that jsfunfuzz was not able to trigger them due to their high complexity even after minimization. Figure 6 shows an example of code that triggered an assertion jsfunfuzz was not able to trigger. In the original bug report, Jesse Ruderman confirmed that he tweaked jsfunfuzz in response to this report: “After seeing this bug report, I tweaked jsfunfuzz to be able to trigger it.” After adaptation, jsfunfuzz eventually produced an even shorter code fragment triggering the assertion (we used the tweaked version in our experiments).

5.2 Generation vs. Mutation

The last experiment compares LangFuzz with the state of the art JavaScript engine fuzzer jsfunfuzz. The aim of this experiment is to compare the internal fragment replacement approaches of LangFuzz: code generation against code mutation. The first option learned code fragments to replace code fragments while the second option uses code generation (see Section 4.2) for replacement instead.

```

1 options('tracejit');
2 for (var j = 0; uneval({'-1':true}); ++j) {
3     (-0).toString();
4 }

```

Figure 6: Test case generated by LangFuzz causing the TraceMonkey JavaScript interpreter to violate an internal assertion when executed. Reported as Mozilla bug 626345 [2].

This experiment should clarify whether only one of the approaches accounts for most of the results (and the other only slightly improves it or is even dispensable) or if both approaches must be combined to achieve good results.

- Q3. How important is it that LangFuzz generates new code?
- Q4. How important is it that LangFuzz uses learned code when replacing code fragments?

To measure the influence of either approach, we require two independent runs of LangFuzz with different configurations but using equal limitation on resources and runtime. The first configuration forced LangFuzz to use only learned code snippets for fragment replacement (mutation configuration). The second configuration allowed code fragmentation by code generation only (generation configuration).

Intuitively, the second configuration should perform random code generation without any code mutation at all—also not using parsed test cases as fragmentation replacement basis. Such a setup would mean to fall back to a purely generative approach eliminating the basic idea behind LangFuzz. It would also lead to incomparable results. The length of purely generated programs is usually small. The larger the generated code the higher the chance to introduce errors leaving most of the generated code meaningless. Also, when mutating code, we can adjust the newly introduced fragment to the environment (see Section 3.3). Using purely generated code instead, this is not possible since there exists no consistent environment around the location where a fragment is inserted (in the syntax tree at the end of generation). Although it would be possible to track the use of identifiers during generation the result would most likely not be comparable to results derived using code mutation.

Since we compared different LangFuzz configurations only, there is no need to use the testing windows from the previous experiment described in Section 5.1. Instead, we used the two testing windows that showed most defect detection potential when comparing LangFuzz with jsfunfuzz (see Section 5.1): W_1 and W_5 . Both windows

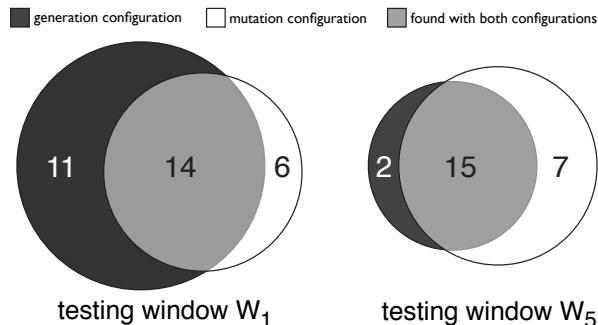


Figure 7: Defects found with/without code generation.

showed a high defect rate in the experimental setup described in Section 5.1 and spanned over 5,000 revisions giving each configuration enough defect detection potential to get comparable results.

Limiting the number of testing windows to compare the two different configurations of LangFuzz, we were able to increase the runtime for each configuration, thus minimizing the randomization impact on the experiment at the same time. Both configurations ran on both testing windows for 72 hours (complete experiment time was 12 days). For all runs we used the default parameters (see Table 3), except for the `synth.prob` parameter. This parameter can be used to force LangFuzz to use code generation only (set to 1.0) and to ignore code generation completely (set to 0.0).

Since the internal process of LangFuzz is driven by randomization, we have to keep in mind that both runs are independent and thus produce results that are hard to compare. Thus, we should use these results as indications only.

5.2.1 Result of the internal comparison

Figure 7 shows the results of the internal comparison experiment as overlapping circles. The left overlapping circle pair shows the result of the comparison using test window W_1 , the right pair the results using test window W_5 . The dark circles represent the runs using the generation configuration while the white circles represent runs using the mutation configuration. The overlap of the dark and white circles contains those defects that can be detected using both fragment replacement strategies. The numbers left and right of the overlap show the number of defects found exclusively by the corresponding configuration.

For W_1 a total of 31 defect were found. The majority of 14 defects is detected by both configurations. But 11 defects were only found when using code generation whereas six defects could only be detected using the mu-

```

('false'? length(input + ' '); delete(null?0:{}),0 ).
  watch('x', function f() { });

```

Figure 8: Test case generated by LangFuzz using code generation. The code cause the TraceMonkey JavaScript interpreter to write to a null pointer. Reported as Mozilla bug 626436 [3].

tation replacement strategy. Interestingly, this proportion is reversed in test window W_5 . Here a total of 24 defects and again the majority of 15 defects were found using both configurations. But in W_5 the number of defects found by mutation configuration exceeds the number of defects found by code generation. Combining the number of defects found by either configurations exclusively, code generation detected 13 defects that were not detected by the mutation configuration. Vice versa, code mutation detected 9 defects that were not detected during the code generation configuration run. Although the majority of 29 defects found by both configurations, these numbers and proportions show that both of LangFuzz internal fragmentation replacement approaches are crucial for LangFuzz success and should be combined. Thus, an ideal approach should be a mixed setting where both code generation and direct fragment replacement is done, both with a certain probability.

The combination of code mutation and code generation detects defects not detected by either internal approach alone. Combining both approaches makes LangFuzz successful.

5.2.2 Example of defect detected by code generation

The code shown in Figure 8 triggered an error in the parser subsystem of Mozilla TraceMonkey. This test was partly produced by code generation. The complex and unusual syntactic nesting here is unlikely to happen by only mutating regular code.

5.2.3 Example for detected incomplete fix

The bug example shown in Figure 9 caused an assertion violation in the V8 project and is a good example for both an incomplete fix detected by LangFuzz and the benefits of mutating existing regression tests: Initially, the bug had been reported and fixed as usual. Fixes had been merged into other branches and of course a new regression test based on the LangFuzz test has been added to the repository. Shortly after, LangFuzz triggered exactly the same assertion again using the newly added regression test in a mutated form. V8 developers confirmed that the initial fix was incomplete and issued another fix.

```

1 var loop_count = 5
2 function innerArrayLiteral(n) {
3   var a = new Array(n);
4   for (var i = 0; i < n; i++) {
5     a[i] = void ! delete 'object',%
6     ~ delete 4
7   }
8 }
9 function testConstructOfSizeSize(n) {
10  var str = innerArrayLiteral(n);
11 }
12 for (var i = 0; i < loop_count; i++) {
13   for (var j = 1000; j < 12000; j += 1000) {
14     testConstructOfSizeSize(j);
15   }
16 }

```

Figure 9: Test case generated by LangFuzz discovering an incomplete fix triggering an assertion failure in the Google V8 JavaScript engine. Reported as Google V8 bug 1167 [4].

5.3 Field tests

The first two experiments are targeting the external and internal comparison of LangFuzz. But so far, we did not check whether LangFuzz is actually capable of finding real world defects within software projects used within industry products. To address this issue, we conducted an experiment applying LangFuzz to three different language interpreter engines: Mozilla TraceMonkey (JavaScript), Google V8 (JavaScript), and the PHP engine. In all experiments, we used the default configuration of LangFuzz including code generation. An issue was marked as security relevant if the corresponding development team marked the issue accordingly as security issue. Thus the security relevance classification was external and done by experts.

Mozilla TraceMonkey We tested LangFuzz on the trunk branch versions of Mozilla’s TraceMonkey JavaScript engine that were part of the Firefox 4 release. At the time we ran this experiment, Firefox 4 and its corresponding TraceMonkey version were pre-release (beta version). Changes to the TraceMonkey trunk branch were regularly merged back into the main repository. Additionally, we ran LangFuzz on Mozilla’s type inference branch of TraceMonkey. At that time, this branch had alpha status and has not been part of Firefox 4 (but was eventually included in Firefox 5). Since this branch was no product branch, no security assessment was done for issues reported against it.

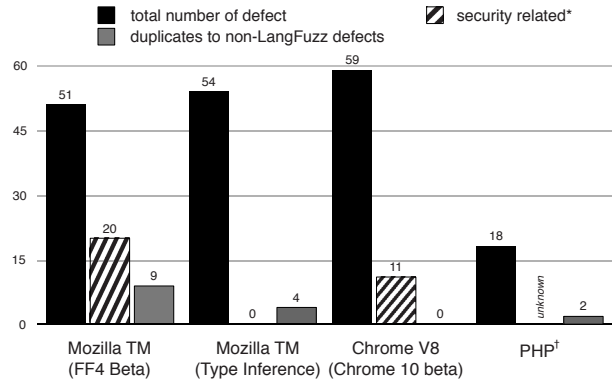


Figure 10: Real defects found on Mozilla, Google V8, and PHP. These defects were reported as customer defects and their numbers are not comparable to the defect numbers in earlier figures.*The security lock might hide the issue report from public because it might be exploitable. †Defects reported for PHP were not classified security relevant.

Google V8 Similar to the Mozilla field test, we tested LangFuzz on the Google V8 JavaScript engine contained within the development trunk branch. At the time of testing, Chrome 10—including the new V8 optimization technique “Crankshaft”—was in beta stage and fixes for this branch were regularly merged back into the Chrome 10 beta branch.

PHP To verify LangFuzz’s language independence, we performed a proof-of-concept adaptation to PHP; see Section 6 for details. The experiment was conducted on the PHP trunk branch (SVN revision 309115). The experiment lasted 14 days.

5.3.1 Can LangFuzz detect real undetected defects?

For all three JavaScript engines, LangFuzz found between 51 and 59 defects (see Figure 10). For the Mozilla TraceMonkey (FF4 Beta) branch, most left group of bars in Figure 10, 39% of the found security issues were classified as security related by the Mozilla development team. Only nine defects were classified as duplicates of bug reports not being related to our experiments. The relatively low number of duplicates within all defect sets shows that LangFuzz detects defects that slipped through the quality gate of the individual projects, showing the usefulness of LangFuzz. Although the fraction of security related defects for the Google V8 branch is lower (19%), it is still a significant number of new security related defects being found. The number of security issues within the Mozilla TraceMonkey (Type Inference) branch is reported as zero, simply because this branch was not part of any product at the time of the experi-

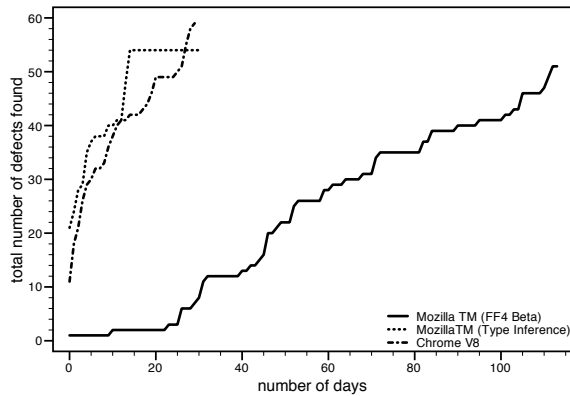


Figure 11: Cumulative sum of the total number of defects found depending on the length of the experiment. Approximate 30 days for Mozilla TM (FF4 Beta) and Google Chrome V8. We did no such analysis for PHP.

ment. This is why the Mozilla development team made no security assessment for these issue reports.

For PHP, the number of detected defects is much lower (see Figure 10). We considered the PHP experiment as a proof-of-concept adaptation and invested considerably less time into this experiment. Still, the total number of 18 defects was detected just within 14 days (1.3 defects per day). The majority of these defects concerned memory safety violations in the PHP engine. We consider these to be potential security vulnerabilities if an engine is supposed to run on untrusted input. However, the PHP development team did not classify reported issues as security relevant.

Figure 11 shows the cumulative sum of the total number of defects found depending on the length of the experiment. The length of the different plotted lines corresponds to the number of days each experiment was conducted. The result for the Mozilla TraceMonkey (FF4 Beta) branch differs in length and shape. This stems from the fact that during this experiment (which was our first experiment) we fixed multiple issues within LangFuzz which did not affect the two later applied experiments. For both Mozilla projects, LangFuzz detected constantly new defects. The curves of the two shorter experiments show a very steep gradient right after starting LangFuzz. The longer the experiment, the lower the number of defects found per time period. Although this is no surprising finding, it manifests that LangFuzz quickly find defects produced by test cases that differ greatly from other test cases used before.

The issue reports corresponding to the defects reported during field experiments can be found online using the links shown in Table 2. Each link references a search query within the corresponding issue tracker showing

exactly those issue reports filed during our field experiments. Due to the fact that some of the bug reports could be used to exploit the corresponding browser version, some issue reports are security locked requiring special permission to open the bug report. At the time of writing this paper, this affects all Google V8 issue reports.

LangFuzz detected 164 real world defects in popular JavaScript engines within four months, including 31 security related defects. On PHP, LangFuzz detected 20 defects within 14 days.

5.3.2 Economic value

The number of defects detected by LangFuzz must be interpreted with regard to the actual *value* of these defects. Many of the defects were rewarded by bug bounty awards. Within nine month of experimenting with LangFuzz, defects found by the tool obtained 18 Chromium Security Rewards and 12 Mozilla Security Bug Bounty Awards. We can only speculate on the potential damage these findings prevented; in real money, however, the above awards translated into 50,000 US\$ of bug bounties. Indeed, during this period, LangFuzz became one of the top bounty collectors for Mozilla TraceMonkey and Google V8.

6 Adaptation to PHP

Although adapting LangFuzz to a new language is kept as simple as possible, some adaptations are required. Changes related to reading/running the respective project test suite, integrating the generated parser/lexer classes, and supplying additional language-dependent information (optional) are necessary. In most cases, the required effort for these changes adaptation changes is considerably lower than the effort required to write a new language fuzzer from scratch. The following is a short description of the changes required for PHP and in general:

Integration of Parser/Lexer Classes. Given a grammar for the language, we first have to generate the Parser/Lexer Java classes using ANTLR (automatic step). For PHP, we choose the grammar supplied by the PHPParser project [5].

LangFuzz uses so called *high-level parser/lexer classes* that override all methods called when parsing non-terminals. These classes extract the non-terminals during parsing and can be automatically generated from the classes provided by ANTLR. All these classes are part of LangFuzz and get integrated into the internal language abstraction layer.

Integration of Tests. LangFuzz provides a test suite class that must be derived and adjusted depending

Experiment branch	Link
Mozilla TM (FF4 Beta)	http://tinyurl.com/lfggraph-search4
Mozilla TM (Type Inference)	http://tinyurl.com/lfggraph-search2
Google V8	http://tinyurl.com/lfggraph-search3

Table 2: Links to bug reports files during field tests. Due to security locks it might be that certain issue reports require extended permission rights and may not be listed or cannot be opened.

on the target test suite. In the case of PHP, the original test suite is quite complex because each test is made up of different sections (not a single source code file). For our proof-of-concept experiment, we only extracted the code portions from these tests, ignoring setup/teardown procedures and other surrounding instructions. The resulting code files are compatible with the standard test runner, so our runner class does not need any new implementation.

Adding Language-dependent Information (optional)

In this step, information about identifiers in the grammar and global built-in objects can be provided (e.g. taken from a public specification). In the case of PHP, the grammar in use provides a single non-terminal in the lexer for all identifiers used in the source code which we can add to our language class. Furthermore, the PHP online documentation provides a list of all built-in functions which we can add to LangFuzz through an external file.

Adapting LangFuzz to test different languages is easy: provide language grammar and integrate tests. Adding language dependent information is not required but highly recommended.

7 Threats to Validity

Our field experiments covered different JavaScript engines and a proof-of-concept adaptation to a second weak typed language (PHP). Nevertheless, we cannot generalize that LangFuzz will be able to detect defects in other interpreters for different languages. It might also be the case that there exist specific requirements or properties that must be met in order to make LangFuzz be effective.

Our direct comparison with jsfunfuzz is limited to a single implementation and limited to certain versions of this implementation. We cannot generalize the results from these experiments. Running LangFuzz and jsfunfuzz on different targets or testing windows might change comparison results.

The size and quality of test suites used by LangFuzz during learning and mutating have a major impact on its performance. Setups with less test cases or biased test suites might decrease LangFuzz’s performance.

Both jsfunfuzz and LangFuzz make extensive use of randomness. While some defects show up very quickly and frequently in all runs, others are harder to detect. Their discovery heavily depend on the time spent and the randomness involved. In our experiments, we tried to find a time limit that is large enough to minimize such effects but remains practical. Choosing different time limits might impact the experimental results.

For most experiments, we report the number of defects found. Some of the reported bugs might be duplicates. Duplicates should be eliminated to prevent bias. Although we invested a lot of efforts to identify such duplicates, we cannot ensure that we detected all of these duplicates. This might impact the number of distinct defects discovered through the experiments.

8 Conclusion

Fuzz testing is easy to apply, but needs language- and project-specific knowledge to be most effective. LangFuzz is an approach to fuzz testing that can easily be adapted to new languages (by feeding it with an appropriate grammar) and to new projects (by feeding it with an appropriate set of test cases to mutate and extend). In our evaluation, this made LangFuzz an effective tool in finding security violations, complementing project-specific tools which had been tuned towards their test subject for several years. The economic value of the bugs uncovered by LangFuzz is best illustrated by the worth of its bugs, as illustrated by the awards and bug bounties it raised. We recommend our approach for simple and effective automated testing of processors of complex input, including compilers and interpreters—especially those dealing with user-defined input.

Acknowledgments. We thank the Mozilla, Google and PHP development teams for their support. Guillaume Destuynder, Florian Gross, Clemens Hammacher, Christian Hammer, Matteo Maffei, and Eva May provided helpful feedback on earlier revisions of this paper.

References

- [1] https://bugzilla.mozilla.org/show_bug.cgi?id=610223.

- [2] https://bugzilla.mozilla.org/show_bug.cgi?id=626345.
- [3] https://bugzilla.mozilla.org/show_bug.cgi?id=626436.
- [4] <https://code.google.com/p/v8/issues/detail?id=1167>.
- [5] The `phpparser` project. Project website. <http://code.google.com/p/phpparser/>.
- [6] AITEL, D. The advantages of block-based protocol analysis for security testing. Tech. rep., 2002.
- [7] GODEFROID, P., KIEZUN, A., AND LEVIN, M. Y. Grammar-based whitebox fuzzing. *SIGPLAN Not.* 43, 6 (2008), 206–215.
- [8] LINDIG, C. Random testing of c calling conventions. *Proc. AADEBUG*. (2005), 3–12.
- [9] MCPEAK, S., AND WILKERSON, D. S. The delta tool. Project website. <http://delta.tigris.org/>.
- [10] MILLER, B. P., FREDRIKSEN, L., AND SO, B. An empirical study of the reliability of unix utilities. *Commun. ACM* 33 (December 1990), 32–44.
- [11] MILLER, C., AND PETERSON, Z. N. J. Analysis of Mutation and Generation-Based Fuzzing. Tech. rep., Independent Security Evaluators, Mar. 2007.
- [12] MOLNAR, D., LI, X. C., AND WAGNER, D. A. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th conference on USENIX security symposium* (Berkeley, CA, USA, 2009), SSYM'09, USENIX Association, pp. 67–82.
- [13] NEUHAUS, S., ZIMMERMANN, T., HOLLER, C., AND ZELLER, A. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (October 2007).
- [14] OEHLERT, P. Violating assumptions with fuzzing. *IEEE Security and Privacy* 3 (March 2005), 58–62.
- [15] PARR, T., AND QUONG, R. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience* 25, 7 (1995), 789–810.
- [16] PURDOM, P. A sentence generator for testing parsers. *BIT Numerical Mathematics* 12 (1972), 366–375. 10.1007/BF01932308.
- [17] RUDERMAN, J. Introducing jsfunfuzz. Blog Entry. <http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/>, 2007.
- [18] SHU, G., HSU, Y., AND LEE, D. Detecting communication protocol security flaws by formal fuzz testing and machine learning. In *Proceedings of the 28th IFIP WG 6.1 international conference on Formal Techniques for Networked and Distributed Systems* (Berlin, Heidelberg, 2008), FORTE '08, Springer-Verlag, pp. 299–304.
- [19] SUTTON, M., AND GREENE, A. The art of file format fuzzing. In *Blackhat USA Conference* (2005).
- [20] SUTTON, M., GREENE, A., AND AMINI, P. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.
- [21] TURNER, B. Random c program generator. Project website. <http://sites.google.com/site/brturn2/randomcprogramgenerator>, 2007.
- [22] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2011), ACM SIGPLAN, ACM.
- [23] ZALEWSKI, M. Announcing cross_fuzz. Blog Entry. <http://lcamtuf.blogspot.com/2011/01/announcing-crossfuzz-potential-0-day-in.html>, 2011.
- [24] ZELLER, A., AND HILDEBRANDT, R. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* (2002), 183–200.

Appendix

Parameter	Default Value
<code>synth.prob</code> – Probability to generate a required fragment instead of using a known one.	0.5
<code>synth.maxsteps</code> – The maximal number of steps to make during the stepwise expansion. The actual amount is 3 + a randomly chosen number between 1 and this value.	5
<code>fragment.max.replace</code> – The maximal number of fragments that are replaced during test mutation. The actual amount is a randomly chosen number between 1 and this value.	2
<code>identifier.whitelist.active.prob</code> – The probability to actively introduce a built-in identifier during fragment rewriting (i.e. a normal identifier in the fragment is replaced by a built-in identifier).	0.1

Table 3: Common parameters in LangFuzz and their default values. See Section 4.4 on how these default values were chosen.

kGuard: Lightweight Kernel Protection against Return-to-user Attacks

Vasileios P. Kemerlis

Georgios Portokalidis

Angelos D. Keromytis

Network Security Lab

Department of Computer Science

Columbia University, New York, NY, USA

{vpk, porto, angelos}@cs.columbia.edu

Abstract

Return-to-user (ret2usr) attacks exploit the operating system kernel, enabling local users to hijack privileged execution paths and execute arbitrary code with elevated privileges. Current defenses have proven to be inadequate, as they have been repeatedly circumvented, incur considerable overhead, or rely on extended hypervisors and special hardware features. We present kGuard, a compiler plugin that augments the kernel with compact inline guards, which prevent ret2usr with low performance and space overhead. kGuard can be used with any operating system that features a weak separation between kernel and user space, requires no modifications to the OS, and is applicable to both 32- and 64-bit architectures. Our evaluation demonstrates that Linux kernels compiled with kGuard become impervious to a variety of control-flow hijacking exploits. kGuard exhibits lower overhead than previous work, imposing on average an overhead of 11.4% on system call and I/O latency on x86 OSs, and 10.3% on x86-64. The size of a kGuard-protected kernel grows between 3.5% and 5.6%, due to the inserted checks, while the impact on real-life applications is minimal ($\leq 1\%$).

1 Introduction

The operating system (OS) kernel is becoming an increasingly attractive target for attackers [30, 60, 61, 64]. Due to the weak separation between user and kernel space, direct transitions from more to less privileged protection domains (*e.g.*, kernel to user space) are permissible, even though the reverse is not. As a result, bugs like NULL pointer dereferences that would otherwise cause only system instability, become serious vulnerabilities that facilitate privilege escalation attacks [64]. When successful, these attacks enable local users to execute arbitrary code with kernel privileges, by redirecting the control flow of the kernel to a user process.

Such *return-to-user* (ret2usr) attacks have affected all major OSs, including Windows [60], Linux [16, 18], and FreeBSD [19, 59, 61], while they are not limited to x86 systems [23], but have also targeted the ARM [30], DEC [31], and PowerPC [25] architectures.

There are numerous reasons to why attacks against the kernel are becoming more common. First and foremost, processes running with administrative privileges have become harder to exploit due to the various defense mechanisms adopted by modern OSs [34, 52]. Second, NULL pointer dereference errors had not received significant attention, until recently, exactly because they were thought impractical and too difficult to exploit. In fact, 2009 has been proclaimed, by some security researchers, as “*the year of the kernel NULL pointer dereference flaw*” [15]. Third, exploiting kernel bugs, besides earning attackers administrative privileges, enables them to mask their presence on compromised systems [6].

Previous approaches to the problem are either impractical for deployment in certain environments or can be easily circumvented. The most popular approach has been to disallow user processes to memory-map the lower part of their address space (*i.e.*, the one including page zero). Unfortunately, this scheme has been circumvented by various means [21, 66] and is not backwards compatible [35]. The PaX [52] patch for x86 and x86-64 Linux kernels does not exhibit the same shortcomings, but greatly increases system call and I/O latency, especially on 64-bit systems.

Recent advances in virtualization have fostered a wave of research on extending virtual machine monitors (VMMs) to enforce the integrity of the virtualized guest kernels. SecVisor [62] and NICKLE [56] are two hypervisor-based systems that can prevent ret2usr attacks by leveraging memory virtualization and VMM introspection. However, virtualization is not always practical. Consider smartphone devices that use stripped-down versions of Linux and Windows, which are also vulnerable to such attacks [30]. Running a complex VMM, like

SecVisor, on current smartphones is not realistic due to their limited resources (*i.e.*, CPU and battery life). On PCs, running the whole OS over a VM incurs performance penalties and management costs, while increasing the complexity and size of a VMM can introduce new bugs and vulnerabilities [44, 58, 71]. To address the latter, we have seen proposals for smaller and less error-prone hypervisors [65], as well as hypervisor protection solutions [4, 67]. The first exclude mechanisms such as SecVisor, while the second add further complexity and overhead, and lead to a “turtles all the way down” problem,¹ by introducing yet another software layer to protect the layers above it. Addressing the problem in hardware would be the most efficient solution, but even though Intel has recently announced a new CPU feature, named SMEP [37], to thwart such attacks, hardware extensions are oftentimes adopted slowly by OSs. Note that other vendors have not publicly announced similar extensions.

We present a lightweight solution to the problem. kGuard is a compiler plugin that augments kernel code with control-flow assertions (CFAs), which ensure that privileged execution remains within its valid boundaries and does not cross to user space. This is achieved by identifying all indirect control transfers during compilation, and injecting compact dynamic checks to attest that the kernel remains confined. When a violation is detected, the system is halted by default, while a custom fault handler can also be specified. kGuard is able to protect against attacks that overwrite a branch target to directly transfer control to user space [23], while it also handles more elaborate, two-step attacks that overwrite data pointers to point to user-controlled memory, and hence hijack execution via tampered data structures [20].

Finally, we introduce two novel code diversification techniques to protect against attacks that employ bypass trampolines to avoid detection by kGuard. A trampoline is essentially an indirect branch instruction contained within the kernel. If an attacker manages to obtain the address of such an instruction and can also control its operand, he can use it to bypass our checks. Our techniques randomize the locations of the CFA-indirect branch pairs, both during compilation and at runtime, significantly reducing the attackers’ chances of guessing their location. The main contributions of this paper can be summarized in the following:

- We present the design and implementation of kGuard, a compiler plugin that protects the kernel from ret2usr attacks by injecting fine-grained inline guards during compilation. Our approach does not require modifications to the kernel or additional software, such as a VMM. It is also architecture in-

dependent by design, allowing us to compile OSs for different target architectures and requires little modifications for supporting new OSs.

- We introduce two code diversification techniques to randomize the location of indirect branches, and their associated checks, for thwarting elaborate exploits that employ bypass trampolines.
- We implement kGuard as a GCC extension, which is *freely* available. Its maintenance cost is low and can successfully compile functional x86/x86-64 Linux and FreeBSD kernels. More importantly, it can be easily combined with other compiler-based protection mechanisms and tools.
- We assess the effectiveness of kGuard using real privilege escalation attacks against 32- and 64-bit Linux kernels. In all cases, kGuard was able to successfully detect and prevent the respective exploit.
- We evaluate the performance of kGuard using a set of macro- and micro-benchmarks. Our technique incurs minimal runtime overhead on both x86 and x86-64 architectures. Particularly, we show negligible impact on real-life applications, and an average overhead of 11.4% on system call and I/O latency on x86 Linux, and 10.3% on x86-64. The space overhead of kGuard due to the instrumentation is between 3.5% – 5.6%, while build time increases by 0.05% to 0.3%.

kGuard is to some extent related to previous research on control-flow integrity (CFI) [2]. Similar to CFI, we rely on inline checks injected before every unsafe control-flow transfer. Nevertheless, CFI depends on a precomputed control-flow graph for determining the permissible targets of every indirect branch, and uses binary rewriting to inject labels and checks in binaries.

CFI is not effective against ret2usr attacks. Its integrity is only guaranteed if the attacker cannot overwrite the code of the protected binary or execute data. During a ret2usr attack, the adversary completely controls user space memory, both in terms of contents and rights, and hence, can subvert CFI by prepending his shellcode with the respective label. Additionally, CFI induces considerable performance overhead, thereby making it difficult to adopt. Ongoing work tries to overcome the limitations of the technique [72]. kGuard can be viewed as a lightweight variant of CFI and Program Shepherding [43] that is more suitable and efficient in protecting kernel code from ret2usr threats.

The rest of this paper is organized as follows. In Section 2, we look at how ret2usr attacks work and why the current protection schemes are insufficient. Section 3 presents kGuard. We discuss the implementation of the

¹http://en.wikipedia.org/wiki/Turtles_all_the_way_down

kGuard GCC plugin in Section 4, and evaluate its effectiveness and performance in Section 5. Section 6 discusses possible extensions. Related work is in Section 7 and conclusions in Section 8.

2 Overview of ret2usr Attacks

2.1 Why Do They Work?

Commodity OSs offer process isolation through private, hardware-enforced virtual address spaces. However, as they strive to squeeze more performance out of the hardware, they adopt a “shared” process/kernel model for minimizing the overhead of operations that cross protection domains, like system calls. Specifically, Unix-like OSs divide virtual memory into *user* and *kernel* space. The former hosts user processes, while the latter holds the kernel, device drivers, and kernel extensions (interested readers are referred to Figure 5, in the appendix, for more information regarding the virtual memory layout of kernel and user space in Linux).

In most CPU architectures, the segregation of the two spaces is assisted and enforced by two hardware features. The first is known as *protection rings* or CPU modes, and the second is the memory management unit (MMU). The x86/x86-64 CPU architecture supports four protection rings, with the kernel running in the most privileged one (ring 0) and user applications in the least privileged one (ring 3).² Similarly, the PowerPC platform supports two CPU modes, SPARC and MIPS three, and ARM seven. All these architectures also feature a MMU, which implements virtual memory and ensures that memory assigned to a ring is not accessible by less privileged ones.

Since code running in user space cannot directly access or jump into the kernel, specific hardware facilities (*i.e.*, interrupts) or special instructions (*e.g.*, `SYS{ENTER, CALL}` and `SYS{EXIT, RET}` in x86/x86-64) are provided for crossing the user/kernel boundary. Nevertheless, while executing kernel code, complete and unrestricted access to *all* memory and system objects is available. For example, when servicing a system call for a process, the kernel has to directly access user memory for storing the results of the call. Hence, when kernel code is abused, it can jump into user space and execute arbitrary code with kernel privileges. Note that although some OSs have completely separated kernel and user spaces, such as the 32-bit XNU and Linux running on UltraSPARC, most popular platforms use a shared layout. In fact, on MIPS the shared address space is mandated by the hardware.

²Some x86/x86-64 CPUs have more than four rings. Hardware-assisted virtualization is colloquially known as ring -1, while System Management Mode (SMM) is supposedly at ring -2.

As a consequence, software bugs that are only a source of instability in user space, like NULL pointer dereferences, can have more dire effects when located in the kernel. Spengler [64] demonstrated such an attack by exploiting a NULL pointer dereference bug, triggered by the invocation a system call with specially crafted parameters. Earlier, it was generally thought that such flaws could only be used to perform denial-of-service (DoS) attacks [29], but Spengler’s exploit showed that mapping code segments with different privileges inside the same scope can be exploited to execute arbitrary user code with kernel privileges. Note that SELinux [47], the hardened version of the Linux kernel, is also vulnerable to this attack.

2.2 How Do They Work?

ret2usr attacks are manifested by overwriting kernel-level control data (*e.g.*, return addresses, jump tables, function pointers) with user space addresses. In early versions of such exploits, this was accomplished by invoking a system call with carefully crafted arguments to nullify a function pointer. When the null function pointer is eventually dereferenced, control is transferred to address zero that resides in user space. Commonly, that address is not used by processes and is unmapped.³ However, if the attacker has local access to the system, he can build a program with arbitrary data or code mapped at address zero (or any other address in his program for that matter). Notice that since the attacker controls the program, its memory pages can be mapped both writable and executable (*i.e.*, `W^X` anti-measures do not apply).

```
736 sock = file->private_data;
737 flags = !(file->f_flags & O_NONBLOCK) ? \
738         0 : MSG_DONTWAIT;
739 if (more)
740     flags |= MSG_MORE;
741 /*[!] NULL pointer dereference (sendpage) [!]/
742 return sock->ops->sendpage(sock, page, offset,
743                             size, flags);
```

Snippet 1: NULL *function* pointer in Linux (*net/socket.c*)

Snippet 1 presents a straightforward NULL function pointer vulnerability [17] that affected all Linux kernel versions released between May 2001 and August 2009 (2.4.4/2.6.0 – 2.4.37/2.6.30.4). In this exploit, if the `sendfile` system call is invoked with a socket descriptor belonging to a vulnerable protocol family, the value of the `sendpage` pointer in line 742 is set to NULL. This results in an indirect function call to address zero, which can be exploited by attackers to execute arbitrary code with kernel privileges. A more detailed analysis of this attack is presented in Appendix A.

³In Linux accessing an unmapped page, when running in kernel mode, results into a *kernel oops* and subsequently causes the OS to kill the offending process. Other OSs fail-stop with a kernel panic.

```
1333 /*[!] NULL pointer dereference (ops) [!]*/
1334 ibuf->ops->get(ipipe, ibuf);
1335 obuf = opipe->bufs + nbuf;
1336 *obuf = *ibuf;
```

Snippet 2: NULL *data* pointer in Linux (*fs/splice.c*)

Snippet 2 shows the Linux kernel bug exploited by Spengler [64], which is more elaborate. The `ops` field in line 1334, which is a data pointer of type `struct pipe_buf_operations`, becomes NULL after the invocation of the `tee` system call. Upon dereferencing `ops`, the effective address of a function is read via `get`, which is mapped to the seventh double word (assuming an x86 32-bit architecture) after the address pointed by `ops` (*i.e.*, due to the definition of the structure). Hence, the kernel reads the branch target from `0x0000001C`, which is controlled by the user. This enables an attacker to redirect the kernel to an arbitrary address.

NULL pointer dereferences are not the only attack vector for `ret2usr` exploits. Attackers can partially corrupt, or completely overwrite with user space addresses, kernel-level control data, after exploiting memory safety bugs. Examples of common targets include, return addresses, global dispatch tables, and function pointers stored in kernel stack and heap. In addition, other vulnerabilities allow attackers to corrupt arbitrary kernel memory, and consequently any function or data pointer, due to the improper sanitization of user arguments [22, 23]. Use-after-free vulnerabilities due to race conditions in FreeBSD 6.x/7.x and Linux kernels before 2.6.32-rc6 have also been used for the same purpose [19, 20]. These flaws are more complex and require multiple simultaneous kernel entrances to trigger the bug. Once they succeed, the attacker can corrupt a pointer to a critical kernel data structure that grants him complete control over its contents by mapping a tampered data structure at user space memory. If the structure contains a function pointer, the attacker can achieve user code execution.

The end effect of all these attacks is that the kernel is hijacked and control is redirected to user space code. Throughout the rest of this paper, we will refer to this type of exploitation as *return-to-user* (`ret2usr`), since it resembles the older *return-to-libc* [27] technique that redirected control to existing code in the C library. Interestingly, `ret2usr` attacks are yet another incarnation of the confused deputy problem [39], where a user “cheats” the kernel (deputy) to misuse its authority and execute arbitrary, non-kernel code with elevated privileges. Finally, while most of the attacks discussed here target Linux, similar flaws have been reported against FreeBSD, OpenBSD, and Windows [19, 26, 59–61].

2.3 Limitations of Current Defenses

Restricting `mmap` The mitigation strategy adopted by most Linux and BSD systems is to restrict the ability to map the first pages of the address space to users with administrative privileges only. In Linux and FreeBSD, this is achieved by modifying the `mmap` system call to apply the respective restrictions, as well as preventing binaries from requesting page zero mappings. OpenBSD completely removed the ability to map page zero, while NetBSD has not adopted any protective measures yet.

Unfortunately, this approach has *several* limitations. First and foremost, it does not solve the actual problem, which is the weak separation of spaces. Disallowing access to lower logical addresses is merely a protection scheme against exploits that rely on NULL pointer bugs. If an attacker bypasses the restriction imposed by `mmap`, he can still orchestrate a `ret2usr` attack. Second, it does not protect against exploits where control is redirected to memory pages above the forbidden `mmap` region (*e.g.*, by nullifying one or two bytes of a pointer, or overwriting a branch target with an arbitrary value). Third, it breaks compatibility with applications that rely on having access to low logical addresses, such as QEMU [5], Wine [70], and DOSEMU [28]. Similar problems have been reported for the FreeBSD distribution [35].

In fact, shortly after these protection mechanisms were set in place, many techniques were developed for circumventing them. The first technique for bypassing `mmap` restrictions used the `brk` system call for changing the location of the program break (marked as `brk_offset` in Figure 5), which indicates where the heap segment starts. By setting the break to a low logical address, it was possible to dynamically allocate memory chunks inside page zero. Another technique used the `mmap` system call to map pages starting from an address above the forbidden region and extend the allocated region downwards, by supplying the `MAP_GROWSDOWN` parameter to the call. A more elaborate mechanism utilized the different execution domains supported by Linux, which can be set with the `personality` system call, for executing binaries compiled for different OSs. Specifically, an attacker could set the personality of a binary to `SRV4`, thus mapping page zero, since `SRV4` utilizes the lower pages of the address space [66]. Finally, the combination of a NULL pointer with an integer overflow has also been demonstrated, enabling attackers to completely bypass the memory mapping restrictions [20, 21]. Despite the fact that all the previous techniques were fixed shortly after they were discovered, it is possible that other approaches can (and probably will be) developed by persistent attackers, since the root cause of this new manifestation of control hijacking attacks is the weak separation of spaces.

Hardening with PaX UDEREF [53] and KERNEXEC are two patches included in PaX [52] for hardening the Linux kernel. In particular, they provide protection against dereferencing, or branching to, user space memory. In x86, PaX relies on memory segmentation. It maps kernel space into an expand-down segment that returns a memory fault whenever privileged code tries to dereference pointers to other segments.⁴ In x86-64, where segmentation is not available, PaX resorts in temporarily remapping user space memory into a different area, using non-executable rights, when execution enters the kernel, and restoring it when it exits.

PaX has limitations. First, it requires kernel patching and is platform and architecture specific (*i.e.*, x86/x86-64 Linux only). On the other hand, ret2usr attacks not only have been demonstrated on many architectures, such as ARM [30], DEC Alpha [31], and PowerPC [25], but also on different OSs, like the BSDs [19, 26, 59, 61]. Second, as we experimentally confirmed, PaX incurs non-negligible performance overhead (see Section 5). In x86, it achieves strong isolation using the segmentation unit, but the kernel still needs to interact with user-level processes. Hence, PaX modifies the stub that executes during kernel entry for setting the respective segments, and also patches code that copies data to/from user space, so as to temporarily flatten the privileged segment for the duration of the copy. Evidently, this approach increases system call latency. In x86-64, remapping user space requires page table manipulation, which results in a TLB flush and exacerbates the problem [41].

3 Protection with kGuard

3.1 Overview

We propose a defensive mechanism that builds upon *inline monitoring* and *code diversification*. kGuard is a cross-platform compiler plugin that enforces address space segregation, without relying on special hardware features [37, 53] or custom hypervisors [56, 62]. It protects the kernel from ret2usr attacks with low-overhead, by augmenting exploitable control transfers with dynamic *control-flow assertions* (CFAs) that, at runtime, prevent the unconstrained transition of privileged execution paths to user space. The injected CFAs perform a small runtime check before indirect branches to verify that the target address is always in kernel space. If the assertion is true, execution continues normally, while if it fails because of a violation, execution is transferred to a handler that was inserted during compilation. The default handler appends a warning message to the kernel log and halts the system. We choose to coerce assertion

⁴In x86, UDEREF restricts only the SS, DS, and ES segments. CS is taken care by the accompanying KERNEXEC patch.

failures into a kernel fail-stop to prevent unsafe conditions, such as leaving the OS into an inconsistent state (*e.g.*, by aborting an in-flight kernel thread that might hold locks or other resources). In Section 6, we discuss how we can implement custom handlers for facilitating forensic analysis, error virtualization [63], selective confinement, and protection against persistent attacks.

After compiling a kernel with kGuard, its execution is limited to the privileged address space segment (*e.g.*, addresses higher than 0xC0000000 in x86 Linux and BSD). kGuard does not rely on any mapping restriction, so the previously restricted addresses can be dispensed to the process, lifting the compatibility issues with various applications [5, 28, 35, 70]. Furthermore, the checks cannot be bypassed using `mmap` hacks, like the ones described in the previous section, nor can they be circumvented by elaborate exploits that manage to jump to user space by avoiding the forbidden low memory addresses. More importantly, the kernel can still read and write user memory, so its functionality remains unaffected.

3.2 Threat Model

In this work, we ascertain that an adversary is able to completely overwrite, partially corrupt (*e.g.*, zero out only certain bytes), or nullify control data that are stored inside the address space of the kernel. Notice that overwriting certain data with arbitrary values, differs significantly from overwriting *arbitrary kernel memory with arbitrary values*. kGuard does not deal with such an adversary. In addition, we assume that the attacker can tamper with whole data structures (*e.g.*, by mangling data pointers), which in turn may contain control data.

Our technique is straightforward and guarantees that *kernel/user space boundary violations are prevented*. However, it is not a panacea that protects the kernel from all types control-flow hijacking attacks. For instance, kGuard does not address direct code-injection inside kernel space, nor it thwarts code-reuse attacks that utilize return-oriented/jump-oriented programming (ROP/JOP) [7, 40]. Nevertheless, note the following. First and foremost, our approach is orthogonal to many solutions that do protect against such threats [4, 14, 42, 45, 53, 62]. For instance, canaries injected by the compiler [34] can be used against ret2usr attacks performed via kernel stack-smashing. Second, the unique nature of address space sharing casts many protection schemes, for the aforementioned problems, ineffective. As an example, consider again the case of ROP/JOP in the kernel setting. No matter what anti-ROP techniques have been utilized [45, 51], the attacker can still execute arbitrary code, as long as there is no strict process/kernel separation, by mapping his code to user space and transferring control to it (after hijacking a privileged execution path).

Finally, in order to protect kGuard from being subverted, we utilize a lightweight diversification technique for the kernel’s text, which can also mitigate kernel-level attacks that use code “gadgets” in a ROP/JOP fashion (see Section 3.5). Overall, the aim of kGuard is not to provide strict control-flow integrity for the kernel, but rather to render a realistic threat ineffective.

3.3 Preventing ret2usr Attacks with CFAs

In the remainder of this section, we discuss the fundamental aspects of kGuard using examples based on x86-based Linux systems. However, kGuard is by no means restricted to 32-bit systems and Linux. It can be used to compile any kernel that suffers from ret2usr attacks for both 32- and 64-bit CPUs. kGuard “guards” indirect control transfers from exploitation. In the x86 instruction set architecture (ISA), such control transfers are performed using the `call` and `jmp` instructions with a register or memory operand, and the `ret` instruction, which takes an implicit memory operand from the stack (*i.e.*, the saved return address). kGuard injects CFAs in both cases to check that the branch target, specified by the respective operand, is inside kernel space.

```
81 fb 00 00 00 c0 ; cmp $0xc0000000,%ebx
73 05             ; jae call_1b1
bb 00 00 00 00   ; mov $0xc05af8f1,%ebx
ff d3           ; call_1b1: call *%ebx
```

Snippet 3: CFA_R guard applied on an indirect call in x86 Linux (*drivers/cpufreq/cpufreq.c*)

```
register void *target_address;
...
if (target_address < 0xC0000000)
    target_address = &<violation handler>;
call *target_address;
```

Snippet 4: CFA_R guard in C-like code (x86)

We use two different CFA guards, namely CFA_R and CFA_M , depending on whether the control transfer that we want to confine uses a register or memory operand. Snippet 3 shows an example of a CFA_R guard. The code is from the `show()` routine of the `cpufreq` driver. kGuard instruments the indirect `call` (`call *%ebx`) with 3 additional instructions. First, the `cmp` instruction compares the `ebx` register with the lower kernel address `0xC0000000`.⁵ If the assertion is true, the control transfer is *authorized* by jumping to the `call` instruction. Otherwise, the `mov` instruction loads the address of the violation handler (`0xc05af8f1`; `panic()`) into the branch register and proceeds to execute `call`, which will result into invoking the violation handler. In C-like code, this is equivalent to injecting the statements shown in Snippet 4.

⁵The same is true for x86 FreeBSD/NetBSD, whereas for x86-64 the check should be with address `0xFFFFFFFF80000000`. OpenBSD maps the kernel to the upper 512MB of the virtual address space, and hence, its base address in x86 CPUs is `0xD0000000`.

```
57             ; push %edi
8d 7b 50       ; lea 0x50(%ebx),%edi
81 ff 00 00 00 c0 ; cmp $0xc0000000,%edi
73 06         ; jae kmem_1b1
5f           ; pop %edi
e8 43 d6 2d b8 ; call 0xc05af8f1
5f           ; kmem_1b1: pop %edi
81 7b 50 00 00 00 c0 ; cmpl $0xc0000000,0x50(%ebx)
73 05         ; jae call_1b1
c7 43 50 f1 f8 5a c0 ; movl $0xc05af8f1,0x50(%ebx)
ff 53 50     ; call_1b1: call *0x50(%ebx)
```

Snippet 5: CFA_M guard applied on an indirect call in x86 Linux (*net/socket.c*)

```
register void *target_address_ptr;
...
target_address_ptr = &target_addr;
if (target_address_ptr < 0xC0000000)
    call <violation handler>;
if (target_address < 0xC0000000)
    target_address = &<violation handler>;
call *target_address;
```

Snippet 6: CFA_M guard in C-like code (x86)

Similarly, CFA_M guards confine indirect branches that use memory operands. Snippet 5 illustrates how kGuard instruments the faulty control transfer of `sock_sendpage()` (the original code is shown in Snippet 1). The indirect `call` (`call 0x50(%ebx)`; Figure 5) is prepended by a sequence of 10 instructions that perform two distinct assertions. CFA_M not only asserts that the branch target is within the kernel address space, but also *ensures that the memory address where the branch target is loaded from is also in kernel space*. The latter is necessary for protecting against cases where the attacker has managed to hijack a data pointer to a structure that contains function pointers (see Snippet 2 in Section 2.2). Snippet 6 illustrates how this can be represented in C-like code. In order to perform this dual check, we first need to spill one of the registers in use, unless the basic block where the CFA is injected has spare registers, so that we can use it as a temporary variable (*i.e.*, `edi` in our example). The address of the memory location that stores the branch target (`ebx + 0x50 = 0xfa7c8538`; Figure 5), is dynamically resolved via an arithmetic expression entailing registers and constant offsets. We load its effective address into `edi` (`lea 0x50(%ebx),%edi`), and proceed to verify that it points in kernel space. If a violation is detected, the spilled register is restored and control is transferred to the runtime violation handler (`call 0xc05af8f1`). Otherwise, we proceed with restoring the spilled register and confine the branch target similarly to the CFA_R case.

```
81 7b 50 00 00 00 c0 ; cmpl $0xc0000000,0x50(%ebx)
73 05             ; jae call_1b1
c7 43 50 f1 f8 5a c0 ; movl $0xc05af8f1,0x50(%ebx)
ff 53 50         ; call_1b1: call *0x50(%ebx)
```

Snippet 7: Optimized CFA_M guard

3.4 Optimizations

In certain cases, we can statically determine that the address of the memory location that holds the branch target is always mapped in kernel space. Examples include a branch operand read from the stack (assuming that the attacker has not seized control of the stack pointer), or taken from a global data structure mapped at a fixed address inside the data segment of the kernel. In this case, the first assertion of a CFA_M guard will always be true, since the memory operand points within kernel space. We optimize such scenarios by removing the redundant assertion, effectively reducing the size of the inline guard to 3 instructions. For instance, Snippet 7 depicts how we can optimize the code shown in Snippet 5, assuming that `ebx` is loaded with the address of a global symbol from kernel's data segment. *ret instructions are always confined using the optimized CFA_M variant.*

3.5 Mechanism Protection

CFA_R and CFA_M guards, as presented thus far, provide reliable protection against `ret2usr` attacks, only if the attacker exploits a vulnerability that allows him to partially control a computed branch target. Currently, all the well known and published `ret2usr` exploits, which we analyzed in Section 2 and further discuss in Section 5.1, fall in this category. However, vulnerabilities where the attacker can overwrite kernel memory with arbitrary values also exist [22]. When such flaws are present, exploits could attempt to bypass kGuard. This section discusses how we protect against such attacks.

3.5.1 Bypass Trampolines

To subvert kGuard, an attacker has to be able to determine the address of a (indirect) control transfer instruction inside the text segment of the kernel. Moreover, he should also be able to reliably control the value of its operand (*i.e.*, its branch target). We shall refer to that branch as a *bypass trampoline*. Note that in ISAs with overlapping variable-length instructions, it is possible to find an embedded opcode sequence that translates directly to a control branch in user space [40]. By overwriting the value of a protected branch target with the address of a bypass trampoline, the attacker can successfully execute a jump to user space. The first CFA corresponding to the initially exploited branch will succeed, since the address of the trampoline remains inside the privileged memory segment, while the second CFA that guards the bypass trampoline is completely bypassed by jumping directly to the branch instruction.

Similarly, jumping in the middle of an instruction that contains an indirect branch within, could also be used to subvert kGuard. At this point, we would like to stress that

if an attacker is armed with a powerful exploit for a vulnerability that allows him to overwrite arbitrary kernel memory with arbitrary values, he can easily elevate his privileges by overwriting the credentials associated with a process under his control. In other words, the attacker can achieve his goal without violating the control-flow by jumping into user-level shellcode.

3.5.2 Code Diversification Against Bypasses

kGuard implements two diversification techniques that aid in thwarting attacks exploiting bypass trampolines.

Code inflation This technique *reshapes* the kernel's text area. We begin with randomizing the starting address of the text segment. This is achieved by inserting a random NOP sled at its beginning, which effectively shifts all executable instructions by an arbitrary offset. Next, we continue by inserting NOP sleds of *random length* at the beginning of each CFA. The end result is that the location of every indirect control transfer instruction is randomized, making it harder for an attacker to guess the exact address of a confined branch to use as a bypass trampoline. The effects of the sleds are cumulative because each one pushes all instructions and NOP sleds following, further to higher memory addresses. The size of the initial sled is chosen by kGuard based on the target architecture. For example, in Linux and BSD the kernel space is at least 1GB. Hence, we can achieve more than 20 bits of entropy (*i.e.*, the NOP sled can be $\geq 1\text{MB}$) without notably consuming address space.

The per-CFA NOP sled is randomly selected from a user-configured range. By specifying the range, users can trade higher overhead (both in terms of space and speed), for a smaller probability that an attacker can reliably obtain the address of a bypass trampoline. An important assumption of the aforementioned technique is the secrecy of the kernel's text and symbols. If the attacker has access to the binary image of the confined kernel or is armed with a kernel-level memory leak [32], the probability of successfully guessing the address of a bypass trampoline increases. We posit that assigning safe file permissions to the kernel's text, modules, and debugging symbols is not a limiting factor.⁶ In fact, this is considered standard practice in OS hardening, and is automatically enabled in PaX and similar patches, as well as the latest Ubuntu Linux releases. Also note that the kernel should harden access to the system message ring buffer (`dmesg`), in order to prevent the leakage of kernel addresses.⁷

⁶This can be trivially achieved by changing the permissions in the file system to disallow reads, from non-administrative users, in `/boot` and `/lib/modules` in Linux/FreeBSD, `/bsd` in OpenBSD, *etc.*

⁷In Linux, this can be done by asserting the `kptr_restrict` [24] sysctl option that hides exposed kernel pointers in `/proc` interfaces.

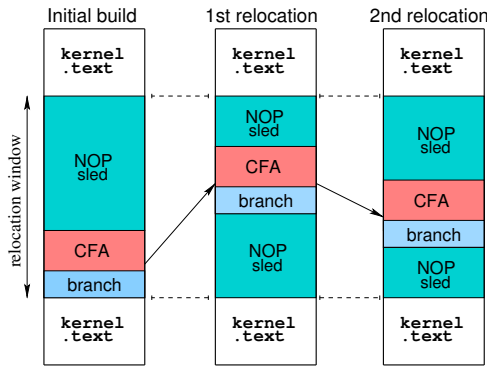


Figure 1: CFA motion synopsis. kGuard relocates each inline guard and protected branch, within a certain window, by routinely rewriting the text segment of the kernel.

CFA motion The basic idea behind this technique is the “continuous” relocation of the protected branches and injected guards, by rewriting the text segment of the kernel. Figure 1 illustrates the concept. During compilation, kGuard emits information regarding each injected CFA, which can be used later for relocating the respective code snippets. Specifically, kGuard logs the exact location of the CFA inside kernel’s text, the type and size of the guard, the length of the prepended NOP sled, as well as the size of the protected branch. Armed with that information, we can then migrate every CFA and indirect branch instruction separately, by moving it inside the following window: $\text{sizeof}(\text{nop_sled}) + \text{sizeof}(\text{cfa}) + \text{sizeof}(\text{branch})$. Currently, we only support CFA motion during kernel bootstrap. In Linux, this is performed after the boot loader (*e.g.*, LILO, GNU GRUB) extracts the kernel image and right before jumping to the `setup()` routine [8]. In BSDs, we perform the relocation after the `boot` program has executed and right before transferring control to the machine-dependent initialization routines (*i.e.*, `mi_startup()` in FreeBSD and `main()` in {Net, Open}BSD [49]). Finally, note that CFA motion can also be performed at runtime, on a live system, by trading runtime overhead for safety. In Section 6, we discuss how we can expand our current implementation, with moderate engineering effort, to support real-time CFA migration.

To further protect against evasion, kGuard can be combined with other techniques that secure kernel code against code-injection [46] and code-reuse attacks [45, 51]. That said, mind that `ret2usr` violations are detected at runtime, and hence one false guess is enough for identifying the attacker and restricting his capabilities (*e.g.*, by revoking his access to prevent brute-force attacks). In Section 6, we further discuss how kGuard can deal with persistent threats.

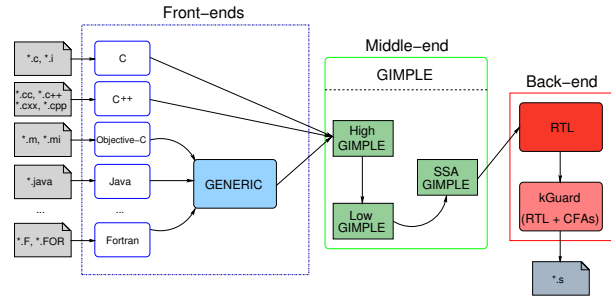


Figure 2: Architectural overview of GCC. The compilation process involves 3 distinct translators (front-end, middle-end, back-end), and more than 250 optimization passes. kGuard is implemented as a back-end optimization pass.

4 Implementation

We implemented kGuard as a set of modifications to the pipeline of a C compiler. Specifically, we instrument the intermediate language (IL) used during the translation process, in order to perform the CFA-based confinement discussed in Section 3. Our implementation consists of a plugin for the GNU Compiler Collection (GCC) that contains the “de-facto” C compiler for building Linux and BSD kernels. Note that although other compilers, such as Clang and PCC, are capable of building much of Linux/FreeBSD and NetBSD/OpenBSD respectively, they are not officially supported by the corresponding development groups, due to the excessive use of the GNU C dialect in the kernel.

Starting with v4.5.1, GCC has been re-designed for facilitating better isolation between its components, and allowing the use of plugins for dynamically adding features to the translators without modifying them. Figure 2 illustrates the internal architecture of GCC. The compilation pipeline is comprised by 3 distinct components, namely the front-end, middle-end, and back-end, which transform the input into various ILs (*i.e.*, GENERIC, GIMPLE, and RTL). The kGuard plugin consists of ~ 1000 lines of code in C and builds into a position-independent (PIC) dynamic shared object that is loaded by GCC. Upon loading kGuard, the plugin manager of GCC invokes `plugin_init()` (*i.e.*, the initialization callback assumed to be exported by every plugin), which parses the plugin arguments (if any) and registers `pass_branchprot` as a new “optimization” pass.⁸ Specifically, we chain our instrumentation callback, namely `branchprot_instrument()`, after the `vartrack` RTL optimization pass, by calling GCC’s `register_callback()` function and requesting to hook with the pass manager (see Figure 2).

⁸Currently, kGuard accepts 3 parameters: `stub`, `nop`, and `log`. `stub` provides the runtime violation handler, `nop` stores the maximum size of the random NOP sled inserted before each CFA, and `log` is used to define an instrumentation logfile for CFA motion.

The reasons for choosing to implement the instrumentation logic at the RTL level, and not as annotations to the GENERIC or GIMPLE IL, are mainly the following. First, by applying our assertions after most of the important optimizations have been performed, which may result into moving or transforming instructions, we guarantee that we instrument only relevant code. For instance, we do not inject CFAs for dead code or control transfers that, due to optimization transformations like inline expansion, do not need to be confined. Second, we secure implicit control transfers that are exposed later in the translation (*e.g.*, after the High-GIMPLE IL has been “lowered”). Third, we tightly couple the CFAs with the corresponding unsafe control transfers. This way, we protect the guards from being removed or shifted from the respective points of check, due to subsequent optimization passes (*e.g.*, code motion). For more information regarding the internals of RTL instrumentation, interested readers are referred to Appendix B.

5 Evaluation

In this section, we present the results from the evaluation of kGuard both in terms of performance and effectiveness. Our testbed consisted of a single host, equipped with two 2.66GHz quad-core Intel Xeon X5500 CPUs and 24GB of RAM, running Debian Linux v6 (“squeeze” with kernel v2.6.32). Note that while conducting our performance measurements, the host was idle with no other user processes running apart from the evaluation suite. Moreover, the results presented here are mean values, calculated after running 10 iterations of each experiment; the error bars correspond to 95% confidence intervals. kGuard and the corresponding Linux kernels were compiled with GCC v4.5.1, and unless otherwise noted, we used Debian’s default configuration that results into a complete build of the kernel, including all modules and device drivers. Finally, we configured kGuard to use a random NOP sled of 20 instructions on average. Mind you that we also measured the effect of various NOP sled sizes, which was insignificant for the range 0 – 20.

5.1 Preventing Real Attacks

The main goal of the effectiveness evaluation is to apply kGuard on commodity OSs, and determine whether it can detect and prevent real-life ret2usr attacks. Table 1 summarizes our test suite, which consisted of a collection of 8 exploits that cover a broad spectrum of different flaws, including direct NULL pointer dereferences, control hijacking via tampered data structures (data pointer corruption), function and data pointer overwrite, arbitrary kernel-memory nullification, and ret2usr via kernel stack-smashing.

	x86 kernel			x86-64 kernel		
	call	jmp	ret	call	jmp	ret
<i>CFAM</i>	20767	1803	—	17740	1732	—
<i>CFAMopt</i>	2253	12	113053	1789	0	105895
<i>CFAR</i>	6325	0	—	8780	0	—
Total	29345	1815	113053	28309	1732	105895

Table 2: Number of indirect branches instrumented by kGuard in the vanilla Linux kernel v2.6.32.39.

We instrumented 10 different vanilla Linux kernels, ranging from v2.6.18 up to v2.6.34, both in x86 and x86-64 architectures. Additionally, in this experiment, we used a home-grown violation handler for demonstrating the customization features of kGuard. Upon the detection of a ret2usr attack, the handler takes a snapshot of the memory that contains the user-provided code for analyzing the behavior of the offending process. Such a feature could be useful in a honeypot setup for performing malware analysis and studying new ret2usr exploitation vectors. All kernels were compiled with and without kGuard, and tested against the respective set of exploits. In every case, we were able to successfully detect and prevent the corresponding exploitation attempt. Also note that the tested exploits circumvented the page mapping restrictions of Linux, by using one or more of the techniques discussed in Section 2.3.

5.2 Translation Overhead

We first quantify the additional time needed to inspect the RTL IL and emit the CFAs (see Section 4). Specifically, we measured the total build time with Unix’s `time` utility, when compiling the v2.6.32.39 Linux kernel natively and with kGuard. On average, we observed a 0.3% increase on total build time on the x86 architecture, and 0.05% on the x86-64. Moreover, the size of the kernel image/modules was increased by 3.5%/0.43% on the x86, and 5.6%/0.56% on the x86-64.

In Table 2, we show the number of exploitable branches instrumented by kGuard, categorized by architecture, and confinement and instruction type. As expected, `ret` instructions dominate the computed branches. Note that both in x86 and x86-64 scenarios, we were able to optimize approximately 10% of the total indirect calls via memory locations, using the optimization scheme presented in Section 3.4. Overall, the `drivers/` subsystem was the one with the most instrumentations, followed by `fs/`, `net/`, and `kernel/`. Additionally, a significant amount of instrumented branches was due to architecture-dependent code (`arch/`) and “inlined” C functions (`include/`).

Vulnerability	Description	Impact	Exploit	
			x86	x86-64
CVE-2009-1897	NULL <i>function</i> pointer dereference in <i>drivers/net/tun.c</i> due to compiler optimization	2.6.30–2.6.30.1	✓	—
CVE-2009-2692	NULL <i>function</i> pointer dereference in <i>net/socket.c</i> due to improper initialization	2.6.0–2.6.30.4	✓	✓
CVE-2009-2908	NULL <i>data</i> pointer dereference in <i>fs/ecryptfs/inode.c</i> due to a negative reference counter (function pointer affected via tampered data flow)	2.6.31	✓	✓
CVE-2009-3547	<i>data</i> pointer corruption in <i>fs/pipe.c</i> due to a use-after-free bug (function pointer under user control via tampered data structure)	≤ 2.6.32-rc6	✓	✓
CVE-2010-2959	<i>function</i> pointer overwrite via integer overflow in <i>net/can/bcm.c</i>	2.6.{27.x, 32.x, 35.x}	✓	—
CVE-2010-4258	<i>function</i> pointer overwrite via arbitrary kernel memory nullification in <i>kernel/exit.c</i>	≤ 2.6.36.2	✓	✓
EDB-15916	NULL <i>function</i> pointer overwrite via a signedness error in Phonet protocol (function pointer affected via tampered data structure)	2.6.34	✓	✓
CVE-2009-3234	ret2usr via kernel stack buffer overflow in <i>kernel/perf_counter.c</i> (return address is overwritten with user space memory)	2.6.31-rc1	✓	✓

✓: detected and prevented successfully —: exploit unavailable

Table 1: Effectiveness evaluation suite. We instrumented 10 x86/x86-64 vanilla Linux kernels, ranging from v2.6.18 to v2.6.34, for assessing kGuard. We successfully detected and prevented all the listed exploits.

5.3 Performance Overhead

The injected CFAs also introduce runtime latency. We evaluated kGuard to quantify this overhead and establish a set of performance bounds for different types of system services. Moreover, we used the overhead imposed by PaX (*i.e.*, UDEREF [53] and KERNEXEC) as a reference. Mind you that on x86, PaX offers protection against ret2usr attacks by utilizing the segmentation unit for isolating the kernel from user space. In x86-64 CPUs, where segmentation is not supported by the hardware, it temporarily remaps user space into a different location with non-execute permissions.

Macro benchmarks We begin with the evaluation of kGuard using a set of real-life applications that represent different workloads. In particular, we used a kernel build and two popular server applications. The Apache web server, which performs mainly I/O, and the MySQL RDBMS that is both I/O driven and CPU intensive. We run all the respective tests over a vanilla Linux kernel v2.6.32.39, the same kernel patched with PaX, and instrumented with kGuard.

First, we measured the time taken to build a vanilla Linux kernel (v2.6.32.39), using the Unix `time` utility. On the x86, the PaX-protected kernel incurs a 1.26% run-time overhead, while on the x86-64 the overhead is 2.89%. In contrast, kGuard ranges between 0.93% on x86-64, and 1.03% on x86. Next, we evaluated MySQL v5.1.49 using its own benchmark suite (`sql-bench`). The suite consists of four different tests, which assess the completion time of various DB operations, like table creation and modification, data selection and insertion, and so forth. On average, kGuard’s slowdown ranges from 0.85% (x86-64) to 0.93% (x86), while PaX lies between 1.16% (x86) and 2.67% (x86-64). Finally, we measured Apache’s performance using its own utility `ab` and static HTML files of different size. We used Apache v2.2.16 and configured it to pre-fork all the worker processes (pre-forking is a standard multiprocessing module), in order to avoid high fluctuations in performance,

due to Apache spawning extra processes for handling the incoming requests at the beginning of our experiments. We chose files with sizes of 1KB, 10KB, 100KB, and 1MB, and measured the average throughput in requests per second (req/sec). All other options were left to their default setting. The kernel patched with PaX incurs an average slowdown that ranges between 0.01% and 0.09% on the x86, and 0.01% and 1.07% on x86-64. In antithesis, kGuard’s slowdown lies between 0.001% and 0.01%. Overall, our results indicate that in both x86 and x86-64 Linux the impact of kGuard in real-life applications is negligible (≤1%).

Micro benchmarks Since the injected CFAs are distributed throughout many kernel subsystems, such as the essential `net/` and `fs/`, we used the LMBench [50] microbenchmark suite to measure the impact of kGuard on the performance of core kernel system calls and facilities. We focus on both latency and bandwidth. For the first, we measured the latency of entering the OS, by investigating the `null` system call (`syscall`) and the most frequently used I/O-related calls: `read`, `write`, `fstat`, `select`, `open/close`. Additionally, we measured the time needed to install a signal with `sigaction`, inter-process communication (IPC) latency with `socket` and `pipe`, and process creation latency with `fork+{exit, execve, /bin/sh}`.

Figure 3 summarizes the latency overhead of kGuard in contrast to the vanilla Linux kernel and a kernel with the PaX patch applied and enabled. Note that the time is measured in microseconds (μsec). kGuard ranges from 2.7% to 23.5% in x86 (average 11.4%), and 2.9% to 19.1% in x86-64 (average 10.3%). In contrast, the PaX-protected kernel exhibits a latency ranging between 5.6% and 257% (average 84.5%) on the x86, whereas on x86-64, the latency overhead ranges between 19% and 531% (average 172.2%). Additionally, kGuard’s overhead for process creation (in both architectures) lies between 7.1% and 9.7%, whereas PaX ranges from 8.1% to 56.3%.

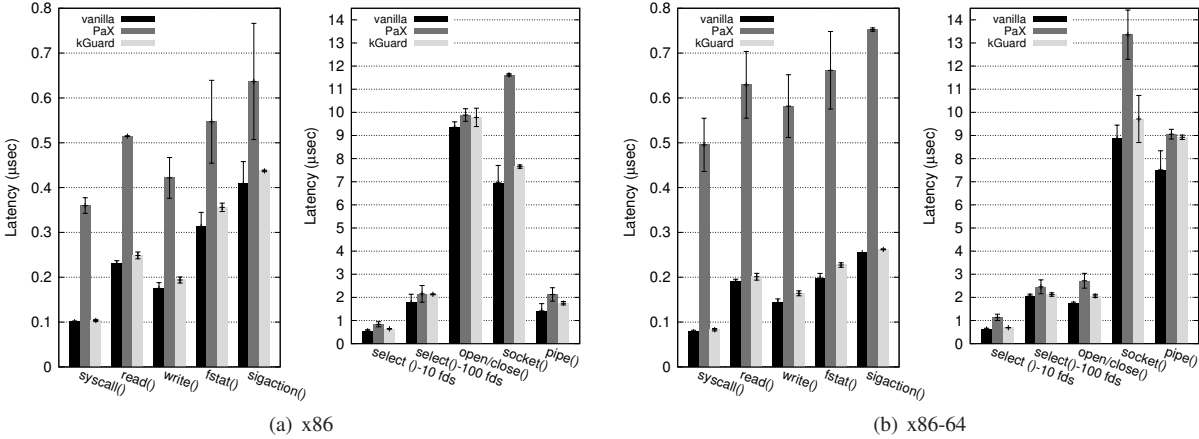


Figure 3: Latency overhead incurred by kGuard and PaX on essential system calls (x86/x86-64 Linux).

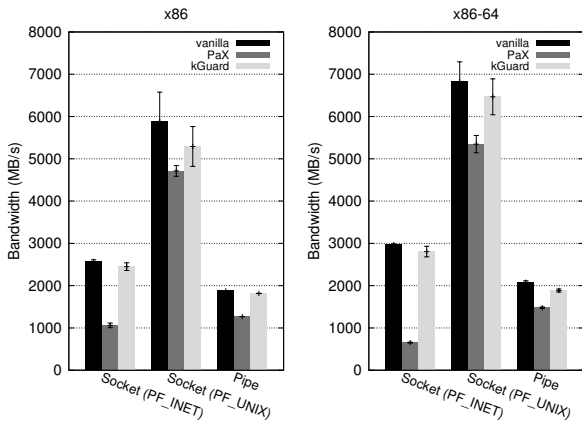


Figure 4: IPC bandwidth achieved by kGuard and PaX, using TCP (PF_INET), Unix sockets (PF_UNIX), and pipes.

As far as bandwidth is concerned, we measured the degradation imposed by kGuard and PaX in the maximum achieved bandwidth of popular IPC facilities, such as sockets and pipes. Figure 4 shows our results (bandwidth is measured in MB/s). kGuard’s slowdown ranges between 3.2% – 10% on x86 (average 6%), and 5.25% – 9.27% on x86-64 (average 6.6%). PaX’s overhead lies between 19.9% – 58.8% on x86 (average 37%), and 21.7% – 78% on x86-64 (average 42.8%). Overall, kGuard exhibits lower overhead on x86-64, due to the fewer CFA_M guards (see Table 2). Recall that CFA_R confinement can be performed with just 3 additional instructions, and hence incurs less run-time overhead, whereas CFA_M might need up to 10 (*e.g.*, when we cannot optimize). However, the same is not true for PaX, since the lack of segmentation in x86-64 results in higher performance penalty.

6 Discussion and Future Work

Custom violation handlers kGuard’s default violation handler appends a message in system log and halts the OS. We coerce assertion violations into a kernel fail-stop to prevent brute-force attempts and to avoid leaving the OS in inconsistent states (*e.g.*, by aborting an in-flight kernel thread that holds a lock). However, kGuard can be configured to use a custom handler. Upon enabling this option, our instrumentation becomes slightly different. Instead of overwriting offending branch targets with the address of our handler, we push the value of the branch target into the stack and invoke the handler directly. In the case of a CFA_R guard this means that the `mov` instruction (see Snippet 3) will be replaced with a `push` and `call`. CFA_M guards are modified accordingly.

This instrumentation increases slightly the size of our inline guards, but does not incur additional overhead, since the extra instructions are on the error path. Additionally, the custom violation handler has access to the location where the violation occurred, by reading the return address of the callee (pushed into the stack from `call`), as well as to the offending branch target (passed as argument to the handler). Using that information, one can implement adaptive defense mechanisms, including selective confinement (*e.g.*, deal with VMware’s I/O backdoor that needs to “violate” protection domains), error virtualization [63], as well as forensic analysis (*e.g.*, dump the shellcode). The latter can be useful in honeypot setups for studying new ret2usr exploitation vectors.

Persistent threats By building upon the previous feature, we implemented a handler that actively responds to persistent threats (*i.e.*, users that repeatedly try to perform a ret2usr attack). Once invoked, due to a violation, it performs the following. First, it checks the execution context of the kernel to identify if it runs inside a user-

level process or an interrupt handler. If the violation occurred while executing an interrupt service routine, or the current execution path is holding a lock⁹, then we fail-stop the kernel. Else, if the kernel is preemptible, we terminate all processes with the same `uid` of the offending process and prevent the user from logging in. Other possible approaches include inserting an exponentially increased delay for user logins (*i.e.*, make the bruteforce attack slow and impractical), activate CFA motion, *etc.*

Future considerations Currently, we investigate how to apply the CFA motion technique (see Section 3.5), while a kernel is running and the OS is live. Our early Linux prototype utilizes a dedicated kernel thread, which upon a certain condition, freezes the kernel and performs rewriting. Thus far, we achieve CFA relocation in a coarse-grained manner, by exploiting the suspend subsystem of the Linux kernel. Specifically, we bring the system to pre-suspend state for preventing any kernel code from being invoked during the relocation (note that the BSD OSs have similar facilities). Possible events to initiate live CFA motion are the number of executed system calls or interrupts (*i.e.*, diversify the kernel every n invocation events), CFA violations, or in the case of smartphone devices, dock station attach and charging. However, our end goal is to perform CFA motion in a more fine-grained, non-interruptible and efficient manner, without “locking” the whole OS.

7 Related Work

kGuard is inspired by the numerous compiler-based techniques that explicitly or implicitly constrain control flow and impose a specific execution policy. StackGuard [14] and ProPolice [34] are GCC patches that extend the behavior of the translator for inserting a canary word prior to the saved return address on the stack. The canary is checked again before a function return is performed, and execution is halted if it has been overwritten (*e.g.*, due to a stack-smashing attack). Stack Shield [1] is a similar extension that saves the return address, upon function entry, into a write-protected memory area that is not affected by buffer overflows and restores it before returning.

Generally, these approaches have limitations [9, 69]. However, they significantly mitigate real-life exploits by assuring that functions will always return to caller sites, incur low performance overhead, and do not require any change to the runtime environment or platform of the protected applications. For these reasons, they have been adopted by mainstream compilers, such as GCC, and enabled by default in many BSD and Linux distributions.

⁹In Linux, we can check if the kernel is holding locks by looking at the `preempt_count` variable in the current process's `thread_info` structure [48].

kGuard operates analogously, by hooking to the compilation process and dynamically instrumenting code with inline guards. However, note that we leverage the plugin API of GCC, and do not require patching the compiler itself, thus aiding the adoption of kGuard considerably. More importantly, since stack protection is now enabled by default, kGuard can be configured to offload the burden of dealing with the integrity of return control data to GCC. If random XOR canaries [14] are utilized, then any attempt to tamper with saved return addresses on the stack, for redirecting the privileged control flow to user space, will be detected and prevented. Hence, the protection of kernel-level `ret` instructions with CFAs can be turned off. Note that during our preliminary evaluation we also measured such a scenario. The average overhead of kGuard, with no `ret` protection, on system call and I/O latency was 6.5% on x86 and 5.4% on x86-64, while its impact on real-life applications was $\leq 0.5\%$. This “offloading” cannot be performed in the case of simple random canaries or terminator canaries. Nevertheless, it demonstrates that our approach is indeed orthogonal to complementary mitigation schemes, and operates nicely with confinement checks injected during compile time.

PointGuard [13] is another GCC extension that works by encrypting all pointers while they reside in memory and decrypting them before they are loaded into a CPU register. PointGuard could provide some protection against `ret2usr` attacks, especially if a function pointer is read directly from user-controlled memory [20]. However, it cannot deal with cases where an attacker can nullify kernel-level function pointers by exploiting a race condition [19] or supplying carefully crafted arguments to buggy system calls [23]. In such scenarios, the respective memory addresses are altered by legitimate code (*i.e.*, kernel execution paths), and not directly by the attacker. kGuard provides solid protection against `ret2usr` attacks by policing every computed control transfer for kernel/user space boundary violations.

Other compiler-based approaches include DFI [11] that enforces data flow integrity based on a statically calculated reaching definition analysis. However, the main focus of DFI, and similar techniques [3, 12, 33], is the enforcement of spatial safety for mitigating bounds violations and preventing bounds-related vulnerabilities.

Control-Flow Integrity (CFI) [2], Program Shepherding [43], and Strata [57], employ binary rewriting and dynamic binary instrumentation (DBI) for retrofitting security enforcement capabilities into unmodified binaries. The major issue with such approaches has been mainly the large performance overhead they incur, as well as the reliance on interpretation engines, which complicates their adoption. Program Shepherding exhibits $\sim 100\%$ overhead on SPEC benchmarks, while CFI has an average overhead of 15%, and a maximum of 45%, on the

same test suite. CFI-based techniques rewrite programs so that every branch target is given a label, and each indirect branch instruction is prepended with a check, which ensures that the target's label is in accordance with a pre-computed control-flow graph (CFG). Unfortunately, CFI is not effective against `ret2usr` attacks. The integrity of the CFI mechanism is guaranteed as long as the attacker cannot overwrite the code of the protected binary, or execute user-provided data. However, during a `ret2usr` attack, the attacker completely controls user space memory, both in terms of contents and rights. Therefore, CFI can be subverted by prepending user-provided shellcode with the respective label.

As an example, consider again Snippet 1 and assume that the attacker has managed to overwrite the function pointer `sendpage` with an address pointing in user space. CFI will prepend the instruction that invokes `sendpage` with an inline check that fetches a label ID (placed right before the first instruction in functions that `sendpage` can point to), and compares it with the allowed label IDs. If the two labels match, the control transfer will be authorized. Unluckily, since the attacker controls the contents and rights of the memory that `sendpage` is now pointing, he can easily prepend his code with the label ID that will authorize the control transfer. Furthermore, Petroni and Hicks [55] noted that computing in advance a precise CFG for a modern kernel is a nontrivial task, due to the rich control structure and the several levels of interrupt handling and concurrency. CFI-based proposals can be combined with kGuard to overcome the individual limitations of each technique. kGuard can guarantee that privileged execution will always be confined in kernel space, thus leaving no other options to attackers than targeting kernel-level control flow violations, which can be solidly protected by CFI.

Garfinkel and Rosenblum proposed Livewire [36], which was the first system that used a virtual machine monitor (VMM) for implementing invariant-based kernel protection. Similarly, Grizzard uses a VMM for monitoring kernel execution and validating control flow [38]. For LMBench, he reports an average of 30% overhead, and a maximum of 74%, on top of VMM's performance penalty. SecVisor [62] is a tiny hypervisor that ensures the integrity of commodity OS kernels. It relies on physical memory virtualization for protecting against code injection attacks and kernel rootkits, by allowing only approved code to execute in kernel mode and ensuring that such code cannot be modified. However, it requires modern CPUs that support virtualization in hardware, as well as kernel patching to add the respective hypercalls that authorize module loading. Along the same lines, NICKLE [56] offers similar guarantees, without requiring any OS modification, by relying on an innovative memory shadowing scheme and real-time kernel

code authentication via VMM introspection. Petroni and Hicks proposed state-based CFI (SBCFI) [55], which reports violations of the kernel's control flow due to the presence of rootkits. Similarly, Lares [54] and Hook-Safe [68] protect kernel hooks (including function pointers) from being manipulated by kernel malware. The focus of those techniques, however, has been kernel attestation and kernel code integrity [10], which is different from the control-flow integrity of kernel code. On the other hand, kGuard focuses on solving a different problem: *privilege escalation* via hijacked kernel-level execution paths. Although VMMs provide stronger security guarantees than kGuard, and SecVisor and NICKLE can prevent `ret2usr` attacks by refusing execution from user space while running in kernel mode, they incur larger performance penalties and require running the whole OS over custom hypervisors and specialized hardware. It is also worth noting that SecVisor and NICKLE cannot protect against execution hijacking via tampered data structures containing control data [18,20]. kGuard offers solid protection against that type of `ret2usr` due to the way it handles control data stored in memory.

Supervisor Mode Execution Prevention (SMEP) [37] is an upcoming Intel CPU feature, which prevents code executing in kernel mode from branching to code located in pages without the supervisor bit set in their page table entry. Although it allows for a confinement mechanism similar to PaX with zero performance penalty, it is platform specific (*i.e.*, x86, x86-64), requires kernel patching, and does not protect legacy systems.

8 Conclusions

We presented kGuard, a lightweight compiler-based mechanism that protects the kernel from `ret2usr` attacks. Unlike previous work, kGuard is *fast*, *flexible*, and offers *cross-platform* support. It works by injecting fine-grained inline guards during the translation phase that are resistant to bypass, and it does not require any modification to the kernel or additional software such as a VMM. kGuard can safeguard 32- or 64-bit OSs that map a mixture of code segments with different privileges inside the same scope and are susceptible to `ret2usr` attacks. We believe that kGuard strikes a balance between safety and functionality, and provides comprehensive protection from `ret2usr` attacks, as demonstrated by our extensive evaluation with real exploits against Linux.

Availability

The prototype implementation of kGuard is freely available at: <http://www.cs.columbia.edu/~vpk/research/kguard/>

Acknowledgments

We thank Michalis Polychronakis and Willem de Bruijn for their valuable feedback on earlier drafts of this paper. This work was supported by DARPA and the US Air Force through Contracts DARPA-FA8750-10-2-0253 and AFRL-FA8650-10-C-7024, respectively. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, DARPA, or the Air Force.

References

- [1] Stack Shield. <http://www.angelfire.com/sk/stackshield/>, January 2000.
- [2] ABADI, M., BUDI, M., ERLINGSSON, U., AND LIGATTI, J. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)* (2005), pp. 340–353.
- [3] AKRITIDIS, P., CADAR, C., RAICIU, C., COSTA, M., AND CASTRO, M. Preventing memory error exploits with WIT. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P)* (2008), pp. 263–277.
- [4] AZAB, A. M., NING, P., WANG, Z., JIANG, X., ZHANG, X., AND SKALSKY, N. C. HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)* (2010), pp. 38–49.
- [5] BELLARD, F. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the 7th USENIX Annual Technical Conference (FREENIX track)* (2005), pp. 41–46.
- [6] BICKFORD, J., O’HARE, R., BALIGA, A., GANAPATHY, V., AND IFTODE, L. Rootkits on Smart Phones: Attacks, Implications and Opportunities. In *Proceedings of the 11th International Workshop on Mobile Computing Systems and Applications (Hot-Mobile)* (2010), pp. 49–54.
- [7] BLETSCH, T., JIANG, X., FREEH, V. W., AND LIANG, Z. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2011), pp. 30–40.
- [8] BOVET, D. P., AND CESATI, M. *Understanding the Linux Kernel*, 3rd ed. O’Reilly Media, Sebastopol, CA, USA, 2005, ch. System Startup, pp. 835–841.
- [9] BULBA AND KIL3R. Bypassing StackGuard and StackShield. *Phrack* 5, 56 (May 2000).
- [10] CARBONE, M., CUI, W., LU, L., LEE, W., PEINADO, M., AND JIANG, X. Mapping Kernel Objects to Enable Systematic Integrity Checking. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)* (2009), pp. 555–565.
- [11] CASTRO, M., COSTA, M., AND HARRIS, T. Securing Software by Enforcing Data-Flow Integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 147–160.
- [12] CASTRO, M., COSTA, M., MARTIN, J.-P., PEINADO, M., AKRITIDIS, P., DONNELLY, A., BARHAM, P., AND BLACK, R. Fast Byte-granularity Software Fault Isolation. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)* (2009), pp. 45–58.
- [13] COWAN, C., BEATTIE, S., JOHANSEN, J., AND WAGLE, P. PointGuardTM: Protecting Pointers From Buffer Overflow Vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium (USENIX Sec)* (2003), pp. 91–104.
- [14] COWAN, C., PU, C., MAIER, D., HINTON, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium (USENIX Sec)* (1998), pp. 63–78.
- [15] COX, M. J. Red Hat’s Top 11 Most Serious Flaw Types for 2009. <http://www.awe.com/mark/blog/20100216.html>, February 2010.
- [16] CVE. CVE-2009-1897. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1897>, June 2009.
- [17] CVE. CVE-2009-2692. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2692>, August 2009.
- [18] CVE. CVE-2009-2908. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2908>, August 2009.
- [19] CVE. CVE-2009-3527. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3527>, October 2009.
- [20] CVE. CVE-2009-3547. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3547>, October 2009.
- [21] CVE. CVE-2010-2959. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2959>, August 2010.
- [22] CVE. CVE-2010-3904. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3904>, October 2010.
- [23] CVE. CVE-2010-4258. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-4258>, November 2010.
- [24] DAN ROSENBERG. `kpitr_restrict` for hiding kernel pointers. <http://lwn.net/Articles/420403/>, December 2010.
- [25] DE C VALLE, R. Linux `sock_sendpage()` NULL Pointer Dereference (PPC/PPC64 exploit). http://packetstormsecurity.org/files/81212/Linux-sock_sendpage-NULL-Pointer-Dereference.html, September 2009.
- [26] DE RAADT, T. CVS-200910282103. <http://marc.info/?l=openbsd-cvs&m=125676466108709&w=2>, October 2009.
- [27] DESIGNER, S. Getting around non-executable stack (and fix). <http://seclists.org/bugtraq/1997/Aug/63>, August 1997.
- [28] DOSEMU. DOS Emulation. <http://www.dosemu.org>, June 2012.
- [29] DOWD, M. Application-Specific Attacks: Leveraging The ActionScript Virtual Machine. Tech. rep., IBM Corporation, April 2008.
- [30] EDB. EDB-9477. <http://www.exploit-db.com/exploits/9477/>, August 2009.
- [31] EDB. EDB-17391. <http://www.exploit-db.com/exploits/17391/>, June 2011.
- [32] EDB. EDB-18080. <http://www.exploit-db.com/exploits/18080/>, November 2011.

- [33] ERLINGSSON, U., ABADI, M., VRABLE, M., BUDI, M., AND NECULA, G. C. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 75–88.
- [34] ETOH, H. GCC extension for protecting applications from stack-smashing attacks. <http://www.tr1.ibm.com/projects/security/ssp/>, August 2005.
- [35] FREEBSD. sysutils/vbetool doesn't work with FreeBSD 8.0-RELEASE and STABLE. <http://forums.freebsd.org/showthread.php?t=12889>, April 2010.
- [36] GARFINKEL, T., AND ROSENBLUM, M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)* (February 2003).
- [37] GEORGE, V., PIAZZA, T., AND JIANG, H. Technology Insight: Intel©Next Generation Microarchitecture Codename Ivy Bridge. www.intel.com/inf/library/pdf/sf_2011/SF11_SPCS005_101F.pdf, September 2011.
- [38] GRIZZARD, J. B. *Towards Self-Healing Systems: Re-establishing Trust in Compromised Systems*. PhD thesis, Georgia Institute of Technology, 2006.
- [39] HARDY, N. The Confused Deputy (or why capabilities might have been invented). *SIGOPS Operating Systems Review* 22, 4 (October 1988), 36–38.
- [40] HUND, R., HOLZ, T., AND FREILING, F. C. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proceedings of the 18th USENIX Security Symposium (USENIX Sec)* (2009), pp. 383–398.
- [41] INGO MOLNAR. 4G/4G split on x86, 64 GB RAM (and more) support. <http://lwn.net/Articles/39283/>, July 2003.
- [42] KC, G. S., KEROMYTIS, A. D., AND PREVELAKIS, V. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)* (2003), pp. 272–280.
- [43] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium (USENIX Sec)* (2002), pp. 191–206.
- [44] KORTCHINSKY, K. CLOUDBURST: A VMware Guest to Host Escape Story. In *Proceedings of the 12th Black Hat USA* (2009).
- [45] LI, J., WANG, Z., JIANG, X., GRACE, M., AND BAHAM, S. Defeating Return-Oriented Rootkits With “Return-less” Kernels. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)* (2010), pp. 195–208.
- [46] LIAKH, S., GRACE, M., AND JIANG, X. Analyzing and Improving Linux Kernel Memory Protection: A Model Checking Approach. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)* (2010), pp. 271–280.
- [47] LOSCOCCO, P., AND SMALLEY, S. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the 3rd USENIX Annual Technical Conference (FREENIX track)* (2001), pp. 29–42.
- [48] LOVE, R. *Linux Kernel Development*, 2nd ed. Novel Press, Indianapolis, IN, USA, 2005.
- [49] MCKUSICK, M. K., BOSTIC, K., KARELS, M. J., AND QUARTERMAN, J. S. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996, ch. Kernel Services, pp. 49–73.
- [50] MCVOY, L., AND STAELIN, C. Imbench: Portable Tools for Performance Analysis. In *Proceedings of the 1st USENIX Annual Technical Conference (USENIX ATC)* (1996), pp. 279–294.
- [51] ONARLIOGLU, K., BILGE, L., LANZI, A., BALZAROTTI, D., AND KIRDA, E. G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)* (2010), pp. 49–58.
- [52] PAX. Homepage of The PaX Team. <http://pax.grsecurity.net>, June 2012.
- [53] PAX TEAM. UDEREf. <http://grsecurity.net/~spender/uderef.txt>, April 2007.
- [54] PAYNE, B. D., CARBONE, M., SHARIF, M., AND LEE, W. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P)* (2008), pp. 233–247.
- [55] PETRONI, JR., N. L., AND HICKS, M. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)* (October 2007), pp. 103–115.
- [56] RILEY, R., JIANG, X., AND XU, D. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID)* (2008), pp. 1–20.
- [57] SCOTT, K., AND DAVIDSON, J. Safe Virtual Execution Using Software Dynamic Translation. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC)* (December 2002), pp. 209–218.
- [58] SECURITYFOCUS. Xbox 360 Hypervisor Privilege Escalation Vulnerability. <http://www.securityfocus.com/archive/1/461489>, February 2007.
- [59] SECURITYFOCUS. BID 36587. <http://www.securityfocus.com/bid/36587>, October 2009.
- [60] SECURITYFOCUS. BID 36939. <http://www.securityfocus.com/bid/36939>, November 2009.
- [61] SECURITYFOCUS. BID 43060. <http://www.securityfocus.com/bid/43060>, September 2010.
- [62] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)* (2007), pp. 335–350.
- [63] SIDIROGLOU, S., LAADAN, O., PEREZ, C. R., VIENNOT, N., NIEH, J., AND KEROMYTIS, A. D. ASSURE: Automatic Software Self-healing Using REscue points. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2009), pp. 37–48.
- [64] SPENGLER, B. On exploiting null ptr derefs, disabling SELinux, and silently fixed linux vulns. <http://seclists.org/dailydave/2007/q1/224>, March 2007.
- [65] STEINBERG, U., AND KAUER, B. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)* (2010), pp. 209–222.
- [66] TINNES, J. Bypassing Linux NULL pointer dereference exploit prevention (mmap_min_addr). <http://blog.cr0.org/2009/06/bypassing-linux-null-pointer.html>, June 2009.
- [67] WANG, Z., AND JIANG, X. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P)* (2010), pp. 380–395.
- [68] WANG, Z., JIANG, X., CUI, W., AND NING, P. Countering Kernel Rootkits with Lightweight Hook Protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)* (2009), pp. 545–554.

- [69] WILANDER, J., AND KAMKAR, M. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. In *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)* (2003).
- [70] WINEHQ. Run Windows applications on Linux, BSD, Solaris and Mac OS X. <http://www.winehq.org>, June 2012.
- [71] WOJTCZUK, R. Subverting the Xen hypervisor. In *Proceedings of the 11th Black Hat USA* (2008).
- [72] ZENG, B., TAN, G., AND MORRISETT, G. Combining Control-flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)* (2011), pp. 29–40.

A Step-by-step Analysis of the sendpage ret2usr Exploit

Figure 5 illustrates the steps taken by a malicious process to exploit the vulnerability shown in Snippet 1 (in x86). It starts by invoking the `sendfile` system call with the offending arguments (*i.e.*, a datagram socket of a vulnerable protocol family, such as `PF_IPX`). The corresponding `libc` wrapper (0xb7f50d20) traps to the OS via the `sysenter` instruction (0xb7fe2419) and the generated software interrupt leads to executing the system call handler of Linux (`sysenter_do_call()`). The handler dynamically resolves the address of `sys_sendfile` (0xc01d0ccf) using the array `sys_call_table`, which includes the kernel-level address of every supported system call indexed by system call number (0xc01039db).¹⁰ Privileged execution then continues until the offending `sock_sendpage()` routine is invoked. Due to the arguments passed in `sendfile`, the value of the `sendpage` pointer (0xfa7c8538) is NULL and results in an indirect function call to address zero. This transfers control to the attacker, who can execute arbitrary code with kernel privileges.

B GCC RTL Instrumentation Internals

`branchprot_instrument()`, our instrumentation callback, is invoked by GCC’s pass manager for every translation unit after all the RTL optimizations have been applied, and exactly before target code is emitted. At that point, the corresponding translation unit is maintained as graph of basic blocks (BBs) that contain chained sequences of RTL instructions, also known as `rtx` expressions (*i.e.*, LISP-like assembler code for an abstract machine with infinite registers). GCC maintains a specific graph-based data structure (`call-graph`) that holds information for every internal/external call site. However, indirect control transfers are not represented in it

¹⁰The address 0xc03fd3a8 corresponds to the kernel-level memory address of `sys_call_table`.

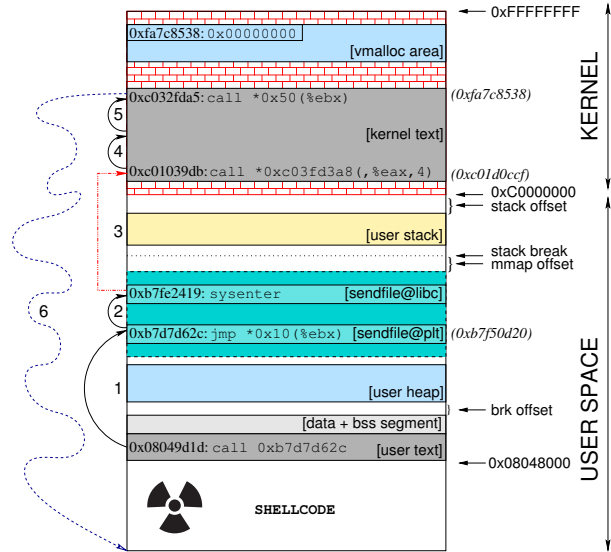


Figure 5: Control transfers that occur during the exploitation of a `ret2usr` attack. The `sendfile` system call, on x86 Linux, causes a function pointer in kernel to become NULL, illegally transferring control to user space code.

and are assumed to be control-flow neutral. For that reason we perform the following. We begin by iterating over all the BBs and `rtx` expressions of the respective translation unit, selecting only the computed calls and jumps. This includes `rtx` objects of type `CALL_INSN` or `JUMP_INSN` that branch via a register or memory location. Note that `ret` instructions are also encoded as `rtx` objects of type `JUMP_INSN`. Next, we modify the `rtx` expression stream for inserting the `CFAR` and `CFAM` guards. The `CFAR` guards are inserted by splitting the original BB into 3 new ones. The first hosts all the `rtx` expressions before `{CALL, JUMP}_INSN`, along with the random NOP sled and two more `rtx` expressions that match the compare (`cmp`) and jump (`jae`) instructions shown in Snippet 3. The second BB contains the code for loading the address of the violation handler into the branch register (*i.e.*, `mov` in x86), while the last BB contains the actual branch expression along with the remaining `rtx` expressions of the original BB. Note that the process also involves altering the control-flow graph, by chaining the new BBs accordingly and inserting the proper branch labels to ensure that the injected code remains inlined. `CFAM` instrumentation is performed in a similar fashion.

Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization

Cristiano Giuffrida
Vrije Universiteit, Amsterdam
giuffrida@cs.vu.nl

Anton Kuijsten
Vrije Universiteit, Amsterdam
akuijst@cs.vu.nl

Andrew S. Tanenbaum
Vrije Universiteit, Amsterdam
ast@cs.vu.nl

Abstract

In recent years, the deployment of many application-level countermeasures against memory errors and the increasing number of vulnerabilities discovered in the kernel has fostered a renewed interest in kernel-level exploitation. Unfortunately, no comprehensive and well-established mechanism exists to protect the operating system from arbitrary attacks, due to the relatively new development of the area and the challenges involved.

In this paper, we propose the first design for fine-grained address space randomization (ASR) inside the operating system (OS), providing an efficient and comprehensive countermeasure against classic and emerging attacks, such as return-oriented programming. To motivate our design, we investigate the differences with application-level ASR and find that some of the well-established assumptions in existing solutions are no longer valid inside the OS; above all, perhaps, that information leakage becomes a major concern in the new context. We show that our ASR strategy outperforms state-of-the-art solutions in terms of both performance and security without affecting the software distribution model. Finally, we present the first comprehensive *live rerandomization* strategy, which we found to be particularly important inside the OS. Experimental results demonstrate that our techniques yield low run-time performance overhead (less than 5% on average on both SPEC and syscall-intensive benchmarks) and limited run-time memory footprint increase (around 15% during the execution of our benchmarks). We believe our techniques can greatly enhance the level of OS security without compromising the performance and reliability of the OS.

1 Introduction

Kernel-level exploitation is becoming increasingly popular among attackers, with local and remote exploits surfacing for Windows [5], Linux [2], Mac OS X [3], BSD

variants [37, 4], and embedded operating systems [25]. This emerging trend stems from a number of important factors. First, the deployment of defense mechanisms for user programs has made application-level exploitation more challenging. Second, the kernel codebase is complex, large, and in continuous evolution, with many new vulnerabilities inevitably introduced over time. Studies on the Linux kernel have shown that its codebase has more than doubled with a steady fault rate over the past 10 years [55] and that many known but potentially critical bugs are at times left unpatched indefinitely [29]. Third, the number of targets in large-scale attacks is significant, with a plethora of internet-connected machines running the same kernel version independently of the particular applications deployed. Finally, an attacker has generally more opportunities inside the OS, for example the ability to disable in-kernel defense mechanisms or the option to execute shellcode at the user level (similar to classic application-level attacks) or at the kernel level (approach taken by kernel *rootkits*).

Unfortunately, existing OS-level countermeasures fail to provide a comprehensive defense mechanism against generic memory error exploits. A number of techniques aim to thwart code injection attacks [65, 28, 60], but are alone insufficient to prevent *return-into-kernel-text* attacks [56] and *return-oriented programming* (ROP) in general [35]. Other approaches protect kernel hooks or generally aim at preserving control-flow integrity [69, 74, 44, 57]. Unfortunately, this does not prevent attackers from tampering with noncontrol data, which may lead to privilege escalation or allow other attacks. In addition, most of these techniques incur high overhead and require virtualization support, thus increasing the size of the trusted computing base (TCB).

In this paper, we explore the benefits of address space randomization (ASR) inside the operating system and present the first comprehensive design to defend against classic and emerging OS-level attacks. ASR is a well-established defense mechanism to protect user programs

against memory error exploits [12, 39, 14, 72, 73]; all the major operating systems include some support for it at the application level [1, 68]. Unfortunately, the OS itself is typically not randomized at all. Recent Windows releases are of exception, as they at least randomize the base address of the text segment [56]. This randomization strategy, however, is wholly insufficient to counter many sophisticated classes of attacks (e.g., noncontrol data attacks) and is extremely vulnerable to information leakage, as better detailed later. To date, no strategy has been proposed for comprehensive and fine-grained OS-level ASR. Our effort lays the ground work to fill the gap between application-level ASR and ASR inside the OS, identifying the key requirements in the new context and proposing effective solutions to the challenges involved.

Contributions. The contributions of this paper are threefold. First, we identify the challenges and the key requirements for a comprehensive OS-level ASR solution. We show that a number of assumptions in existing solutions are no longer valid inside the OS, due to the more constrained environment and the different attack models. Second, we present the first design for fine-grained ASR for operating systems. Our approach addresses all the challenges considered and improves existing ASR solutions in terms of both performance and security, especially in light of emerging ROP-based attacks. In addition, we consider the application of our design to component-based OS architectures, presenting a fully fledged prototype system and discussing real-world applications of our ASR technique. Finally, we present the first generic *live rerandomization* strategy, particularly central in our design. Unlike existing techniques, our strategy is based on run-time state migration and can transparently rerandomize arbitrary code and data with no state loss. In addition, our rerandomization code runs completely sandboxed. Any run-time error at rerandomization time simply results in restoring normal execution without endangering the reliability of the OS.

2 Background

The goal of address space randomization is to ensure that code and data locations are unpredictable in memory, thus preventing attackers from making precise assumptions on the memory layout. To this end, fine-grained ASR implementations [14, 39, 72] permute the order of individual memory objects, making both their addresses and their relative positioning unpredictable. This strategy attempts to counter several classes of attacks.

Attacks on code pointers. The goal of these attacks is to override a function pointer or the return address on the stack with attacker-controlled data and subvert control flow. Common memory errors that can directly allow these attacks are buffer overflows, format bugs, use-

after-free, and uninitialized reads. In the first two cases, the attack requires assumptions on the relative distance between two memory objects (e.g., a vulnerable buffer and a target object) to locate the code pointer correctly. In the other cases, the attack requires assumptions on the relative alignment between two memory objects in case of memory reuse. For example, use-after-free attacks require control over the memory allocator to induce the allocation of an object in the same location of a freed object still pointed by a vulnerable dangling pointer. Similarly, attacks based on stack/heap uninitialized reads require predictable allocation strategies to reuse attacker-controlled data from a previously deallocated object. All these attacks also rely on the absolute location of the code the attacker wants to execute, in order to adjust the value of the code pointer correctly. In detail, code injection attacks rely on the location of attacker-injected shellcode. Attacks using *return-into-libc* strategies [22] rely on the location of a particular function—or multiple functions in case of chained *return-into-libc* attacks [52]. More generic attacks based on *return-oriented programming* [66] rely on the exact location of a number of *gadgets* statically extracted from the program binary.

Attacks on data pointers. These attacks commonly exploit one of the memory errors detailed above to override the value of a data pointer and perform an arbitrary memory read/write. Arbitrary memory reads are often used to steal sensitive data or information on the memory layout. Arbitrary memory writes can also be used to override particular memory locations and indirectly mount other attacks (e.g., control-flow attacks). Attacks on data pointers require the same assumptions detailed for code pointers, except the attacker needs to locate the address of some data (instead of code) in memory.

Attacks on nonpointer data. Attacks in this category target noncontrol data containing sensitive information (e.g., uid). These attacks can be induced by an arbitrary memory write or commonly originate from buffer overflows, format bugs, integer overflows, signedness bugs, and use-after-free memory errors. While unable to directly subvert control flow, they can often lead to privilege escalation or indirectly allow other classes of attacks. For example, an attacker may be able to perform an arbitrary memory write by corrupting an array index which is later used to store attacker-controlled data. In contrast to all the classes of attacks presented earlier, nonpointer data attacks only require assumptions on the relative distance or alignment between memory objects.

3 Challenges in OS-level ASR

This section investigates the key challenges in OS-level address space randomization, analyzing the differences with application-level ASR and reconsidering some of

the well-established assumptions in existing solutions. We consider the following key issues in our analysis.

W \oplus X. A number of ASR implementations complement their design with W \oplus X protection [68]. The idea is to prevent code injection attacks by ensuring that no memory page is ever writable and executable at the same time. Studies on the Linux kernel [45], however, have shown that enforcing the same property for kernel pages introduces implementation issues and potential sources of overhead. In addition, protecting kernel pages in a combined user/kernel address space design does not prevent an attacker from placing shellcode in an attacker-controlled application and redirecting execution there. Alternatively, the attacker may inject code into W \wedge X regions with double mappings that operating systems share with user programs (e.g., `vsyscall` page on Linux) [56].

Instrumentation. Fine-grained ASR techniques typically rely on code instrumentation to implement a comprehensive randomization strategy. For example, Bhaktar et al. [14] heavily instrument the program to create self-randomizing binaries that completely rearrange their memory layout at load time. While complex instrumentation strategies have been proven practical for application-level solutions, their applicability to OS-level ASR raises a number of important concerns. First, heavyweight instrumentation may introduce significant run-time overhead which is ill-affordable for the OS. Second, these load-time ASR strategies are hardly sustainable, given the limited operations they would be able to perform and the delay they would introduce in the boot process. Finally, complex instrumentation may introduce a lot of untrusted code executed with no restriction at runtime, thus endangering the reliability of the OS or even opening up new opportunities for attack.

Run-time constraints. There are a number of constraints that significantly affect the design of an OS-level ASR solution. First, making strong assumptions on the memory layout at load time simplifies the boot process. This means that some parts of the operating system may be particularly hard to randomize. In addition, existing rerandomization techniques are unsuitable for operating systems. They all assume a stateless model in which a program can gracefully exit and restart with a fresh rerandomized layout. Loss of critical state is not an option for an OS and neither is a full reboot, which introduces unacceptable downtime and loss of all the running processes. Luckily, similar restrictions also apply to an adversary determined to attack the system. Unlike application-level attacks, an exploit needs to explicitly recover any critical memory object corrupted during the attack or the system will immediately crash after successful exploitation.

Attack model. Kernel-level exploitation allows for a powerful attack model. Both remote and local attacks are possible, although local attacks mounted from a compro-

mised or attacker-controlled application are more common. In addition, many known attack strategies become significantly more effective inside the OS. For example, noncontrol data attacks are more appealing given the amount of sensitive data available. In addition, ROP-based control-flow attacks can benefit from the large codebase and easily find all the necessary gadgets to perform arbitrary computations, as demonstrated in [35]. This means that disclosing information on the locations of “useful” text fragments can drastically increase the odds of successful ROP-based attacks. Finally, the particular context opens up more attack opportunities than those detailed in Section 2. First, unchecked pointer dereferences with user-provided data—a common vulnerability in kernel development [18]—can become a vector of arbitrary kernel memory reads/writes with no assumption on the location of the original pointer. Second, the combined user/kernel address space design used in most operating systems may allow an attacker controlling a user program to directly leverage known application code or data for the attack. The conclusion is that making both the relative positioning between *any* two memory objects and the location of individual objects unpredictable becomes much more critical inside the OS.

Information leakage. Prior work on ASR has often dismissed information leakage attacks—in which the attacker is able to acquire information about the internal memory layout and carry out an exploit in spite of ASR—as relatively rare for user applications [14, 67, 72]. Unfortunately, the situation is completely different inside the OS. First, there are several possible entry points and a larger leakage surface than user applications. For instance, a recent study has shown that uninitialized data leading to information leakage is the most common vulnerability in the Linux kernel [18]. In addition, the common combined user/kernel address space design allows arbitrary memory writes to easily become a vector of information leakage for attacker-controlled applications. To make things worse, modern operating systems often disclose sensitive information to unprivileged applications voluntarily, in an attempt to simplify deployment and debugging. An example is the `/proc` file system, which has already been used in several attacks that exploit the exposed information in conventional [56] and nonconventional [76] ways. For instance, the `/proc` implementation on Linux discloses details on kernel symbols (i.e., `/proc/kallsyms`) and slab-level memory information (i.e., `/proc/slabinfo`). To compensate for the greater chances of information leakage, ASR at the finest level of granularity possible and continuous rerandomization become both crucial to minimize the knowledge acquired by an attacker while probing the system.

Brute forcing. Prior work has shown that many existing application-level ASR solutions are vulnerable to

simple brute-force attacks due to the low randomization entropy of shared libraries [67]. The attack presented in [67] exploits the crash recovery capabilities of the Apache web server and simply reissues the same return-into-libc attack with a newly guessed address after every crash. Unlike many long-running user applications, crash recovery cannot be normally taken for granted inside the OS. An OS crash is normally fatal and immediately hinders the attack while prompting the attention of the system administrator. Even assuming some crash recovery mechanism inside the OS [43, 27], brute-force attacks need to be far less aggressive to remain unnoticed. In addition, compared to remote clients hiding their identity and mounting a brute-force attack against a server application, the source of an OS crash can be usually tracked down. In this context, blacklisting the offensive endpoint/request becomes a realistic option.

4 A design for OS-level ASR

Our fine-grained ASR design requires confining different OS subsystems into isolated event-driven components. This strategy is advantageous for a number of reasons. First, this enables selective randomization and rerandomization for individual subsystems. This is important to fully control the randomization and rerandomization process with per-component ASR policies. For example, it should be possible to retune the rerandomization frequency of *only* the virtual filesystem after noticing a performance impact under particular workloads. Second, the event-driven nature of the OS components greatly simplifies synchronization and state management at rerandomization time. Finally, direct intercomponent control transfer can be more easily prevented, thus limiting the freedom of a control-flow attack and reducing the number of potential ROP gadgets by design.

Our ASR design is currently implemented by a microkernel-based OS architecture running on top of the MINIX 3 microkernel [32]. The OS components are confined in independent hardware-isolated processes. Hardware isolation is beneficial to overcome the problems of a combined user/kernel address space design introduced earlier and limit the options of an attacker. In addition, the MMU-based protection can be used to completely sandbox the execution of the untrusted rerandomization code. Our ASR design, however, is not bound to its current implementation and has more general applicability.

For example, our ASR design can be directly applied to other component-based OS architectures, including microkernel-based architectures used in common embedded OSes—such as L4 [41], Green Hills Integrity [7], and QNX [33]—and research operating systems using software-based component isolation schemes—such as Singularity [36]. Commodity operating systems, in con-

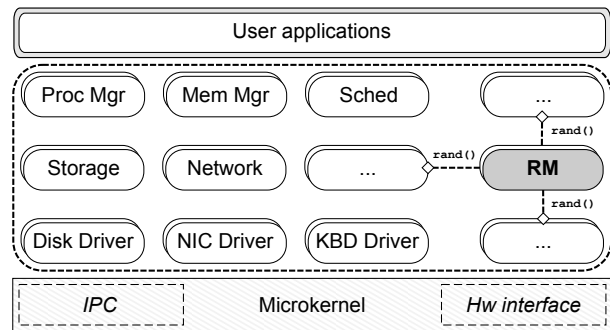


Figure 1: The OS architecture for our ASR design.

trast, are traditionally based on monolithic architectures and lack well-defined component boundaries. While this does not prevent adoption of our randomization technique, it does eliminate the ability to selectively rerandomize specific parts of the OS, yielding poorer flexibility and longer rerandomization times to perform whole-OS state migration. Encouragingly, there is an emerging trend towards allowing important commodity OS subsystems to run as isolated user-space processes, including filesystems [6] and user-mode drivers in Windows [50] or Linux [16]. Our end-to-end design can be used to protect all these subsystems as well as other operating system services from several classes of attacks. Note that, while running in user space, operating system services are typically trusted by the kernel and allowed to perform a variety of critical system operations. An example is `udev`, the device manager for the Linux kernel, which has already been target of several different exploits [17]. Finally, given the appropriate run-time support, our design could also be used to improve existing application-level ASR techniques and offer better protection against memory error exploits for generic user-space programs.

Figure 1 shows the OS architecture implementing our ASR design. At the heart lies the microkernel, providing only IPC functionalities and low-level resource management. All the other core subsystems are confined into isolated OS processes, including drivers, memory management, process management, scheduling, storage and network stack. In our design, all the OS processes (and the microkernel) are randomized using a link-time transformation implemented with the LLVM compiler framework [42]. The transformation operates on prelinked LLVM bitcode to avoid any lengthy recompilation process at runtime. Our link-time strategy avoids the need for fine-grained load-time ASR, eliminating delays in the boot process and the run-time overhead introduced by the indirection mechanisms adopted [14]. In addition, this strategy reduces the instrumentation complexity to the bare minimum, with negligible amount of untrusted code exposed to the runtime. The vast majority of our ASR

transformations are statically verified by LLVM at the bitcode level. As a result, our approach is also safer than prior ASR solutions relying on binary rewriting [39].

As pointed out in [14], load-time ASR has a clear advantage over alternative strategies: the ability to create self-randomizing binaries distributed to every user in identical copies, thus preserving today’s software distribution model. Fortunately, our novel live rerandomization strategy can fully address this concern. In our model, every user receives the same (unrandomized) binary version of the OS, as well as the prelinked LLVM bitcode of each OS component. The bitcode files are stored in a protected disk partition inaccessible to regular user programs, where a background process periodically creates new randomized variants of the OS components using our link-time ASR transformation (and any valid LLVM backend to generate the final binary). The generated variants are consumed by the *randomization manager* (RM), a special component that periodically rerandomizes every OS process (including itself). Unlike all the existing solutions, rerandomization is applied transparently online, with no system reboot or downtime required. The conclusion is that we can directly leverage our live rerandomization technique to randomize the original OS binary distributed to the user. This strategy retains the advantages of link-time ASR without affecting the software distribution model.

When the OS boots up for the first time, a full rerandomization round is performed to relinquish any unrandomized code and data present in the original binary. To avoid slowing down the first boot process, an option is to perform the rerandomization lazily, for example replacing one OS process at the time at regular time intervals. After the first round, we continuously perform live rerandomization of individual OS components in the background. Currently, the microkernel is the only piece of the OS that does not support live rerandomization. Rerandomization can only be performed after a full reboot, with a different variant loaded every time. While it is possible to extend our current implementation to support live rerandomization for the microkernel, we believe this should be hardly a concern. Microkernel implementations are typically in the order of 10kLOC, a vastly smaller TCB than most hypervisors used for security enforcement, as well as a candidate for formal verification, as demonstrated in prior work [40].

Our live rerandomization strategy for an OS process, in turn, is based on run-time state migration, with the entire execution state transparently transferred to the new randomized process variant. The untrusted rerandomization code runs completely sandboxed in the new variant and, in case of run-time errors, the old variant immediately resumes execution with no disruption of service or state loss. To support live migration, we rely on another

LLVM link-time transformation to embed relocation and type information into the final process binary. This information is exposed to the runtime to accurately introspect the state of the two variants and migrate all the randomized memory objects in a layout-independent way.

5 ASR transformations

The goal of our link-time ASR transformation is to randomize all the code and data for every OS component. Our link-time strategy minimizes the time to produce new randomized OS variants on the deployment platform and automatically provides randomization for the program and all the statically linked libraries. Our transformation design is based on five key principles: (i) minimal performance impact; (ii) minimal amount of untrusted code exposed to the runtime; (iii) architecture-independence; (iv) no restriction on compiler optimizations; (v) maximum randomization granularity possible. The first two principles are particularly critical for the OS, as discussed earlier. Architecture-independence enhances portability and eliminates the need for complex binary rewriting techniques. The fourth principle dictates compiler-friendly strategies, for example avoiding indirection mechanisms used in prior solutions [12], which inhibit a number of standard optimizations (e.g., inlining). Eliminating the need for indirection mechanisms is also important for debuggability reasons. Our transformations are all debug-friendly, as they do not significantly change the code representation—only allocation sites are transformed to support live rerandomization, as detailed later—and preserve the consistency of symbol table and stack information. Finally, the last principle is crucial to provide lower predictability and better security than existing techniques.

Traditional ASR techniques [1, 68, 12] focus on randomizing the base address of code and data regions. This strategy is ineffective against all the attacks that make assumptions only about relative distances/alignments between memory objects, is prone to brute forcing [67], and is extremely vulnerable to information leakage. For instance, many examples of application-level information leakage have emerged on Linux over the years, and experience shows that, even by acquiring minimal knowledge on the memory layout, an attacker can completely bypass these basic ASR techniques [24].

To overcome these limitations, second-generation ASR techniques [14, 39, 72] propose fine-grained strategies to permute individual memory objects and randomize their relative distances/alignments. While certainly an improvement over prior techniques, these strategies are still vulnerable to information leakage, raising serious concerns on their applicability at the OS level. Unlike traditional ASR techniques, these strategies make it

normally impossible for an attacker to make strong assumptions on the locations of arbitrary memory objects after learning the location of a single object. They are completely ineffective, however, in inhibiting precise assumptions on the layout of the leaked object itself. This is a serious concern inside the OS, where information leakage is the norm rather than the exception.

To address all the challenges presented, our ASR transformation is implemented by an LLVM link-time pass which supports fine-grained randomization of both the relative distance/alignment between *any* two memory objects and the *internal layout* of individual memory objects. We now present our transformations in detail and draw comparisons with state-of-the-art techniques.

Code randomization. The code-transformation pass performs three primary tasks. First, it enforces a random permutation of all the program functions. In LLVM, this is possible by shuffling the symbol table in the intended order and setting the appropriate linkage to preserve the permutation at code generation time. Second, it introduces (configurable) random-sized padding before the first function and between any two functions in the bitcode, making the layout even more unpredictable. To generate the padding, we create dummy functions with a random number of instructions and add them to the symbol table in the intended position. Thanks to demand paging, even very large padding sizes do not significantly increase the run-time physical memory usage. Finally, unlike existing ASR solutions, we randomize the internal layout of every function.

To randomize the function layout, an option is to permute the basic blocks and the instructions in the function. This strategy, however, would hinder important compiler optimizations like branch alignment [75] and optimal instruction scheduling [49]. Nonoptimal placements can result in poor instruction cache utilization and inadequate instruction pipelining, potentially introducing significant run-time overhead. To address this challenge, our pass performs *basic block shifting*, injecting a dummy basic block with a random number of instructions at the top of every function. The block is never executed at runtime and simply skipped over, at the cost of only one additional jump instruction. Note that the order of the original instructions and basic blocks is left untouched, with no noticeable impact on run-time performance. The offset of every instruction with respect to the address of the function entry point is, however, no longer predictable.

This strategy is crucial to limit the power of an attacker in face of information leakage. Suppose the attacker acquires knowledge on the absolute location of a number of kernel functions (e.g., using `/proc/kallsyms`). While return-into-kernel-text attacks for these functions are still conceivable (assuming the attacker can subvert control flow), arbitrary ROP-based computations are structurally

prevented, since the location of individual gadgets is no longer predictable. While the dummy basic block is in a predictable location, it is sufficient to cherry-pick its instructions to avoid giving rise to any new useful gadget. It is easy to show that a sequence of nop instructions does not yield any useful gadget on the x86 [54], but other strategies may be necessary on other architectures.

Static data randomization. The data-transformation pass randomly permutes all the static variables and read-only data on the symbol table, as done before for functions. We also employ the same padding strategy, except random-sized dummy variables are used for the padding. Buffer variables are also separated from other variables to limit the power of buffer overflows. In addition, unlike existing ASR solutions, we randomize the internal layout of static data, when possible.

All the aggregate types in the C programming language are potential candidates for layout randomization. In practice, there are a number of restrictions. First, the order of the elements in an array cannot be easily randomized without changing large portions of the code and resorting to complex program analysis techniques that would still fail in the general case. Even when possible, the transformation would require indirection tables that translate many sequential accesses into random array accesses, sensibly changing the run-time cache behavior and introducing overhead. Second, unions are currently not supported natively by LLVM and randomizing their layout would introduce unnecessary complications, given their rare occurrence in critical system data structures and their inherent ambiguity that already weakens the assumptions made by an attacker. Finally, packed structs cannot be randomized, since the code makes explicit assumptions on their internal layout.

In light of these observations, our transformation focuses on randomizing the layout of regular struct types, which are pervasively used in critical system data structures. The layout randomization permutes the order of the struct members and adds random-sized padding between them. To support all the low-level programming idioms allowed by C, the type transformations are operated uniformly for all the static and dynamic objects of the same struct type. To deal with code which treats nonpacked structs as implicit unions through pointer casting, our transformation pass can be instructed to detect unsafe pointer accesses and refrain from randomizing the corresponding struct types.

Layout randomization of system data structures is important for two reasons. First, it makes the relative distance/alignment between two struct members unpredictable. For example, an overflow in a buffer allocated inside a struct cannot make precise assumptions about which other members will be corrupted by the overflow. Second, this strategy is crucial to limit the assumptions

of an attacker in face of information leakage. Suppose an attacker is armed with a reliable arbitrary kernel memory write generated by a missing pointer check. If the attacker acquires knowledge on the location of the data structure holding user credentials (e.g., `struct cred` on Linux) for an attacker-controlled unprivileged process, the offset of the `uid` member is normally sufficient to surgically override the user ID and escalate privileges. All the existing ASR solutions fail to thwart this attack. In contrast, our layout randomization hinders any precise assumptions on the final location of the `uid`. While brute forcing is still possible, this strategy will likely compromise other data structures and trigger a system crash.

Stack randomization. The stack randomization pass performs two primary tasks. First, it randomizes the base address of the stack to make the absolute location of any stack object unpredictable. In LLVM, this can be accomplished by creating a dummy `alloca` instruction—which allocates memory on the stack frame of the currently executing function—at the beginning of the program, which is later expanded by the code generator. This strategy provides a portable and efficient mechanism to introduce random-sized padding for the initial stack placement. Second, the pass randomizes the relative distance/alignment between any two objects allocated on the stack. Prior ASR solutions have either ignored this issue [39, 72] or relied on a shadow stack and dynamically generated random padding [14], which introduces high run-time overhead (10% in the worst case in their experiments for user applications).

To overcome these limitations, our approach is completely static, resulting in good performance and code which is statically verified by LLVM. In addition, this strategy makes it realistic to use cryptographically random number generators (e.g., `/dev/random`) instead of pseudo-random generators to generate the padding. While care should be taken not to exhaust the randomness pool used by other user programs, this approach yields much stronger security guarantees than pseudo-random generators, like recent attacks on ASR demonstrate [24]. Our transformations can be configured to use cryptographically random number generators for code, data, and stack instrumentation, while, similar to prior approaches [14], we always resort to pseudo-random generation in the other cases for efficiency reasons.

When adopting a static stack padding strategy, great care should be taken not to degrade the quality of the randomization and the resulting security guarantees. To randomize the relative distances between the objects in a stack frame, we permute all the `alloca` instructions used to allocate local variables (and function parameters). The layout of every stack-allocated `struct` is also randomized as described earlier. Nonbuffer variables are all grouped and pushed to the top of the frame, close

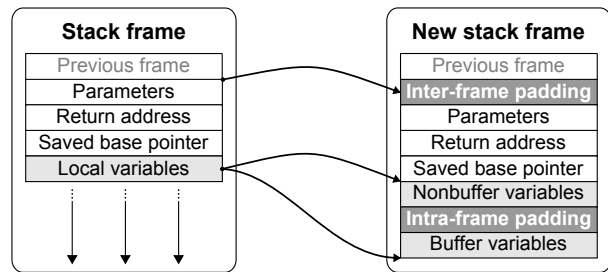


Figure 2: The transformed stack layout.

to the base pointer and the return address. Buffer variables, in turn, are pushed to the bottom, with randomized padding (i.e., dummy `alloca` instructions) added before and between them. This strategy matches our requirements while allowing the code generator to emit a maximally efficient function prologue.

To randomize the relative alignment between any two stack frame allocations of the same function (and thus the relative alignment between their objects), we create random-sized padding before every function call. Albeit static, this strategy faithfully emulates dynamically generated padding, given the level of unpredictability introduced across different function calls. Function calls inside loops are an exception and need to be handled separately. Loop unrolling is a possible solution, but enforcing this optimization in the general case may be expensive. Our approach is instead to precompute N random numbers for each loop, and cycle through them before each function call. Figure 2 shows the randomized stack layout generated by our transformation.

Dynamic data randomization. Our operating system provides `mmap`/`mmap`-like abstractions to every OS process. Ideally, we would like to create memory allocation wrappers to accomplish the following tasks for both heap and memory-mapped regions: (i) add random-sized padding before the first allocated object; (ii) add random-sized padding between objects; (iii) permute the order of the objects. For memory-mapped regions, all these strategies are possible and can be implemented efficiently [39]. We simply need to intercept all the new allocations and randomly place them in any available location in the address space. The only restriction is for fixed OS component-specific virtual memory mappings, which cannot be randomized and need to be explicitly reserved at initialization time.

For heap allocations, we instrument the code to randomize the heap base address and introduce randomized padding at allocation time. Permuting heap objects, however, is normally impractical in standard allocation schemes. While other schemes are possible—for example, the slab allocator in our memory manager randomizes block allocations within a slab page—state-of-

the-art allocators that enforce a fully and globally randomized heap organization incur high overhead (117% worst-case performance penalty) [53]. This limitation is particularly unfortunate for kernel *Heap Feng Shui* attacks [25], which aim to carefully drive the allocator into a deterministic exploitation-friendly state. While random interobject padding makes these attacks more difficult, it is possible for an attacker to rely on more aggressive exploitation strategies (i.e., heap spraying [59]) in this context. Suppose an attacker can drive the allocator into a state with a very large unallocated gap followed by only two allocated buffers, with the latter vulnerable to underflow. Despite the padding, the attacker can induce a large underflow to override all the traversed memory locations with the same target value. Unlike stack-based overflows, this strategy could lead to successful exploitation without the attacker worrying about corrupting other critical data structures and crashing the system. Unlike prior ASR solutions, however, our design can mitigate these attacks by periodically rerandomizing every OS process and enforcing a new unpredictable heap permutation. We also randomize (and rerandomize) the layout of all the dynamically allocated structs, as discussed earlier.

Kernel modules randomization. Traditional loadable kernel module designs share many similarities—and drawbacks, from a security standpoint—with application-level shared libraries. The attack presented in [61] shows that the data structures used for dynamic linking are a major source of information leakage and can be easily exploited to bypass any form of randomization for shared libraries. Prior work on ASR [67, 14] discusses the difficulties of reconciling sharing with fine-grained randomization. Unfortunately, the inability to perform fine-grained randomization on shared libraries opens up opportunities for attacks, including probing, brute forcing [67], and partial pointer overwrites [23].

To overcome these limitations, our design allows only statically linked libraries for OS components and inhibits any form of dynamic linking inside the operating system. Note that this requirement does by no means limit the use of loadable modules, which our design simply isolates in independent OS processes following the same distribution and deployment model of the core operating system. This approach enables sharing and lazy loading/unloading of individual modules with no restriction, while allowing our rerandomization strategy to randomize (and rerandomize) every module in a fine-grained manner. In addition, the process-based isolation prevents direct control-flow and data-flow transfer between a particular module and the rest of the OS (i.e., the access is always IPC- or capability-mediated). Finally, this strategy can be used to limit the power of untrusted loadable kernel modules, an idea also explored in prior work on commodity operating systems [16].

6 Live rerandomization

Our live rerandomization design is based on novel automated run-time migration of the execution state between two OS process variants. The variants share the same operational semantics but have arbitrarily different memory layouts. To migrate the state from one variant to the other at runtime, we need a way to remap all the corresponding global state objects. Our approach is to transform the bitcode with another LLVM link-time pass, which embeds metadata information into the binary and makes run-time state introspection and automated migration possible.

Metadata transformation. The goal of our pass is to record metadata describing all the static state objects in the program and instrument the code to create metadata for dynamic state objects at runtime. Access to these objects at the bitcode level is granted by the LLVM API. In particular, the pass creates static metadata nodes for all the static variables, read-only data, and functions whose address is taken. Each metadata node contains three key pieces of information: node ID, relocation information, and type. The node ID provides a layout-independent mechanism to map corresponding metadata nodes across different variants. This is necessary because we randomize the order and the location of the metadata nodes (and write-protect them) to hinder new opportunities for attacks. The relocation information, in turn, is used by our run-time migration component to locate every state object in a particular variant correctly. Finally, the type is used to introspect any given state object and migrate the contained elements (e.g., pointers) correctly at runtime.

To create a metadata node for every dynamic state object, our pass instruments all the memory allocation and deallocation function calls. The node is stored before the allocated data, with canaries to protect the in-band metadata against buffer overflows. All the dynamic metadata nodes are stored in a singly-linked list, with each node containing relocation information, allocation flags, and a pointer to an allocation descriptor. Allocation flags define the nature of a particular allocation (e.g., heap) to reallocate memory in the new variant correctly at migration time. The allocation descriptors, in turn, are statically created by the pass for all the allocation sites in the program. A descriptor contains a site ID and a type. Similar to the node ID, the site ID provides a layout-independent mechanism to map corresponding allocation descriptors (also randomized and write-protected) across different variants. The type, in contrast, is determined via static analysis and used to correctly identify the runtime type of the allocated object (e.g., a `char` type with an allocation of 7 bytes results in a `[7 x char]` runtime type). Our static analysis can automatically identify the type for all the standard memory allocators and custom allocators that use simple allocation wrappers. More

advanced custom allocation schemes, e.g., region-based memory allocators [11], require instructing the pass to locate the proper allocation wrappers correctly.

The rerandomization process. Our OS processes follow a typical event-driven model based on message passing. At startup, each process initializes its state and immediately jumps to the top of a long-running event-processing loop, waiting for IPC messages to serve. Each message can be processed in cooperation with other OS processes or the microkernel. The message dispatcher, isolated in a static library linked to every OS process, can transparently intercept two special system messages sent by the *randomization manager* (RM): *sync* and *init*. These messages cannot be spoofed by other processes because the IPC is mediated by the microkernel.

The rerandomization process starts with RM loading a new variant in memory, in cooperation with the microkernel. Subsequently, it sends a sync message to the designated OS process, which causes the current variant to immediately block in a well-defined execution point. A carefully selected synchronization point (e.g., in *main*) eliminates the need to instrument transient stack regions to migrate additional state, thus reducing the run-time overhead and simplifying the rerandomization strategy. The new variant is then allowed to run and delivered an *init* message with detailed instructions. The purpose of the *init* message is to discriminate between fresh start and rerandomization *init*. In the latter scenario, the message contains a capability created by the microkernel, allowing the new variant to read arbitrary data and metadata from the old variant. The capability is attached to the IPC endpoint of the designated OS process and can thus only be consumed by the new variant, which by design inherits the old variant's endpoint. This is crucial to transparently rerandomize individual operating system processes without exposing the change to the rest of the system.

When the rerandomization *init* message is intercepted, the message dispatcher requests the run-time migration component to initialize the new variant properly and then jumps to the top of the event-processing loop to resume execution. This preserves the original control flow semantics and transparently restores the correct execution state. The migration component is isolated in a library and runs completely sandboxed in the new variant. RM monitors the execution for run-time errors (i.e., panics, crashes, timeouts). When an error is detected, the new variant is immediately cleaned up, while the old variant is given back control to resume execution normally. When the migration completes correctly, in contrast, the old variant is cleaned up, while the new variant resumes execution with a rerandomized memory layout. We have also implemented rerandomization for RM itself, which only required some extra microkernel changes to detect run-time errors and arbitrate control transfer between the

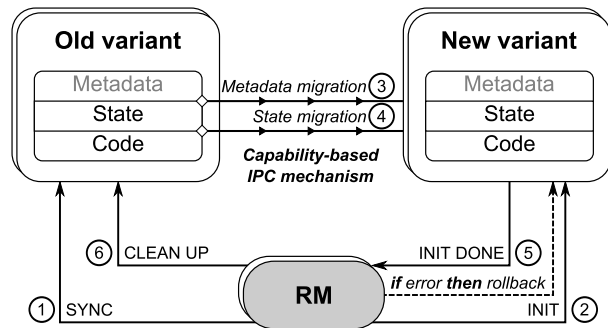


Figure 3: The rerandomization process.

two variants. Our run-time error detection mechanism allows for safe rerandomization without trusting the (complex) migration code. Moreover, the reversibility of the rerandomization process makes detecting semantic errors in the migration code a viable option. For example, one could transparently migrate the state from one variant to another, migrate it again to another instance of the original variant, and then compare the results. Figure 3 depicts the proposed rerandomization process.

State migration. The migration starts by transferring all the metadata from the old variant to a local cache in the new variant. Our capability-based design allows the migration code to locate a root metadata descriptor in the old variant and recursively copy all the metadata nodes and allocation descriptors to the new variant. To automate the metadata transfer, all the data structures copied use a fixed and predetermined layout. At the end, both the old and the new metadata are available locally, allowing the code to arbitrarily introspect the state of the two variants correctly. To automate the data transfer, we map every old metadata node in the local cache with its counterpart in the new variant. This is done by pairing nodes by ID and carefully reallocating every old dynamic state object in the new variant. Reallocations are performed in random order, thus enforcing a new unpredictable permutation of heap and memory-mapped regions. An interesting side effect of the reallocation process is the compaction of all the live heap objects, an operation that reduces heap fragmentation over time. Our reallocation strategy is indeed inspired by the way a compacting garbage collector operates [70].

The mapping phase generates all the perfect pairs of state objects in the two variants, ready for data migration. Note that paired state objects may not reflect the same type or size, due to the internal layout rerandomization. To transfer the data, the migration code introspects every state object in the old variant by walking its type recursively and examining each inner state element found. Nonpointer elements are simply transferred by value, while pointer elements require a more careful transfer strategy. To deal with layout randomization,

each recursive step requires mapping the current state element to its counterpart (and location) in the new variant. This can be easily accomplished because the old type and the new type have isomorphic structures and only differ in terms of member offsets for randomized struct types. For example, to transfer a struct variable with 3 primitive members, the migration code walks the original struct type to locate all the members, computes their offsets in the two variants, and recursively transfers the corresponding data in the correct location.

Pointer migration. The C programming language allows several programming constructs that make pointer migration particularly challenging in the general case. Our approach is to fully automate migration of all the common cases and only delegate the undecidable cases to the programmer. The first case to consider is a pointer to a valid static or dynamic state object. When the pointer points to the beginning of the object, we simply reinitialize the pointer with the address of the pointed object in the new variant. Interior pointers (i.e., pointers into the middle of an object) in face of internal layout rerandomization require a more sophisticated strategy. Similar to our introspection strategy, we walk the type of the pointed object and recursively remap the offset of the target element to its counterpart. This strategy is resilient to arbitrary layout rerandomization and makes it easy to reinitialize the pointer in the new variant correctly.

Another scenario of interest is a pointer which is assigned a special integer value (e.g., NULL or MAP_FAILED (-1)). Our migration code can explicitly recognize special ranges and transfer the corresponding pointers by value. Currently, all the addresses in reserved memory ranges (e.g., zero pages) are marked as special values.

In another direction, memory addresses or other layout-specific information may be occasionally stored in integer variables. This is, unfortunately, a case of unsolvable ambiguity which cannot be automatically settled without programmer assistance. To this end, we support annotations to mark “hidden” pointers in the code.

Pointers stored in unions are another case of unsolvable ambiguity. Since C does not support tagged unions, it is impossible to resolve these cases automatically. In our experiments with OS code, unions with pointers were the only case of ambiguity that required manual intervention. Other cases are, however, possible. For example, any form of pointer encoding or obfuscation [13] would require knowledge on the particular encoding to migrate pointers correctly. Other classes of pointers—guard pointers, uninitialized pointers, dangling pointers—are instead automatically handled in our implementation. In the last two cases, the general strategy is to try to transfer the pointer as a regular pointer, and simply reinitialize it to NULL in the new variant whenever our dynamic pointer analysis reports an error.

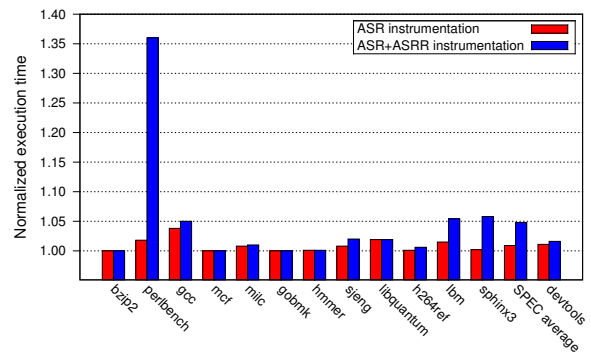


Figure 4: Execution time of the SPEC CPU 2600 benchmarks and our *devtools* benchmark normalized against the baseline (no OS/benchmark instrumentation).

7 Evaluation

We have implemented our ASR design on the MINIX 3 microkernel-based operating system [32], which already guarantees process-based isolation for all the core operating system components. The OS is x86-based and exposes a complete POSIX interface to user applications. We have heavily modified and redesigned the original OS to implement support for our ASR techniques for all the possible OS processes. The resulting operating system comprises a total of 20 OS processes (7 drivers and 13 servers), including process management, memory management, storage and network stack. Subsequently, we have applied our ASR transformations to the system and evaluated the resulting solution.

7.1 Performance

To evaluate the performance of our ASR technique, we ported the C programs in the SPEC CPU 2006 benchmark suite to our prototype system. We also put together a *devtools* macrobenchmark, which emulates a typical syscall-intensive workload with the following operations performed on the OS source tree: compilation, find, grep, copying, and deleting. We performed repeated experiments on a workstation equipped with a 12-core 1.9Ghz AMD Opteron “Magny-Cours” processor and 4GB of RAM, and averaged the results. All the OS code and our benchmarks were compiled using Clang/LLVM 2.8 with -O2 optimization level. To thoroughly stress the system and identify all the possible bottlenecks, we instrumented *both* the OS and the benchmarks using the same transformation in each run. The default padding strategy used in the experiments extends the memory occupancy of every state object or struct member by 0-30%, similar to the default values suggested in [14]. Figure 4 depicts the resulting execution times.

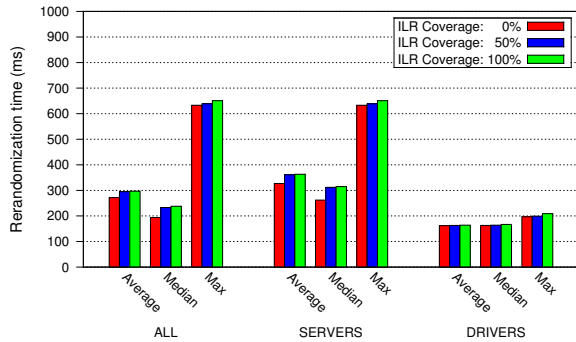


Figure 5: Rerandomization time against coverage of internal layout rerandomization.

The ASR instrumentation alone introduces 0.9% run-time overhead on average on SPEC benchmarks and 1.1% on *devtools*. The average run-time overhead increases to 4.8% and 1.6% respectively with ASRR instrumentation. The maximum overhead reported across all the benchmarks was found for *perlbench* (36% ASRR overhead). Profiling revealed this was caused by a massive amount of dynamic memory allocations. This test case pinpoints a potential source of overhead introduced by our technique, which, similar to prior approaches, relies on memory allocation wrappers to instrument dynamically allocated objects. Unlike prior comprehensive solutions, however, our run-time overhead is barely noticeable on average (1.9% for ASRR without *perlbench*). The most comprehensive second-generation technique presented in [14]—which, compared to other techniques, also provides fine-grained stack randomization—introduces a run-time overhead of 11% on average and 23% in the worst case, even by instrumenting *only* the test programs. The main reasons for the much higher overheads are the use of heavyweight stack instrumentation and indirection mechanisms that inhibit compiler optimizations and introduce additional pointer dereferences for every access to code and data objects. Their stack instrumentation, however, includes a shadow stack implementation that could complement our techniques to offer stronger protection against stack spraying attacks.

Although we have not observed strong variations in our macrobenchmark performance across different runs, our randomization technique can potentially affect the original spatial locality and yield nonoptimal cache usage at runtime. The possible performance impact introduced—inherent in all the fine-grained ASR techniques—is subject to the particular compiler and system adopted and should be carefully evaluated in each particular deployment scenario.

Figure 5 shows the rerandomization time (average,

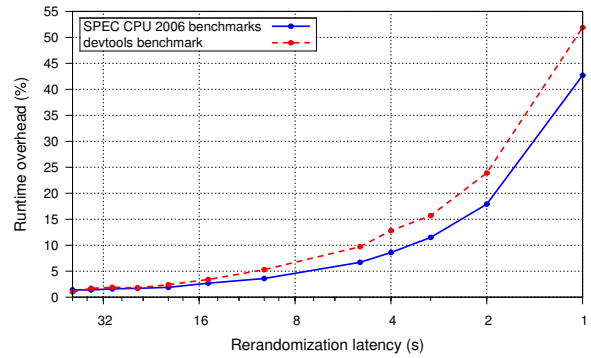


Figure 6: Run-time overhead against periodic rerandomization latency.

median, max) measured across all the OS components. With no internal layout rerandomization (ILR), a generic component completes the rerandomization process in 272ms on average. A higher ILR coverage increases the average rerandomization time only slightly (297ms at 100% coverage). The impact is more noticeable for OS servers than drivers, due to the higher concentration of complex rerandomized structs (and pointers to them) that need to be remapped during migration. Albeit satisfactory, we believe these times can be further reduced, for example using indexes to speed up our dynamic pointer analysis. Unfortunately, we cannot compare our current results against existing solutions, given that no other live rerandomization strategy exists to date.

Finally, Figure 6 shows the impact of periodic rerandomization on the execution time of SPEC and *devtools*. The experiment was performed by rerandomizing a single OS component at the end of every predetermined time interval. To ensure uniform coverage, the OS components were all rerandomized in a round-robin fashion. Figure 6 reports a barely noticeable overhead for rerandomization latencies higher than 20s. For lower latencies, the overhead increases steadily, reaching the value of 42.7% for SPEC and 51.9% for *devtools* at 1s. The rerandomization latency defines a clear tradeoff between performance and unobservability of the system. Reasonable choices of the rerandomization latencies introduce no performance impact and leave a small window with a stable view of the system to the attacker. In some cases, a performance penalty may also be affordable to offer extra protection in face of particularly untrusted components.

7.2 Memory usage

Table 1 shows the average run-time virtual memory overhead introduced by our technique inside the OS during the execution of our benchmarks. The overhead measured is comparable to the space overhead we observed

Type	Overhead
ASRR state	16.1%
ASRR overall	14.6%
ASR padding _a	$((8a_s + 2a_h + 4a_f) \cdot 10^{-4} + c_{base})\%$
ASR padding _r	$((2r_s + 0.6r_h + 3r_f) \cdot 10^{-1} + c_{base})\%$

Table 1: Average run-time virtual memory overhead measured during the execution of our benchmarks.

for the OS binaries on the disk. In the table, we report the virtual memory overhead to also account for dynamic state object overhead at runtime. For the average OS component, support for rerandomization introduces 16.1% state overhead (the extra memory necessary to store state metadata w.r.t. the original memory occupancy of all the static and dynamic static objects) and 14.6% overall memory overhead (the extra memory necessary to store state metadata and migration code w.r.t. the original memory footprint) on average. The virtual memory overhead (not directly translated to physical memory overhead, as noted earlier) introduced by our randomization strategy is only due to padding. Table 1 reports the overhead for two padding schemes using byte granularity (but others are possible): (i) padding_a, generating an inter-object padding of a bytes, with a uniformly distributed in $[0; a_{s,h,f}]$ for static, heap, and function objects, respectively; (ii) padding_r, generating an inter-object padding of $r \cdot s$ bytes, with a preceding object of size s , and r uniformly distributed in $[0; r_{s,h,f}]$ for static, heap, and function objects, respectively. The coefficient c_{base} is the overhead introduced by the one-time padding used to randomize the base addresses. The formulations presented here omit stack frame padding, which does not introduce persistent memory overhead.

7.3 Effectiveness

As pointed out in [14], an analytical analysis is more general and effective than empirical evaluation in measuring the effectiveness of ASR. Bhaktar et al. [14] present an excellent analysis on the probability of exploitation for different vulnerability classes. Their entropy analysis applies also to other second-generation ASR techniques, and, similarly, to our technique, which, however, provides additional entropy thanks to internal layout randomization and live rerandomization. Their analysis, however, is mostly useful in evaluating the effectiveness of ASR techniques against guessing and brute-force attacks. As discussed earlier, these attacks are far less attractive inside the operating system. In contrast, information leakage dominates the scene.

For this reason, we explore another direction in our analysis, answering the question: “How much informa-

	ASR ₁	ASR ₂	ASR ₃
<i>Vulnerability</i>			
Buffer overflows	A_r	R_o	R_e
Format string bugs	A_r	R_o	R_e
Use-after-free	A_r	R_o	R_e
Uninitialized reads	A_r	R_o	R_e
<i>Effect</i>			
Arbitrary memory R W	A_r	A_o	A_e
Controlled code injection	A_r	A_o	A_e
Return-into-libc/text	A_r	$N \cdot A_o$	$N \cdot A_o$
Return-oriented programming	A_r	$N \cdot A_o$	-

A_r = Known region address
 A_o = Known object address
 A_e = Known element address
 R_o = Known relative distance/alignment between objects
 R_e = Known relative distance/alignment between elements

Table 2: Comparison of ASR techniques.

tion does the attacker need to acquire for successful exploitation?”. In this respect, Table 2 compares our ASR technique (ASR₃) with first-generation techniques like PaX [68] and comprehensive second-generation techniques like the one presented in [14]. Most attacks require at least some knowledge of a memory area to corrupt and of another target area to achieve the intended effect (missing kernel pointer checks and non control data attacks are examples of exceptions in the two cases). Table 2 shows that first-generation techniques only require the attacker to learn the address of a memory region (e.g., stack) to locate the target correctly. Second-generation techniques, in turn, allow the attacker to corrupt the right memory location by learning the relative distance/alignment between two memory objects.

In this respect, our internal layout randomization provides better protection, forcing the attacker to learn the relative distance/alignment between two memory elements in the general case. For example, if the attacker learns the relative alignment between two heap-allocated data structures S_1 and S_2 and wants to exploit a vulnerable dangling pointer to hijack a write intended for a member of S_1 to a member of S_2 , he still needs to acquire information on the relative positioning of the members.

Similarly, our technique constraints attacks based on arbitrary memory reads/writes to learn the address of the target element. In contrast, second-generation techniques only require knowledge of the target memory object. This is easier to acquire, because individual objects can be exposed by symbol information (e.g., `/proc/kallsyms`) and are generally more likely to have their address taken (and leaked) than interior elements. Controlled code injection shows similar differences—spraying attacks are normally “ A_r ”, in contrast. Return-

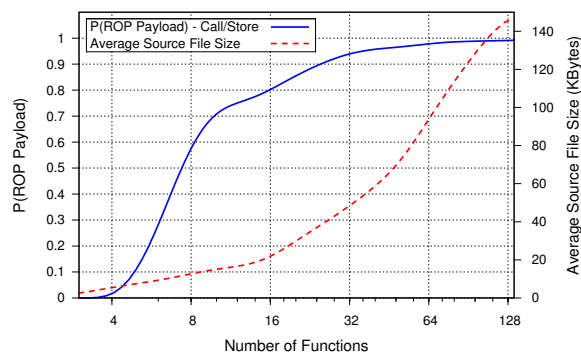


Figure 7: The probability that state-of-the-art techniques [64] can successfully generate ROP payloads to call linked functions or perform attacker-controlled arbitrary memory writes. The (fitted) distribution is plotted against the number of known functions in the program.

into-libc/text, in turn, requires the attacker to learn the location of N chosen functions in both cases, because our function layout randomization has no effect.

Things are different in more general ROP-based attacks. Our strategy completely hinders these attacks by making the location of the gadgets inside a function unpredictable. Given that individual gadgets cannot have their address taken and function pointer arithmetic is generally disallowed in a program, the location of a gadget cannot be explicitly leaked. This makes information leakage attacks ineffective in acquiring any useful knowledge for ROP-based exploitation. In contrast, prior techniques only require the attacker to learn the address of *any* N functions with useful gadgets to mount a successful ROP-based attack. To estimate N , we made an analysis on GNU coreutils (v7.4), building on the results presented in [64]. Figure 7 correlates the number of program functions with the probability of locating all the necessary ROP gadgets, and shows, for example, that learning 16 function addresses is sufficient to carry out an attack in more than 80% of the cases.

Another interesting question is: “*How fast can the attacker acquire the required information?*”. Our live rerandomization technique can periodically invalidate the knowledge acquired by an attacker probing the system (e.g., using an arbitrary kernel memory read). Shacham et al. [67] have shown that rerandomization slows down single-target probing attacks by only a factor of 2. As shown in Table 2, however, many attacks require knowledge of multiple targets when fine-grained ASR is in place. In addition, other attacks (e.g., Heap Feng Shui) may require multiple probing rounds to assert intermediate system states. When multiple rounds are required, the attacker is greatly limited by our rerandomization strategy because *any* knowledge acquired is

only useful in the current rerandomization window. In particular, let us assume the duration of every round to be distributed according to some probability distribution $p(t)$ (e.g., computed from the probabilities given in [14]). Hence, the time to complete an n -round probing phase is distributed according to the convolution of the individual $p_i(t)$. Assuming the same $p_i(t)$ in every round for simplicity, it can be shown that the expected time before the attacker can complete the probing phase in a single rerandomization window (and thus the attack) is:

$$T_{attack} = T \cdot \left(\int_0^T p^{*n}(\tau) d\tau \right)^{-1},$$

where T is the size (ms) of the rerandomization window, n is the number of probing rounds, and $p^{*n}(t)$ is the n -fold convolution power of $p(t)$. Since the convolution power decreases rapidly with the number of targets n , the attack can quickly become impractical. Given a vulnerability and an attack model characterized by some $p(t)$, this formula gives a practical way to evaluate the impact of a given rerandomization frequency on attack prevention. When a new vulnerability is discovered, this formula can also be used to retune the rerandomization frequency (perhaps accepting a performance penalty) and make the attack immediately impractical, even before an appropriate patch is developed and made available. This property suggests that our ASR design can also be used as the first “live-workaround” system for security vulnerabilities, similar, in spirit, to other systems that provide immediate workarounds to bypass races at runtime [71].

8 Related work

Randomization. Prior work on ASR focuses on randomizing the memory layout of user programs, with solutions based on kernel support [39, 1, 68], linker support [73], compiler-based techniques [12, 14, 72], and binary rewriting [39, 15]. A number of studies have investigated attacks against poorly-randomized programs, including brute forcing [67], partial pointer overwrites [23], and return-oriented programming [64, 61]. Our ASR design is more fine-grained than existing techniques and robust against these attacks and information leakage. In addition, none of the existing approaches can support stateful live rerandomization. The general idea of randomization has also been applied to instruction sets (to thwart code injection attacks) [38, 58, 34], data representation (to protect noncontrol data) [13], data structures (to mitigate rootkits) [46], memory allocators (to protect against heap exploits) [53]. Our struct layout randomization is similar to the one presented in [46], but our ASR design generalizes this strategy to the internal layout of any memory object (including code) and also

allows live layout rerandomization. Finally, randomization as a general form of diversification [26] has been proposed to execute multiple program variants in parallel and detect attacks from divergent behavior [20, 62, 63].

Operating system defenses. Prior work on OS defenses against memory exploits focuses on control-flow attacks. SecVisor [65] is a hypervisor-based solution which uses memory virtualization to enforce $W\oplus X$ protection and prevent code injection attacks. Similarly, NICKLE [60] is a VMM-based solution which stores authenticated kernel code in guest-isolated shadow memory regions and transparently redirects execution to these regions at runtime. Unlike SecVisor, NICKLE can support unmodified OSEs and seamlessly handle mixed kernel pages with code and data. hvmHarvard [28] is a hypervisor-based solution similar to NICKLE, but improves its performance with a more efficient instruction fetch redirection strategy at the page level. The idea of memory shadowing is also explored in HookSafe [69], a hypervisor-based solution which relocates kernel hooks to dedicated memory pages and employs a hook indirection layer to disallow unauthorized overrides. Other techniques to defend against kernel hook hijacking have suggested dynamic monitoring strategies [74, 57] and compiler-based indirection mechanisms [44]. Finally, Dalton et al. [21] present a buffer overflow detection technique based on dynamic information flow tracking and demonstrate its practical applicability to the Linux kernel. None of the techniques discussed here provides a comprehensive solution to OS-level attacks. Remarkably, none of them protects noncontrol data, a common target of attacks in local exploitation scenarios.

Live rerandomization. Unlike our solution, none of the existing ASR techniques can support live rerandomization with no state loss. Prior work that comes closest to our live rerandomization technique is in the general area of dynamic software updating. Many solutions have been proposed to apply run-time updates to user programs [51, 47, 8, 19] and operating systems [48, 10, 9]. Our rerandomization technique shares with these solutions the ability to modify code and data of a running system without service interruption. The fundamental difference is that these solutions apply run-time changes in place, essentially assuming a fixed memory layout where any state transformation is completely delegated to the programmer. Our solution, in contrast, is generic and automated, and can seamlessly support arbitrary memory layout transformations between variants at runtime. Other solutions have proposed process-level run-time updates to release some of the assumptions on the memory layout [30, 31], but they still delegate the state transfer process completely to the programmer. This completely hinders their applicability in live rerandomization scenarios where arbitrary layout transformations are allowed.

9 Conclusion

In this paper, we introduced the first ASR design for operating systems. To fully explore the design space, we presented an analysis of the different constraints and attack models inside the OS, while highlighting the challenges of OS-level ASR. Our analysis reveals a fundamental gap with long-standing assumptions in existing application-level solutions. For example, we show that information leakage, traditionally dismissed as a relatively rare event, becomes a major concern inside the OS. Building on these observations, our design takes the first step towards truly fine-grained ASR for OSEs. While our prototype system is targeted towards component-based OS architectures, the principles and the techniques presented are of much more general applicability. Our technique can also be applied to generic user programs, improving existing application-level techniques in terms of both performance and security, and opening up opportunities for third-generation ASR systems. The key to good performance (and no impact on the distribution model) is our link-time ASR strategy used in combination with live rerandomization. In addition, this strategy is more portable and much safer than existing techniques, which either rely on complex binary rewriting or require a substantial amount of untrusted code exposed to the runtime. In our technique, the complex rerandomization code runs completely sandboxed and any unexpected run-time error has no impact on normal execution. The key to good security is the better randomization granularity combined with periodic live rerandomization. Unlike existing techniques, we can (re)randomize the internal layout of memory objects and periodically rerandomize the system with no service interruption or state loss. These properties are critical to counter information leakage attacks and truly maximize the unobservability of the system.

10 Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. This work has been supported by European Research Council under grant ERC Advanced Grant 2008 - R3S3.

References

- [1] ASLR: leopard versus vista. <http://blog.laconicsecurity.com/2008/01/aslr-leopard-versus-vista.html>.
- [2] Linux vmsplince vulnerabilities. http://isec.pl/vulnerabilities/isec-0026-vmsplince_to_kernel.txt.
- [3] The story of a simple and dangerous kernel bug. <http://butnotyet.tumblr.com/post/175132533/the-story-of-a-simple-and-dangerous-kernel-bug>.

- [4] OpenBSD's IPv6 mbufs remote kernel buffer overflow. <http://www.securityfocus.com/archive/1/462728/30/0/threaded>, 2007.
- [5] Microsoft windows TCP/IP IGMP MLD remote buffer overflow vulnerability. <http://www.securityfocus.com/bid/27100>, 2008.
- [6] FUSE: filesystem in userspace. <http://fuse.sourceforge.net/>, 2012.
- [7] Green hills integrity. <http://www.ghs.com/products/rtos/integrity.html>, 2012.
- [8] ALTEKAR, G., BAGRAK, I., BURSTEIN, P., AND SCHULTZ, A. OPUS: online patches and updates for security. In *Proc. of the 14th USENIX Security Symp.* (2005), vol. 14, pp. 19–19.
- [9] ARNOLD, J., AND KAASHOEK, M. F. Ksplice: Automatic rebootless kernel updates. In *Proc. of the Fourth European Conf. on Computer Systems* (2009), pp. 187–198.
- [10] BAUMANN, A., APPAVOO, J., WISNIEWSKI, R. W., SILVA, D. D., KRIEGER, O., AND HEISER, G. Reboots are for hardware: Challenges and solutions to updating an operating system on the fly. In *Proc. of the USENIX Annual Tech. Conf.* (2007), pp. 1–14.
- [11] BERGER, E. D., ZORN, B. G., AND MCKINLEY, K. S. Reconsidering custom memory allocation. In *Proc. of the 17th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications* (2002), pp. 1–12.
- [12] BHATKAR, S., DUVARNEY, D. C., AND SEKAR, R. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proc. of the 12th USENIX Security Symp.* (2003), p. 8.
- [13] BHATKAR, S., AND SEKAR, R. Data space randomization. In *Proc. of the Fifth Int'l Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment* (2008), pp. 1–22.
- [14] BHATKAR, S., SEKAR, R., AND DUVARNEY, D. C. Efficient techniques for comprehensive protection from memory error exploits. In *Proc. of the 14th USENIX Security Symp.* (2005), p. 17.
- [15] BOJINOV, H., BONEH, D., CANNINGS, R., AND MALCHEV, I. Address space randomization for mobile devices. In *Proc. of the Fourth ACM Conf. on Wireless network security* (2011), pp. 127–138.
- [16] BOYD-WICKIZER, S., AND ZELDOVICH, N. Tolerating malicious device drivers in linux. In *Proc. of the USENIX Annual Tech. Conf.* (2010), pp. 9–9.
- [17] C-SKILLS. Linux udev trickery. <http://c-skills.blogspot.com/2009/04/udev-trickery-cve-2009-1185-and-cve.html>.
- [18] CHEN, H., MAO, Y., WANG, X., ZHOU, D., ZELDOVICH, N., AND KAASHOEK, M. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proc. of the Second Asia-Pacific Workshop on Systems* (2011).
- [19] CHEN, H., YU, J., CHEN, R., ZANG, B., AND YEW, P. POLUS: a Powerful live updating system. In *Proc. of the 29th Int'l Conf. on Software Engineering* (2007), pp. 271–281.
- [20] COX, B., EVANS, D., FILIPI, A., ROWANHILL, J., HU, W., DAVIDSON, J., KNIGHT, J., NGUYEN-TUONG, A., AND HISER, J. N-variant systems: a secretless framework for security through diversity. In *Proc. of the 15th USENIX Security Symp.* (2006), pp. 105–120.
- [21] DALTON, M., KANNAN, H., AND KOZYRAKIS, C. Real-world buffer overflow protection for userspace & kernelspace. In *Proc. of the 17th USENIX Security Symp.* (2008), pp. 395–410.
- [22] DESIGNER, S. Getting around non-executable stack (and fix). <http://seclists.org/bugtraq/1997/Aug/63>.
- [23] DURDEN, T. Bypassing PaX ASLR protection.
- [24] EDGE, J. Linux ASLR vulnerabilities. <http://lwn.net/Articles/330866/>, 2009.
- [25] ESSER, S. Exploiting the iOS kernel. In *Black Hat USA* (2011).
- [26] FORREST, S., SOMAYAJI, A., AND ACKLEY, D. Building diverse computer systems. In *Proc. of the 6th Workshop on Hot Topics in Operating Systems* (1997), pp. 67–.
- [27] GIUFFRIDA, C., CAVALLARO, L., AND TANENBAUM, A. S. We crashed, now what? In *Proc. of the 6th Workshop on Hot Topics in System Dependability* (2010), pp. 1–8.
- [28] GRACE, M., WANG, Z., SRINIVASAN, D., LI, J., JIANG, X., LIANG, Z., AND LIAXH, S. Transparent protection of commodity OS kernels using hardware virtualization. In *Proc. of the 6th Conf. on Security and Privacy in Communication Networks* (2010), pp. 162–180.
- [29] GUO, P. J., AND ENGLER, D. Linux kernel developer responses to static analysis bug reports. In *Proc. of the USENIX Annual Tech. Conf.* (2009), pp. 285–292.
- [30] GUPTA, D., AND JALOTE, P. On line software version change using state transfer between processes. *Softw. Pract. and Exper.* 23, 9 (1993), 949–964.
- [31] HAYDEN, C. M., SMITH, E. K., HICKS, M., AND FOSTER, J. S. State transfer for clear and efficient runtime updates. In *Proc. of the Third Int'l Workshop on Hot Topics in Software Upgrades* (2011), pp. 179–184.
- [32] HERDER, J. N., BOS, H., GRAS, B., HOMBURG, P., AND TANENBAUM, A. S. Reorganizing UNIX for reliability. In *Proc. of the 11th Asia-Pacific Conf. on Advances in Computer Systems Architecture* (2006), pp. 81–94.
- [33] HILDEBRAND, D. An architectural overview of QNX. In *Proc. of the Workshop on Micro-kernels and Other Kernel Architectures* (1992), pp. 113–126.
- [34] HU, W., HISER, J., WILLIAMS, D., FILIPI, A., DAVIDSON, J. W., EVANS, D., KNIGHT, J. C., NGUYEN-TUONG, A., AND ROWANHILL, J. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proc. of the Second Int'l Conf. on Virtual Execution Environments* (2006), pp. 2–12.
- [35] HUND, R., HOLZ, T., AND FREILING, F. C. Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In *Proc. of the 18th USENIX Security Symp.* (2009), pp. 383–398.
- [36] HUNT, G. C., AND LARUS, J. R. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.* 41, 2 (2007), 37–49.
- [37] JANMAR, K. FreeBSD 802.11 remote integer overflow. In *Black Hat Europe* (2007).
- [38] KC, G. S., KEROMYTIS, A. D., AND PREVELAKIS, V. Countering code-injection attacks with instruction-set randomization. In *Proc. of the 10th ACM Conf. on Computer and Commun. Security* (2003), pp. 272–280.
- [39] KIL, C., JUN, J., BOOKHOLT, C., XU, J., AND NING, P. Address space layout permutation (ASLP): towards Fine-Grained randomization of commodity software. In *Proc. of the 22nd Annual Computer Security Appl. Conf.* (2006), pp. 339–348.
- [40] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: formal verification of an OS kernel. In *Proc. of the 22nd ACM Symp. on Oper. Systems Prin.* (2009), ACM, pp. 207–220.
- [41] LABS, O. K. OKL4 community site. <http://wiki.ok-labs.com/>, 2012.

- [42] LATTNER, C., AND ADVE, V. LLVM: a compilation framework for lifelong program analysis & transformation. In *Proc. of the Int'l Symp. on Code Generation and Optimization* (2004), p. 75.
- [43] LENHARTH, A., ADVE, V. S., AND KING, S. T. Recovery domains: an organizing principle for recoverable operating systems. In *Proc. of the 14th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* (2009), pp. 49–60.
- [44] LI, J., WANG, Z., BLETSCH, T., SRINIVASAN, D., GRACE, M., AND JIANG, X. Comprehensive and efficient protection of kernel control data. *IEEE Trans. on Information Forensics and Security* 6, 4 (2011), 1404–1417.
- [45] LIAKH, S., GRACE, M., AND JIANG, X. Analyzing and improving linux kernel memory protection: a model checking approach. In *Proc. of the 26th Annual Computer Security Appl. Conf.* (2010), pp. 271–280.
- [46] LIN, Z., RILEY, R. D., AND XU, D. Polymorphing software by randomizing data structure layout. In *Proc. of the 6th Int'l Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment* (2009), pp. 107–126.
- [47] MAKRIS, K., AND BAZZI, R. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proc. of the USENIX Annual Tech. Conf.* (2009), pp. 397–410.
- [48] MAKRIS, K., AND RYU, K. D. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proc. of the Second European Conf. on Computer Systems* (2007), pp. 327–340.
- [49] MALIK, A. M., MCINNES, J., AND BEEK, P. v. Optimal basic block instruction scheduling for Multiple-Issue processors using constraint programming. In *Proc. of the 18th IEEE Int'l Conf. on Tools with Artificial Intelligence* (2006), pp. 279–287.
- [50] MICROSOFT. Windows User-Mode driver framework. <http://msdn.microsoft.com/en-us/windows/hardware/gg463294>, 2010.
- [51] NEAMTIU, I., HICKS, M., STOYLE, G., AND ORIOL, M. Practical dynamic software updating for C. *ACM SIGPLAN Notices* 41, 6 (2006), 72–83.
- [52] NERGAL. The advanced return-into-lib(c) exploits. *Phrack Magazine* 4, 58 (2001).
- [53] NOVARK, G., AND BERGER, E. D. DieHarder: securing the heap. In *Proc. of the 17th ACM Conf. on Computer and Commun. Security* (2010), pp. 573–584.
- [54] ONARLIOGLU, K., BILGE, L., LANZI, A., BALZAROTTI, D., AND KIRDA, E. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proc. of the 26th Annual Computer Security Appl. Conf.* (2010), pp. 49–58.
- [55] PALIX, N., THOMAS, G., SAHA, S., CALVES, C., LAWALL, J., AND MULLER, G. Faults in linux: ten years later. In *Proc. of the 16th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* (2011), pp. 305–318.
- [56] PERLA, E., AND OLDANI, M. *A guide to kernel exploitation: attacking the core*. 2010.
- [57] PETRONI, JR., N. L., AND HICKS, M. Automated detection of persistent kernel control-flow attacks. In *Proc. of the 14th ACM Conf. on Computer and Commun. Security* (2007), pp. 103–115.
- [58] PORTOKALIDIS, G., AND KEROMYTIS, A. D. Fast and practical instruction-set randomization for commodity systems. In *Proc. of the 26th Annual Computer Security Appl. Conf.* (2010), pp. 41–48.
- [59] RATANAWORABHAN, P., LIVSHITS, B., AND ZORN, B. NOZ-ZLE: a defense against heap-spraying code injection attacks. In *Proc. of the 18th USENIX Security Symp.* (2009), pp. 169–186.
- [60] RILEY, R., JIANG, X., AND XU, D. Guest-Transparent prevention of kernel rootkits with VMM-Based memory shadowing. In *Proc. of the 11th Int'l Conf. on Recent Advances in Intrusion Detection* (2008), pp. 1–20.
- [61] ROGLIA, G. F., MARTIGNONI, L., PALEARI, R., AND BRUSCHI, D. Surgically returning to randomized lib(c). In *Proc. of the 2009 Annual Computer Security Appl. Conf.* (2009), pp. 60–69.
- [62] SALAMAT, B., GAL, A., JACKSON, T., MANIVANNAN, K., WAGNER, G., AND FRANZ, M. Multi-variant program execution: Using multi-core systems to defuse Buffer-Overflow vulnerabilities. In *Proc. of the 2008 Int'l Conf. on Complex, Intelligent and Software Intensive Systems* (2008), pp. 843–848.
- [63] SALAMAT, B., JACKSON, T., GAL, A., AND FRANZ, M. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proc. of the Fourth European Conf. on Computer Systems* (2009), pp. 33–46.
- [64] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. Q: exploit hardening made easy. In *Proc. of the 20th USENIX Security Symp.* (2011), p. 25.
- [65] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proc. of the 21st ACM Symp. on Oper. Systems Prin.* (2007), pp. 335–350.
- [66] SHACHAM, H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proc. of the 14th ACM Conf. on Computer and Commun. Security* (2007), pp. 552–561.
- [67] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E., MODADUGU, N., AND BONEH, D. On the effectiveness of address-space randomization. In *Proc. of the 11th ACM Conf. on Computer and Commun. Security* (2004), pp. 298–307.
- [68] TEAM, P. Overall description of the PaX project. <http://pax.grsecurity.net/docs/pax.txt>, 2008.
- [69] WANG, Z., JIANG, X., CUI, W., AND NING, P. Countering kernel rootkits with lightweight hook protection. In *Proc. of the 16th ACM Conf. on Computer and Commun. Security* (2009), pp. 545–554.
- [70] WILSON, P. R. Uniprocessor garbage collection techniques. In *Proc. of the Int'l Workshop on Memory Management* (1992), pp. 1–42.
- [71] WU, J., CUI, H., AND YANG, J. Bypassing races in live applications with execution filters. In *Proc. of the 9th USENIX Symp. on Operating Systems Design and Implementation* (2010), pp. 1–13.
- [72] XU, H., AND CHAPIN, S. J. Improving address space randomization with a dynamic offset randomization technique. In *Proc. of the 2006 ACM Symp. on Applied Computing* (2006), pp. 384–391.
- [73] XU, J., KALBARCZYK, Z., AND IYER, R. K. Transparent runtime randomization for security. In *Proc. of the 22nd Int'l Symp. on Reliable Distributed Systems* (2003), pp. 260–269.
- [74] YIN, H., POOSANKAM, P., HANNA, S., AND SONG, D. HookScout: proactive binary-centric hook detection. In *Proc. of the 7th Int'l Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment* (2010), pp. 1–20.
- [75] YOUNG, C., JOHNSON, D. S., SMITH, M. D., AND KARGER, D. R. Near-optimal intraprocedural branch alignment. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation* (1997), pp. 183–193.
- [76] ZHANG, K., AND WANG, X. Peeping tom in the neighborhood: keystroke eavesdropping on multi-user systems. In *Proc. of the 18th USENIX Security Symp.* (2009), pp. 17–32.

From Throw-Away Traffic to Bots: Detecting the Rise of DGA-Based Malware

Manos Antonakakis^{‡,*}, Roberto Perdisci^{†,*}, Yacin Nadji^{*},
Nikolaos Vasiloglou[‡], Saeed Abu-Nimeh[‡], Wenke Lee^{*} and David Dagon^{*}

[‡]*Damballa Inc.*, [†]*University of Georgia*

{*manos,nvasil,sabunimeh*}@*damballa.com*, *perdisci@cs.uga.edu*

**Georgia Institute of Technology*

{*yacin.nadji, wenke*}@*cc.gatech.edu*, *dagon@sudo.sh*

Abstract

Many botnet detection systems employ a blacklist of known command and control (C&C) domains to detect bots and block their traffic. Similar to signature-based virus detection, such a botnet detection approach is static because the blacklist is updated only after running an external (and often manual) process of domain discovery. As a response, botmasters have begun employing domain generation algorithms (DGAs) to dynamically produce a large number of random domain names and select a small subset for actual C&C use. That is, a C&C domain is randomly generated and used for a very short period of time, thus rendering detection approaches that rely on static domain lists ineffective. Naturally, if we know how a domain generation algorithm works, we can generate the domains ahead of time and still identify and block botnet C&C traffic. The existing solutions are largely based on reverse engineering of the bot malware executables, which is not always feasible.

In this paper we present a new technique to detect randomly generated domains without reversing. Our insight is that most of the DGA-generated (random) domains that a bot queries would result in Non-Existent Domain (NXDomain) responses, and that bots from the same botnet (with the same DGA algorithm) would generate similar NXDomain traffic. Our approach uses a combination of clustering and classification algorithms. The clustering algorithm clusters domains based on the similarity in the make-ups of domain names as well as the groups of machines that queried these domains. The classification algorithm is used to assign the generated clusters to models of known DGAs. If a cluster cannot be assigned to a known model, then a new model is produced, indicating a new DGA variant or family. We implemented a prototype system and evaluated it on real-world DNS traffic obtained from large ISPs in North America. We report the discovery of twelve DGAs. Half of them are variants of known (botnet) DGAs, and the other half are brand new DGAs that have never been reported before.

1 Introduction

Botnets are groups of malware-compromised machines, or *bots*, that can be remotely controlled by an attacker (the *botmaster*) through a *command and control* (C&C) communication channel. Botnets have become the main platform for cyber-criminals to send spam, steal private information, host phishing web-pages, etc. Over time, attackers have developed C&C channels with different network structures. Most botnets today rely on a centralized C&C server, whereby bots query a predefined C&C domain name that resolves to the IP address of the C&C server from which commands will be received. Such centralized C&C structures suffer from the *single point of failure* problem because if the C&C domain is identified and taken down, the botmaster loses control over the entire botnet.

To overcome this limitation, attackers have used P2P-based C&C structures in botnets such as Nugache [35], Storm [38], and more recently Waledac [39], Zeus [2], and Alureon (a.k.a. TDL4) [12]. While P2P botnets provide a more robust C&C structure that is difficult to detect and take down, they are typically harder to implement and maintain. In an effort to combine the simplicity of centralized C&Cs with the robustness of P2P-based structures, attackers have recently developed a number of botnets that locate their C&C server through *automatically generated* pseudo-random domain names. In order to contact the botmaster, each bot periodically executes a *domain generation algorithm* (DGA) that, given a random seed (e.g., the current date), produces a list of *candidate* C&C domains. The bot then attempts to resolve these domain names by sending DNS queries until one of the domains resolves to the IP address of a C&C server. This strategy provides a remarkable level of *agility* because even if one or more C&C domain names or IP addresses are identified and taken down, the bots will eventually get the IP address of the relocated C&C server via DNS queries to the next set of automatically generated domains. Notable examples of DGA-

based botnets (or DGA-bots, for short) are Bobax [33], Kraken [29], Sinowal (a.k.a. Torpig) [34], Srizbi [30], Conficker-A/B [26], Conficker-C [23] and Murofet [31]. A defender can attempt to reverse engineer the bot malware, particularly its DGA algorithm, to pre-compute current and future candidate C&C domains in order to detect, block, and even take down the botnet. However, reverse engineering is not always feasible because the bot malware can be updated very quickly (e.g., hourly) and obfuscated (e.g., encrypted, and only decrypted and executed by external triggers such as time).

In this paper, we propose a novel detection system, called Pleiades, to identify DGA-based bots within a monitored network without reverse engineering the bot malware. Pleiades is placed “below” the local recursive DNS (RDNS) server or at the edge of a network to monitor DNS query/response messages from/to the machines within the network. Specifically, Pleiades analyzes DNS queries for domain names that result in *Name Error* responses [19], also called NXDOMAIN responses, i.e., domain names for which no IP addresses (or other resource records) exist. In the remainder of this paper, we refer to these domain names as NXDomains. The focus on NXDomains is motivated by the fact that modern DGA-bots tend to query large sets of domain names among which relatively few successfully resolve to the IP address of the C&C server. Therefore, to automatically identify DGA domain names, Pleiades searches for relatively large clusters of NXDomains that (i) have similar syntactic features, and (ii) are queried by multiple potentially compromised machines during a given epoch. The intuition is that in a large network, like the ISP network where we ran our experiments, multiple hosts may be compromised with the same DGA-bots. Therefore, each of these compromised assets will generate several DNS queries resulting in NXDomains, and a subset of these NXDomains will likely be queried by more than one compromised machine. Pleiades is able to automatically identify and filter out “accidental”, user-generated NXDomains due to typos or mis-configurations. When Pleiades finds a cluster of NXDomains, it applies statistical learning techniques to build a model of the DGA. This is used later to detect future compromised machines running the same DGA and to detect *active domain names* that “look similar” to NXDomains resulting from the DGA and therefore probably point to the botnet C&C server’s address.

Pleiades has the advantage of being able to discover and model new DGAs without labor-intensive malware reverse-engineering. This allows our system to detect new DGA-bots before any sample of the related malware family is captured and analyzed. Unlike previous work on DNS traffic analysis for detecting malware-related [4] or malicious domains in general [3, 6], Pleiades lever-

ages *throw-away traffic* (i.e., unsuccessful DNS resolutions) to (1) discover the rise of new DGA-based botnets, (2) accurately detect bot-compromised machines, and (3) identify and block the active C&C domains queried by the discovered DGA-bots. Pleiades achieves these goals by monitoring the DNS traffic in local networks, without the need for a large-scale deployment of DNS analysis tools required by prior work.

Furthermore, while botnet detection systems that focus on network flow analysis [13, 36, 44, 46] or require deep packet inspection [10, 14] may be capable of detecting compromised machines within a local network, they do not scale well to the overwhelming volume of traffic typical of large ISP environments. On the other hand, Pleiades employs a *lightweight* DNS-based monitoring approach, and can detect DGA-based malware by focusing on a small fraction of all DNS traffic in an ISP network. This allows Pleiades to scale well to very large ISP networks, where we evaluated our prototype system.

This paper makes the following contributions:

- We propose Pleiades, the first DGA-based botnet identification system that efficiently analyzes streams of unsuccessful domain name resolutions, or NXDomains, in large ISP networks to automatically identify DGA-bots.
- We built a prototype implementation of Pleiades, and evaluated its DGA identification accuracy over a large labeled dataset consisting of a mix of NXDomains generated by four different known DGA-based botnets and NXDomains “accidentally” generated by typos or mis-configurations. Our experiments demonstrate that Pleiades can accurately detect DGA-bots.
- We deployed and evaluated our Pleiades prototype in a large *production* ISP network for a period of 15 months. Our experiments discovered twelve new DGA-based botnets and enumerated the compromised machines. Half of these new DGAs have never been reported before.

The remainder of the paper is organized as follows. In Section 2 we discuss related work. We provide an overview of Pleiades in Section 3. The DGA discovery process is described in Section 4. Section 5 describes the DGA classification and C&C detection processes. We elaborate on the properties of the datasets used and the way we obtained the ground truth in Section 6. The experimental results are presented in Section 7 while we discuss the limitations of our systems in Section 8. We conclude the paper in Section 9.

2 Related Work

Dynamic domain generation has been used by malware to evade detection and complicate mitigation, e.g., Bobax, Kraken, Torpig, Srizbi, and Conficker [26]. To uncover the underlying domain generation algorithm (DGA), researchers often need to reverse engineer the bot binary. Such a task can be time consuming and requires advanced reverse engineering skills [18].

The infamous Conficker worm is one of the most aggressive pieces of malware with respect to domain name generation. The “C” variant of the worm generated 50,000 domains per day. However, Conficker-C only queried 500 of these domains every 24 hours. In older variants of the worm, A and B, the worm cycled through the list of domains every three and two hours, respectively. In Conficker-C, the length of the generated domains was between four and ten characters, and the domains were distributed across 110 TLDs [27].

Stone-Gross et al. [34] were the first to report on domain fluxing. In the past, malware used IP fast-fluxing, where a single domain name pointed to several IP addresses to avoid being taken down easily. However, in domain fluxing malware uses a domain generation algorithm to generate several domain names, and then attempt to communicate with a subset of them. The authors also analyzed Torpig’s DGA and found that the bot utilizes Twitter’s API. Specifically, it used the second character of the most popular Twitter search and generated a new domain every day. It was updated to use the second character of the 5th most popular Twitter search. Srizbi [40] is another example of a bot that utilizes a DGA by using unique magic number. Researchers identified several unique magic numbers from multiple copies of the bot. The magic number is XOR’ed with the current date and a different set of domains is generated. Only the characters “q w e r t y u i o p a s d f” are used in the generated domain names.

Yadav et. al. proposed a technique to identify botnets by finding randomly generated domain names [42], and improvements that also include NXDomains and temporal correlation [43]. They evaluated their approaches by automatically detecting Conficker botnets in an offline dataset from a Tier-1 ISP in South Asia in the first paper, and both the ISP dataset and a university’s DNS logs in the second.

Villamarin-Salomon and Brustoloni [37] compared two approaches to identify botnet C&Cs. In their first approach, they identified domains with high query rates or domains that were temporally correlated. They used Chebyshev’s inequality and Mahalanobis distance to identify anomalous domains. In their second approach, they analyzed recurring “dynamic” DNS replies with NXDomain responses. Their experiments showed that the first approach was ineffective, as several legitimate

services use DNS with short time-to-live (TTL) values. However, their second approach yielded better detection and identified suspicious C&C domains.

Pleiades differs from the approaches described above in the following ways. **(A)** Our work models five different types of bot families including Conficker, Murofet, Sinowal, and Bobax. **(B)** We model these bot families using two clustering techniques. The first utilizes the distribution of the characters and 2-grams in the domain name. The second relies on historical data that shows the relationship between hosts and domain names. **(C)** We build a classification model to predict the maliciousness of domains that deviate from the two clustering techniques.

Unlike previous work, our approach does not require active probing to maintain a fresh list of legitimate domains. Our approach does not rely on external reputation databases (e.g., DNSBLs); instead, it only requires access to local DNS query streams to identify new clusters of DGA NXDomains. Not only does our approach identify new DGAs, but it also builds models for these DGAs to classify hosts that will generate similar NXDomains in the future. Furthermore, among the list of identified domains in the DGAs, our approach pinpoints the C&C domains. Lastly, we note that our work is complementary to the larger collection of previous research that attempts to detect and identify malicious domain names, e.g., [3,4].

3 System Overview

In this section, we provide a high-level overview of our DGA-bot detection system Pleiades. As shown in Figure 1, Pleiades consists of two main modules: a *DGA Discovery* module, and a *DGA Classification and C&C Detection* module. We discuss the roles of these two main modules and their components, and how they are used in coordination to *actively learn* and update DGA-bot detection models. We describe these components in more detail in Sections 4 and 5.

3.1 DGA Discovery

The *DGA Discovery* module analyzes streams of unsuccessful DNS resolutions, as seen from “below” a local DNS server (see Figure 1). All NXDomains generated by network users are collected during a given epoch (e.g., one day). Then, the collected NXDomains are clustered according to the following two similarity criteria: (1) the domain name strings have similar statistical characteristics (e.g., similar length, similar level of “randomness”, similar character frequency distribution, etc.) and (2) the domains have been queried by overlapping sets of hosts. The main objective of this NXDomain clustering process is to group together domain names that likely are automatically generated by the same algorithm running on multiple machines within the monitored network.

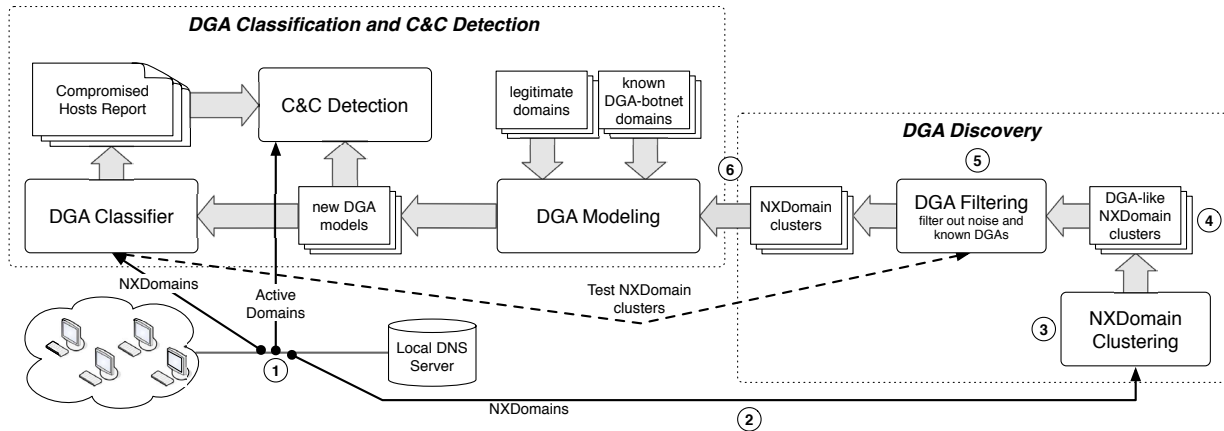


Figure 1: A high level overview of Pleiades.

Naturally, because this clustering step is unsupervised, some of the output NXDomain clusters may contain groups of domains that happen to be similar by chance (e.g., NXDomains due to common typos or to mis-configured applications). Therefore, we apply a subsequent filtering step. We use a supervised *DGA Classifier* to prune NXDomain clusters that appear to be generated by DGAs that we have previously discovered and modeled, or that contain domain names that are similar to popular legitimate domains. The final output of the *DGA Discovery* module is a set of NXDomain clusters, each of which likely represents the NXDomains generated by previously unknown or not yet modeled DGA-bots.

3.2 DGA Classification and C&C Detection

Every time a new DGA is discovered, we use a supervised learning approach to build models of what the domains generated by this new DGA “look like”. In particular, we build two different statistical models: (1) a statistical multi-class classifier that focuses on assigning a specific DGA label (e.g., *DGA-Conficker.C*) to the *set of NXDomains* generated by a host h_i and (2) a Hidden Markov Model (HMM) that focuses on finding *single active domain names* queried by h_i that are likely generated by a DGA (e.g., *DGA-Conficker.C*) running on the host, and are therefore good *candidate C&C domains*.

The *DGA Modeling* component receives different sets of domains labeled as *Legitimate* (i.e., “non-DGA”), *DGA-Bobax*, *DGA-Torpig/Sinowal*, *DGA-Conficker.C*, *New-DGA-v1*, *New-DGA-v2*, etc., and performs the training of the multi-class *DGA Classifier* and the HMM-based *C&C Detection* module.

The *DGA Classification* module works as follows. Similar to the *DGA Discovery* module, we monitor the stream of NXDomains generated by each client machine

“below” the local recursive DNS server.

Given a subset of NXDomains generated by a machine, we extract a number of statistical features related to the NXDomain strings. Then, we ask the *DGA Classifier* to identify whether this subset of NXDomains resembles the NXDomains generated by previously discovered DGAs. That is, the classifier will either label the subset of NXDomains as generated by a known DGA, or tell us that it does not fit any model. If the subset of NXDomains is assigned a specific DGA label (e.g., *DGA-Conficker.C*), the host that generated the NXDomains is deemed to be compromised by the related DGA-bot.

Once we obtain the list of machines that appear to be compromised with DGA-based bots, we take detection one step further. While all previous steps focused on NXDomains, we now turn our attention to domain names for which we observe valid resolutions. Our goal is to identify which domain names, among the ones generated by the discovered DGA-based bots, actually resolve into a valid IP address. In other words, we aim to identify the botnet’s active C&C server.

To achieve this goal, we consider all domain names that are successfully resolved by hosts which have been classified as running a given DGA, say *New-DGA-vX*, by the *DGA Classifier*. Then, we test these successfully resolved domains against an HMM specifically trained to recognize domains generated by *New-DGA-vX*. The HMM analyzes the sequence of characters that compose a domain name d , and computes the likelihood that d is generated by *New-DGA-vX*.

We use an HMM, rather than the *DGA Classifier*, because for the C&C detection phase we need to classify *single domain names*. The *DGA Classifier* is not suitable for this task because it expects as input *sets of NXDomains* generated by a given host to assign a label to the

DGA-bot running on that host. Some of the features used by the *DGA Classifier* cannot be reliably extracted from a single domain name (see Sections 4.1.1 and 5.2).

4 DGA Discovery

The *DGA Discovery* module analyzes sequences of NXDomains generated by hosts in a monitored network, and in a *completely unsupervised* way, clusters NXDomains that are being automatically generated by a DGA. We achieve this goal in multiple steps (see Figure 1). First (*Step 1*), we collect sequences of NXDomains generated by each host during an epoch E . Afterwards (*Step 2*), we split the overall set of NXDomains generated by all monitored hosts into small subsets, and translate each set into a statistical feature vector (see Section 4.1.1). We then apply the X-means clustering algorithm [24] to group these domain subsets into larger clusters of domain names that have similar *string-based* characteristics.

Separately (*Step 3*), we cluster the NXDomains based on a completely different approach that takes into account whether two NXDomains are being queried by overlapping sets of hosts. First, we build a bipartite *host association* graph in which the two sets of vertices represent distinct hosts and distinct NXDomains, respectively. A host vertex V_{h_i} is connected to an NXDomain vertex V_{n_j} if host h_i queried NXDomain n_j . This allows us to identify different NXDomains that have been queried by overlapping sets of hosts. Intuitively, if two NXDomains are queried by multiple common hosts, this indicates that the querying hosts may be running the same DGA. We can then leverage this definition of similarity between NXDomains to cluster them (see Section 4.1.3).

These two *distinct views* of similarities among NXDomains are then reconciled in a *cluster correlation* phase (*Step 4*). This step improves the quality of the final NXDomains clusters by combining the clustering results obtained in *Step 2* and *Step 3*, and reduces possible noise introduced by clusters of domains that may appear similar purely by chance, for example due to similar typos originating from different network users.

The final clusters represent different groups of NXDomains, each containing domain names that are highly likely to be generated by the same DGA. For each of the obtained NXDomain clusters, the question remains if they belong to a known DGA, or a newly discovered one. To answer this question (*Step 5*), we use the *DGA Classifier* described in Section 5.2, which is specifically trained to distinguish between sets of NXDomains generated by currently known DGAs. Clusters that match previously modeled DGAs are discarded. On the other hand, if a cluster of NXDomains does not resemble any previously seen DGAs, we identify the cluster of NXDomains as having been generated by a new, previously unknown DGA. These NXDomains will then be sent (*Step*

6) to the *DGA Modeling* module, which will update (i.e., re-train) the *DGA Classifier* component.

4.1 NXDomain Clustering

We now describe the *NXDomain Clustering* module in detail. First, we introduce the statistical features Pleiades uses to translate small sets of NXDomains into feature vectors, and then discuss how these feature vectors are clustered to find similar NXDomains.

4.1.1 Statistical Features

To ease the presentation of how the statistical features are computed, we first introduce some notation that we will be using throughout this section.

Definitions and Notation A domain name d consists of a set of labels separated by dots, e.g., `www.example.com`. The rightmost label is called the *top-level* domain (TLD or $TLD(d)$), e.g., `com`. The *second-level* domain (2LD or $2LD(d)$) represents the two rightmost labels separated by a period, e.g., `example.com`. The *third-level* domain (3LD or $3LD(d)$) contains the three rightmost labels, e.g., `www.example.com`, and so on.

We will often refer to splitting a sequence $NX = \{d_1, d_2, \dots, d_m\}$ of NXDomains into a number of subsequences (or subsets) of length α , $NX_k = \{d_r, d_{r+1}, \dots, d_{r+\alpha-1}\}$, where $r = \alpha(k-1) + 1$ and $k = 1, 2, \dots, \lfloor \frac{m}{\alpha} \rfloor$. Subscript k indicates the k -th subsequence of length α in the sequence of m NXDomains NX . Each of the NX_k domain sequences can be translated into a feature vector, as described below.

n-gram Features Given a subsequence NX_k of α NXDomains, we measure the frequency distribution of n -grams across the domain name strings, with $n = 1, \dots, 4$. For example, for $n = 2$, we compute the frequency of each 2-gram. At this point, we can compute the median, average and standard deviation of the obtained distribution of 2-gram frequency values, thus obtaining three features. We do this for each value of $n = 1, \dots, 4$, producing 12 statistical features in total. By measuring the median, average and standard deviation, we are trying to capture the *shape* of the frequency distribution of the n -grams.

Entropy-based Features This group of features computes the entropy of the character distribution for separate domain levels. For example, we separately compute the character entropy for the 2LDs and 3LDs extracted from the domains in NX_k . To better understand how these features are measured, consider a set NX_k of α domains. We first extract the 2LD of each domain $d_i \in NX_k$, and for each domain we compute the entropy $H(2LD(d_i))$ of the characters of its 2LD. Then, we compute the average and standard deviation of the set of values $\{H(2LD(d_i))\}_{i=1 \dots \alpha}$. We repeat this for 3LDs and for the overall domain name strings. We measure a total

of six features, which capture the “level of randomness” in the domains. The intuition is that most DGAs produce random-looking domain name strings, and we want to account for this characteristic of the DGAs.

Structural Domain Features This group of features is used to summarize information about the structure of the NXDomains in NX_k , such as their length, the number of unique TLDs, and the number of domain levels. In total, we compute 14 features. Specifically, given NX_k , we compute the average, median, standard deviation, and variance of the length of the domain names (four features), and of the number of domain levels (four features). Also, we compute the number of distinct characters that appear in these NXDomains (one feature), the number of distinct TLDs, and the ratio between the number of domains under the .com TLD and the number of domains that use other TLDs (two features). The remaining features measure the average, median, and standard deviation of the occurrence frequency distribution for the different TLDs (three features).

4.1.2 Clustering using Statistical Features

To find clusters of similar NXDomains, we proceed as follows. Given the set NX of all NXDomains that we observed from all hosts in the monitored network, we split NX into subsets of size α , as mentioned in Section 4.1.1. Assuming m is the number of distinct NXDomains in NX , we split the set NX into $\lfloor \frac{m}{\alpha} \rfloor$ different subsets where $\alpha = 10$.

For each of the obtained subsets NX_k of NX , we compute the aforementioned 33 statistical features. After we have translated each NX_k into its corresponding feature vector, we apply the X-means clustering algorithm [24]. X-means will group the NX_k into X clusters, where X is automatically computed by an optimization process internal to X-means itself. At this point, given a cluster $C = \{NX_k\}_{k=1..l}$ of l NXDomain subsets, we simply take the union of the NX_k in C as an NXDomain cluster.

4.1.3 Clustering using Bipartite Graphs

Hosts that are compromised with the same DGA-based malware naturally tend to generate (with high probability) partially overlapping sets of NXDomains. On the other hand, other “non-DGA” NXDomains are unlikely to be queried by multiple hosts. For example, it is unlikely that multiple distinct users make identical typos in a given epoch. This motivates us to consider NXDomains that are queried by several common hosts as similar, and in turn use this similarity measure to cluster NXDomains that are likely generated by the same DGA.

To this end, we build a sparse association matrix M , where columns represent NXDomains and rows represent hosts that query more than two of the column NXDomains over the course of an epoch. We discard hosts

INPUT : Sparse matrix $M \in \mathbb{R}^{l \times k}$, in which the rows represent l hosts and the columns represent k NXDomains.

[1] : Normalize M : $\forall j = 1, \dots, k \quad \sum_{i=1}^l M_{i,j} = 1$

[2] : Compute the similarity matrix S from M : $S = M^T \cdot M$

[3] : Compute the first ρ eigenvectors from S by eigen-decomposition.

Let $U \in \mathbb{R}^{\rho \times k}$ be the matrix containing k vectors u_1, \dots, u_k of size ρ resulting from the eigen-decomposition of S

(a vector u_i is a reduced ρ -dimensional representation of the i -th NXDomain).

[4] : Cluster the vectors (i.e., the NXDomains) $\{u_i\}_{i=1..k}$ using the X-means algorithm

OUTPUT: Clusters of NXDomains

Algorithm 1: Spectral clustering of NXDomains.

that query only one NXDomain to reduce the dimensionality of the matrix, since they are extremely unlikely to be running a DGA given the low volume of NXDomains they produce. Let a matrix element $M_{i,j} = 0$, if host h_i did not query NXDomain n_j . Conversely, let $M_{i,j} = w_i$ if h_i did query n_j , where w_i is a weight.

All non-zero entries related to a host h_i are assigned the same weight $w_i \sim \frac{1}{k_i}$, where k_i is the number of NXDomains queried by host h_i . Clearly, M can be seen as a representation of a bipartite graph, in which a *host vertex* V_{h_i} is connected to an *NXDomain vertex* V_{n_j} with an edge of weight w_i if host h_i queried NXDomain n_j during the epoch under consideration. The intuition behind the particular method we use to compute the weights w_i is that we expect that the higher the number of unique NXDomains queried by a host h_i (i.e., the higher k_i) the less likely the host is “representative” of the NXDomains it queries. This is in a way analogous to the *inverse document frequency* used in the text mining domain [1, 7].

Once M is computed, we apply a graph partitioning strategy based on spectral clustering [21, 22], as summarized in Algorithm 1. As a first step, we compute the first ρ eigenvectors of M (we use $\rho = 15$ in our experiments), and then we map each NXDomain (each column of M) into a ρ -dimensional vector. In effect, this mapping greatly reduces the dimensionality of the NXDomain vectors from the total number of hosts (the number of rows in M) to ρ . We then used the obtained ρ -dimensional NXDomain representations and apply X-means to cluster the NXDomains based on their “host associations”. Namely, NXDomains are grouped together if they have been queried by a similar set of hosts.

4.1.4 Cluster Correlation

We now have two complementary views of how the NXDomains should be grouped based on two different definitions of similarity between domain names. Nei-

ther view is perfect, and the produced clusters may still contain noise. Correlating the two results helps filter the noise and output clusters of NXDomains that are more likely to be generated by a DGA. Cluster correlation is performed in the following way.

Let $\mathcal{A} = \{A_1, \dots, A_n\}$ be the set of NXDomain clusters obtained by using statistical features, as described in Section 4.1.2, and $\mathcal{B} = \{B_1, \dots, B_m\}$ be the set of NXDomain clusters derived from the bipartite graph partitioning approach discussed in Section 4.1.3. We compute the intersection between all possible pairs of clusters $I_{i,j} = A_i \cap B_j$, for $i = 1, \dots, n$ and $j = 1, \dots, m$. All correlated clusters $I_{i,j}$ that contain less than a predefined number λ of NXDomains (i.e., $|I_{i,j}| < \lambda$) are discarded, while the remaining correlated clusters are passed to the DGA filtering module described in Section 4.2. Clusters that are not sufficiently agreed upon by the two clustering approaches are not considered for further processing. We empirically set $\lambda = 40$ in preliminary experiments.

4.2 DGA Filtering

The DGA filtering module receives the NXDomain clusters from the clustering module. This filtering step compares the newly discovered NXDomain clusters to domains generated by known DGAs that we have already discovered and modeled. If the NXDomains in a correlated cluster $I_{i,j}$ are classified as being generated by a known DGA, we discard the cluster $I_{i,j}$. The reason is that the purpose of the *DGA Discovery* module is to find clusters of NXDomains that are generated (with high probability) by a new, never before seen DGA. At the same time, this filtering step is responsible for determining if a cluster of NXDomains is too noisy, i.e., if it likely contains a mix of DGA and “non-DGA” domains.

To this end, we leverage the *DGA Classifier* described in detail in Section 5. At a high level, we can treat the *DGA Classifier* as a function that takes as input a set NX_k of NXDomains, and outputs a set of tuples $\{(l_t, s_t)\}_{t=1..c}$, where l_t is a label (e.g., `DGA-Conficker.C`), and s_t is a score that indicates how confident the classifier is on attributing label l_t to NX_k , and c is the number of different classes (and labels) that the *DGA Classifier* can recognize.

When the DGA filtering module receives a new correlated cluster of NXDomains $I_{i,j}$, it splits the cluster into subsets of α NXDomains, and then passes each of these subsets to the *DGA Classifier*. Assume $I_{i,j}$ is divided into n different subsets. From the *DGA Classifier*, we obtain as a result n sets of tuples $\{(l_t, s_t)\}_{t=1..c}^{(1)}, \{(l_t, s_t)\}_{t=1..c}^{(2)}, \dots, \{(l_t, s_t)\}_{t=1..c}^{(n)}$.

First, we consider for each set of tuples $\{(l_t, s_t)\}_{t=1..c}^{(k)}$ with $k = 1, \dots, n$, the label $\hat{l}^{(k)}$ that was assigned the maximum score. We consider a cluster $I_{i,j}$ as too noisy if the related labels $\hat{l}^{(k)}$ are too diverse. Specifically, a

cluster is too noisy when the majority label among the $\hat{l}^{(k)}, k = 1, \dots, n$ was assigned to less than $\theta_{maj} = 75\%$ of the n domain subsets. The clusters that do not pass the θ_{maj} “purity” threshold will be discarded. Furthermore, NXDomain clusters whose majority label is the *Legitimate* label will also be discarded.

For each remaining cluster, we perform an additional “purity” check. Let the majority label for a given cluster $I_{i,j}$ be l^* . Among the set $\{(l_t, s_t)\}_{t=1..c}^{(k)}\}_{k=1..n}$ we take all the scores s_t whose related $l_t = l^*$. That is, we take the confidence score assigned by the classifier to the domain subsets that have been labeled as l^* , and then we compute the average $\mu(s_t)$ and the variance $\sigma^2(s_t)$ of these scores (notice that the scores s_t are in $[0, 1]$). We discard clusters whose $\sigma^2(s_t)$ is greater than a predefined threshold $\theta_\sigma = 0.001$, because we consider the domains in the cluster as not being *sufficiently similar* to the majority label class.

At this point, if $\mu(s_t) < \theta_\mu$, with $\theta_\mu = 0.98$, we deem the NXDomain cluster to be not similar enough to the majority label class, and instead we label it as “new DGA” and pass it to the *DGA Modeling* module. On the other hand, if $\mu(s_t) \geq \theta_\mu$, we confirm the majority label class (e.g., `DGA-Conficker.C`) and do not consider it further.

The particular choice for the values of the above mentioned thresholds are motivated in Section 7.2.

5 DGA Classification and C&C Detection

Once a new DGA is reported by the *DGA Discovery* module, we use a supervised learning approach to learn how to identify hosts that are infected with the related DGA-based malware by analyzing the set of NXDomains they generate. To identify compromised hosts, we collect the set of NXDomains NX_{h_i} generated by a host, h_i , and we ask the *DGA Classifier* whether NX_{h_i} likely “belongs” to a previously seen DGA or not. If the answer is yes, h_i is considered to be compromised and will be labeled with the name of the (suspected) DGA-bot that it is running.

In addition, we aim to build a classifier that can analyze the set of active domain names, say AD_{h_i} , resolved by a compromised host h_i and reduce it to a smaller subset $CC_{h_i} \subset AD_{h_i}$ of likely C&C domains generated by the DGA running on h_i . Finally, the set CC_{h_i} may be manually inspected to confirm the identification of C&C domain(s) and related IPs. In turn, the list of C&C IPs may be used to maintain an IP blacklist, which can be employed to block C&C communications and mitigate the effects of the malware infection. We now describe the components of the DGA classification and C&C detection module in more detail.

5.1 DGA Modeling

As mentioned in Section 4.2, the NXDomain clusters that pass the *DGA Filtering* and do not fit any known DGA model are (automatically) assigned a *New-DGA-vX* label, where *X* is a unique identifier. At this point, we build two different statical models representative of *New-DGA-vX*: (1) a statistical multi-class classifier that can assign a specific DGA label to the set of NXDomains generated by a host h_i and (2) a Hidden Markov Model (HMM) that can compute the probability that a single *active* domain queried by h_i was generated by the DGA running on the host, thus producing a list of *candidate C&C domains*.

The *DGA Modeling* module takes as input the following information: (1) a list of popular legitimate domain names extracted from the top 10,000 domains according to [alexa.com](http://www.alexa.com); (2) the list of NXDomains generated by running known DGA-bots in a controlled environment (see Section 6); (3) the clusters of NXDomains received from the *DGA Discovery* module. Let NX be one such newly discovered cluster of NXDomains. Because in some cases NX may contain relatively few domains, we attempt to extend the set NX to a larger set NX' that can help build better statistical models for the new DGA. To this end, we identify all hosts that “contributed” to the NXDomains clustered in NX from our sparse association matrix M and we gather all the NXDomains they generated during an epoch. For example, for a given host h_i that generated some of the domains clustered in NX , we gather all the other NXDomains domains NX'_{h_i} generated by h_i . We then add the set $NX' = \bigcup_i NX'_{h_i}$ to the training dataset (marked with the appropriate new DGA label). The reader may at this point notice that the set NX'_{h_i} may contain not only NXDomains generated by a host h_i due to running a DGA, but it may also include NXDomains “accidentally” generated by h_i . Therefore, this may introduce some noisy instances into the training dataset. However, the number of “accidental” NXDomains is typically very small, compared to the number of NXDomains generated by a DGA. Therefore, we rely on the generalization ability of the statistical learning algorithms we use to smooth away the effects of this potential source of noise. This approach works well in practice, as we will show in Section 7.

5.2 DGA Classifier

The *DGA Classifier* is based on a multi-class version of the Alternating Decision Trees (ADT) learning algorithm [9]. ADT leverages the high classification accuracy obtained by Boosting [17], while producing compact classification rules that can be more easily interpreted.

To detect hosts that are compromised with DGA-based malware, we monitor all NXDomains generated by each

host in the monitored network and periodically send this information to the *DGA Classifier*. Given a set NX_{h_i} of NXDomains generated by host h_i , we split NX_{h_i} into subsets of length α , and from each of these subsets we extract a number of statistical features, as described in Section 4.1.1. If one of these subsets of NXDomains is labeled by the *DGA Classifier* as being generated by a given DGA, we mark host h_i as compromised and we add its IP address and the assigned DGA label to a malware detection report.

5.3 C&C Detection

The *C&C Detection* module is based on Hidden Markov Models (HMM) [28]. We use one distinct HMM per DGA. Given the set $NX_{\mathcal{D}}$ of domains generated by a DGA \mathcal{D} , we consider each domain $d \in NX_{\mathcal{D}}$ separately, and feed these domains to an HMM for training. The HMM sees the domain names simply as a sequence of characters, and the result of the training is a model $HMM_{\mathcal{D}}$ that given a new domain name s in input will output the likelihood that s was generated by \mathcal{D} .

We use *left-to-right* HMM as they are used in practice to decrease the complexity of the model, effectively mitigating problems related to under-fitting. The HMM’s emission symbols are represented by the set of characters allowed in valid domain names (i.e., alphabetic characters, digits, ‘_’, ‘-’, and ‘.’). We set the number of hidden states to be equal to the average length of the domain names in the training dataset.

During operation, the *C&C Detection* module receives active domain names queried by hosts that have been previously classified by the *DGA Classifier* as being compromised with a DGA-based malware. Let h_i be one such host, and \mathcal{D} be the DGA running on h_i . The *C&C Detection* module will send every domain s resolved by h_i to $HMM_{\mathcal{D}}$, which will compute a likelihood score $f(s)$. If $f(s) > \theta_{\mathcal{D}}$, s is flagged as a good candidate C&C domain for DGA \mathcal{D} .

The threshold $\theta_{\mathcal{D}}$ can be learned during the training phase. First, we train the HMM with the set $NX_{\mathcal{D}}$. Then, we use a set L of legitimate “non-DGA” domains from Alexa. For each domain $l \in L$, we compute the likelihood $f(l)$ and set the threshold $\theta_{\mathcal{D}}$ so to obtain a maximum target false positive rate (e.g., max FPs=1%).

6 Data Collection

In this section we provide an overview of the amount of NXDomain traffic we observed during a period of fifteen consecutive months (our evaluation period), starting on November 1st, 2010 and ending on January 15th, 2012. Afterwards, we discuss how we collected the domain names used to train and test our *DGA Classifier* (see Section 5).

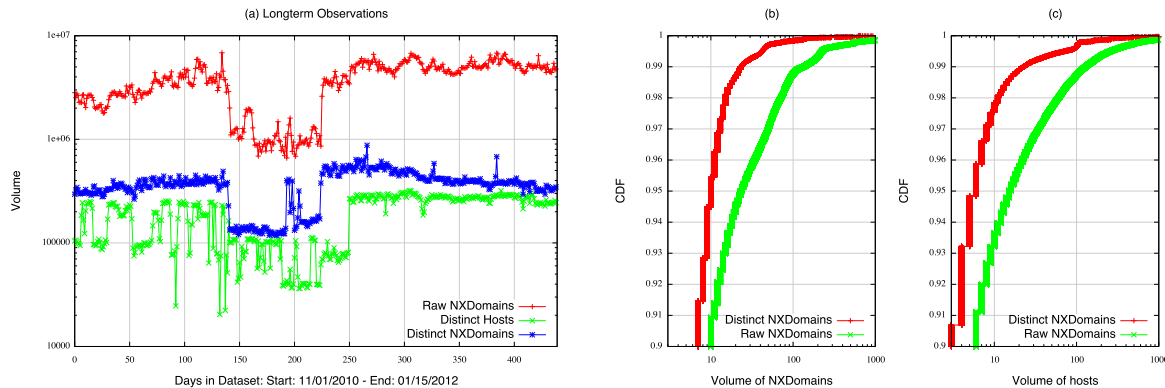


Figure 2: Observations from NXDomain traffic collected below a set of ISP recursive DNS servers over a 439 day window.

6.1 NXDomain Traffic

We evaluated Pleiades over a 15-month period against DNS traffic obtained by monitoring DNS messages to/from a set of recursive DNS resolvers operated by a large North American ISP. These servers were physically located in the US, and served (in average) over 2 million client hosts per day¹. Our monitoring point was “below” the DNS servers, thus providing visibility on the NXDomains generated by the individual client hosts.

Figure 2(a) reports, per each day, (1) the number of NXDomains as seen in the *raw* DNS traffic, (2) the number of distinct hosts that in the considered day query at least one NXDomains, and (3) the number of distinct (de-duplicated) NXDomains (we also filter out domain names that do not have a valid effective TLD [15,19,20]). The abrupt drop in the number of NXDomains and hosts (roughly a 30% reduction) experienced between 2011-03-24 and 2011-06-17 was due to a configuration change at the ISP network.

On average, we observed about 5 millions (raw) NXDomains, 187,600 distinct hosts that queried at least one NXDomains, and 360,700 distinct NXDomains overall, per each day. Therefore, the average size of the association matrix M used to perform spectral clustering (see Section 4.1.3) was $187,600 \times 360,700$. However, it is worth noting that M is sparse and can be efficiently stored in memory. In fact, the vast majority (about 90%) of hosts query less than 10 NXDomains per day, and therefore most rows in M will contain only a few non-zero elements. This is shown in Figure 2(b), which reports the cumulative distribution function (CDF) for the volume of NXDomains queried by a host in the monitored network. On the other hand, Figure 2(c) shows the CDF for the number of hosts that query an NXDomain (this relates directly to the sparseness of M according to its

¹We estimated the number of hosts by computing the average number of distinct client IPs seen per day.

columns).

6.2 Ground Truth

In order to generate the ground truth to train and evaluate the *DGA Classifier* (Section 5), we used a simple approach. To collect the NXDomains generated by known DGA-based malware we used two different methods. First, because the DGA used by different variants of Conficker and by Murofet are known (derived through reverse-engineering), we simply used the respective algorithms to generate a set of domain names from each of these botnets. To obtain a sample set of domains generated by Bobax and Sinowal, whose exact DGA algorithm is not known (at least not to us), we simply executed two malware samples (one per botnet) in a VM-based malware analysis framework that only allows DNS traffic², while denying any other type of traffic. Overall we collected 30,000 domains generated by Conficker, 26,078 from Murofet, 1,283 from Bobax and, 1,783 from Sinowal.

Finally, we used the top 10,000 most popular domains according to `alexa.com`, with and without the `www.` prefix. Therefore, overall we used 20,000 domain names to represent the “negative” (i.e., “non-DGA”) class during the training and testing of the *DGA Classifier*.

7 Analysis

In this section, we present the experimental results of our system. We begin by demonstrating Pleiades’ modeling accuracy with respect to known DGAs like Conficker, Sinowal, Bobax and Murofet. Then, we elaborate on the DGAs we discovered throughout the fifteen month NXDomain monitoring period. We conclude the section by summarizing the most interesting findings from the twelve DGAs we detected. Half of them use a DGA algorithm from a known malware family. The other half,

²We only allowed UDP port 53.

Table 1: Detection results (in %) using 10-fold cross validation for different values of α .

Class	$\alpha = 5$ NXDomains			$\alpha = 10$ NXDomains		
	TP_{rate}	FP_{rate}	AUC	TP_{rate}	FP_{rate}	AUC
Bobax	95	0.4	97	99	0	99
Conficker	98	1.4	98	99	0.1	99
Sinowal	99	0.1	98	100	0	100
Murofet	98	0.7	98	99	0.2	99
Benign	96	0.7	97	99	0.1	99

to the best of our knowledge, have **no** known malware association.

7.1 DGA Classifier’s Detection Results

In this section, we present the accuracy of the DGA classifier. We bootstrap the classifier with NXDomains from Bobax, Sinowal, Conficker-A, Conficker-B, Conficker-C and Murofet. We test the classifier in two modes. The first mode is bootstrapped with a “super” Conficker class composed of an equal number of samples from Conficker-A, Conficker-B and Conficker-C classes and another with each Conficker variant as its own class. As we mentioned in Section 5.2, the DGA classifier is based on a multi-class version of the Alternating Decision Trees (ADT) learning algorithm [9]. We build the vectors for each class by collecting NXDomains from one day of Honeypot traffic (in the case of Sinowal and Bobax) and one day of NXDomains produced by the DGAs for Conficker-A, Conficker-B, Conficker-C and Murofet. Finally, the domain names that were used to represent the benign class were the first 10,000 Alexa domain names with and without the `www.` child labels.

From the raw domain names in each of the classes, we randomly selected 3,000 sets of cardinality α . As a reminder, the values of α that we used were two, five, ten and 30. This was to build different training datasets in order to empirically decide which value of α would provide the best separation between the DGA models.

We generated additional testing datasets. The domain names we used in this case were from each class as in the case of the training dataset but we used different days. We do that so we get the minimum possible domain name overlap between the training and testing datasets. We evaluate the training datasets using two methods: 10-fold cross validation on the **training dataset** and by using the testing datasets computed from domains collected on **different days**. Both methods gave us very similar results. Our system performed the worst in the case of the 10-fold cross validation, therefore we chose to present this worst-case scenario.

In Table 1, we can see the detection results using two values for α , five and ten. We omit the results for the other values due to space limitations. The main confu-

sion between the classes was observed in the datasets that contained separate Conficker classes, specifically between the classes of Conficker-A and Conficker-B. To address this problem, we created a generic Conficker class that had an equal number of vectors from each Conficker variant. This merging of the Conficker variants into a single “super” class allowed the DGA classifier to correctly classify 99.72% (Table 1) of the instances (7,986 correctly classified vs 22 incorrectly classified). Using the datasets with the five classes of DGAs, the weighted average of the TP_{rates} and FP_{rates} were 99.7% and 0.1%, respectively. As we see in Table 1, $\alpha = 5$ performs reasonably well, but with a higher rate of FPs.

7.2 NXDomain Clustering Results

In this section, we will discuss results from the DGA discovery module. In particular, we elaborate on the selection of the thresholds used, the unique clusters identified and the false alerts the DGA discovery module produced over the duration of our study.

7.2.1 Correlation Thresholds

In order to set the thresholds θ_{maj} and θ_{σ} defined in Section 4.2, we spent the first five days of November 2010 labeling the 213 produced clusters as DGA related (Positive) or noisy (Negative). For this experiment, we included all produced clusters without filtering out those with $\theta_{\mu}=98\%$ (or higher) “similarity” to an already known one (see Section 4.2). In Figure 3, we can see in the Y-axis the percentage values for the dominant (non-benign) class in every cluster produced during these five days. In the X-axis we can see the variance that each dominant class had within each cluster. The results show that the Positive and Negative assignments had a clear cut, which we can achieve by setting the thresholds as $\theta_{maj} = 75\%$ and $\theta_{\sigma} = 0.001$. These thresholds gave us very good results throughout the duration of the experiments. As we will discuss in Section 7.2.3, the DGA discovery module falsely reported only five benign clusters over a period of 15 months. All falsely reported clusters had variance very close to 0.001.

7.2.2 New DGAs

Pleiades began clustering NXDomain traffic on the first day of November 2010. We bootstrapped the DGA modeler with domain names from already known DGAs and also a set of Alexa domain names as the benign class. In Table 2, we present all unique clusters we discovered throughout the evaluation period. The “Malware Family” column simply maps the variant to a known malware family if possible. We discover the malware family by checking the NXDomains that overlap with NXDomains we extracted from traffic obtained from a malware repository. Also, we manually inspected the clusters with the help of a security company’s threat team. The “First

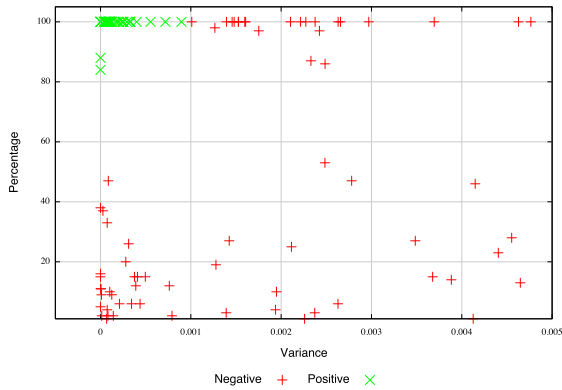


Figure 3: Thresholds θ_{maj} and θ_{σ} from the first five days of November 2010.

Seen” column denotes the first time we saw traffic from each DGA variant. Finally, the “Population on Discovery” column shows the variant population on the discovery day. We can see that we can detect each DGA variant with an average number of 32 “infected hosts” across the entire statewide ISP network coverage.

Table 2: DGAs Detected by Pleiades.

Malware Family	First Seen	Population on Discovery
Shiz/Simda-C [32]	03/20/11	37
Bamital [11]	04/01/11	175
BankPatch [5]	04/01/11	28
Expiro.Z [8]	04/30/11	7
Boonana [41]	08/03/11	24
Zeus.v3 [25]	09/15/11	39
New-DGA-v1	01/11/10	12
New-DGA-v2	01/18/11	10
New-DGA-v3	02/01/11	18
New-DGA-v4	03/05/11	22
New-DGA-v5	04/21/11	5
New-DGA-v6	11/20/11	10

As we see in Table 2, Pleiades reported six variants that belong to known DGA-enabled malware families [5,8,11,25,32,41]. Six more variants of NXDomains were reported and modeled by Pleiades but for these, to the best of our knowledge, no known malware can be associated with them. A sample set of 10 domain names for each one of these variants can be seen in Figure 4.

In the 15 months of our observations we observed an average population of 742 Conficker infected hosts in the ISP network. Murofet had the second largest population of infected hosts at 92 per day, while the Boonana DGA comes third with an average population of 84 infected hosts per day. The fastest growing DGA is Zeus.v3 with an average population of 50 hosts per day, however, during the last four days of the experiments the Zeus.v3 DGA had an average number of 134 infected hosts. It

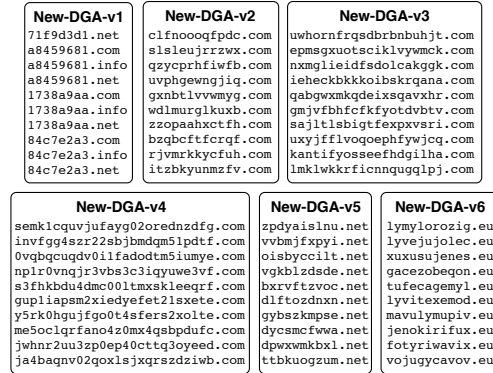


Figure 4: A sample of ten NXDomain for each DGA cluster that we could not associate with a known malware family.

is worth noting the New-DGA-v1 had an average of 19 hosts per day, the most populous of the newly identified DGAs.

7.2.3 False Reports on New DGAs

During our evaluation period we came across five categories of clusters falsely reported as new DGAs. In all of the cases, we modeled these classes in the DGA modeler as variants of the benign class. We now discuss each case in detail.

The first cluster of NXDomains falsely reported by Pleiades were random domain names generated by Chrome [16,45]. Each time the Google Chrome browser starts, it will query three “random looking” domain names. These domain names are issued as a DNS check, so the browser can determine if NXDomain rewriting is enabled. The “Chrome DGA” was reported as a variant of Bobax from Pleiades. We trained a class for this DGA and flagged it as benign. One more case of testing for NXDomain rewriting was identified in a brand of wireless access points. Connectify³, offers wireless hot-spot functionality and one of their configuration option enables the user to hijack the ISP’s default NXDomain rewriting service. The device generates a fixed number of NXDomains to test for rewriting.

Two additional cases of false reports were triggered by domain names from the .it and .edu TLDs. These domain names contained minor variations on common words (i.e. repubblica, gazzetta, computer, etc.). Domain names that matched these clusters appeared only for two days in our traces and never again. The very short lived presence of these two clusters could be explained if the domain names were part of a spam-campaign that was remediated by authorities before it became live.

The fifth case of false report originated from domain names under a US government zone and contained the

³www.connectify.me

Table 3: TPs (%) for C&C detection (1,000 training sequences).

botnet	FPs (%)					
	0.1	0.5	1	3	5	10
Zeus.v3	99.9	99.9	99.9	99.9	99.9	99.9
Expiro.Z	33.03	64.56	78.23	91.77	95.23	98.67
Bamital	100	100	100	100	100	100
Shiz	0	1.64	21.02	96.58	100	100
Boonana	3.8	10.69	15.59	27.67	35.05	48.43
BankPatch	56.21	70.77	93.18	99.9	99.91	99.94

string `wpdhsm`. Our best guess is that these are internal domain names that were accidentally leaked to the recursive DNS server of our ISP. Domain names from this cluster appeared only for one day. This class of NXDomains was also modeled as a benign variant. It is worth noting that all falsely reported DGA clusters, excluding the Chrome cluster, were short lived. If operators are willing to wait a few days until a new DGA cluster is reported by Pleiades, these false alarms would not have been raised.

7.3 C&C Detection

To evaluate the effectiveness of the *C&C Detection*, we proceeded as follows. We considered the six new DGAs which we were able to attribute to specific malware, as shown in Table 3. Let NX_i be the set of NXDomains collected by the *DGA Discovery* (Section 4) and *DGA Modeling* (Section 5.1) modules for the i -th DGA. For each DGA, we set aside a subset $NX_i^{train} \subset NX_i$ of NXDomains to train an HMM_i model. Then we use the remaining $NX_i^{test} = NX_i - NX_i^{train}$ to compute the true positive (TP) rate of HMM_i , and a set A that consists of 602,969 unique domain names related to the consistently popular domain names according to `alexa.com` to compute the false positive (FP) rate. To obtain A we first consider all domain names that have been consistently ranked in the top 100,000 popular domains by `alexa.com` for approximately one year. This gave us a set T of about 60,000 “stable” popular domain names, which we consider as *legitimate* domains. Then, we monitored the stream of successful DNS queries in a large live network for a few hours, and we added to A all the domain names whose effective 2LD is in T .

We performed experiments with a varying number $c = |NX_i^{train}|$ of training samples. Specifically, we set c equal to 100, 200, 500, 1,000, 2,000, 5,000, and 10,000. We then computed the trade-off between TPs and FPs for different detection thresholds. In the interest of space, we report only the results for $c=1,000$ in Table 3. In general, the results improve for increasing numbers of training instances. We set the detection threshold so as to obtain an FP rate equal to 0.1%, 0.5%, 1%, 3%, 5%, and 10%. As we can see, at FP=1% we obtained a high (> 93%) TP rate for three out of six DGAs, and relatively good results

(> 78%) in five out of six cases. At FP=3% we have high TP rate (> 91%) in five out of six cases.

As mentioned in Section 3, the *C&C Detection* module reduces the set of domain names successfully resolved by a host h that have been labeled as compromised with DGA-malware to a smaller set of good *candidate C&C domains* generated by the DGA. The results in Table 3 show that if we rank the domains resolved by h according to the likelihood assigned by the HMM, in most cases we will only need to inspect between 1/100 to 3/100 of the active domains queried by h to discover the C&C.

7.4 Case Studies

7.4.1 Zeus.v3

In September 2011, Pleiades detected a new DGA that we linked to the Zeus.v3 variant a few weeks later. The domain names collected from the machines compromised by this DGA-malware are hosted in six different TLDs: `.biz`, `.com`, `.info`, `.net`, `.org` and `.ru`. Excluding the top level domains, the length of the domain names generated by this DGA are between 33 and 45 alphanumeric characters. By analyzing one sample of the malware⁴ we observed that its primary C&C infrastructure is P2P-based. If the malware fails to reach its P2P C&C network, it follows a contingency plan, where a DGA-based component is used to try to recover from the loss of C&C communication. The malware will then resolve pseudo-random domain names, until an active C&C domain name is found.

To date, we have discovered 12 such C&C domains. Over time, these 12 domains resolved to five different C&C IPs hosted in four different networks, three in the US (AS6245, AS16626 and AS3595) and one in the United Kingdom (AS24931). Interestingly, we observed that the UK-based C&C IP address remained active for a very short period of time of only a few minutes, from Jan 25, 2012 12:14:04 EST to Jan 25, 2012 12:22:37 EST. The C&C moved from a US IP (AS16626) to the UK (AS24931), and then almost immediately back to the US (AS3595).

7.4.2 BankPatch

We picked the BankPatch DGA cluster as a sample case for analysis since this botnet had been active for several months during our experiments and the infected population continues to be significant. The C&C infrastructure that supports this botnet is impressive. Twenty six different clusters of servers acted as the C&Cs for this botnet. The botnet operators not only made use of a DGA but also moved the active C&Cs to different networks every few weeks (on average). During our C&C

⁴Sample MD5s: 8f60afa9ea1e761edd49dfe012c22cbf and cccc69613c71d66f98abe9cc7e2e20ef.

discovery process, we observed IP addresses controlled by a European CERT. This CERT has been taking over domain names from this botnet for several months. We managed to cross-validate with them the completeness and correctness of the C&C infrastructure. Complete information about the C&C infrastructure can be found in Table 4.

The actual structure of the domain name used by this DGA can be separated into a four byte prefix and a suffix string argument. The suffix string arguments we observed were: `seapollo.com`, `tomvader.com`, `aulmala.com`, `apon-tis.com`, `fnomosk.com`, `erhogeld.com`, `erobots.com`, `ndsontex.com`, `rte-hedel.com`, `nconnect.com`, `edsafe.com`, `berhogeld.com`, `musallied.com`, `newnacion.com`, `susaname.com`, `tvolveras.com` and `dminmont.com`.

The four bytes of entropy for the DGA were provided by the prefix. We observe collisions between NXDomains from different days, especially when only one suffix argument was active. Therefore, we registered a small sample of ten domain names at the beginning of 2012 in an effort to obtain a glimpse of the overall distribution of this botnet. Over a period of one month of monitoring the sink-holed data from the domain name of this DGA, this botnet has infected hosts in 270 different networks distributed across 25 different countries. By observing the recursive DNS servers from the domain names we sinkholed, we determined 4,295 were located in the US. The recursives we monitored were part of this list and we were able to measure 86 infected hosts (on average) in the network we were monitoring. The five countries that had the most DNS resolution requests for the sinkholed domain names (besides the US) were Japan, Canada, the United Kingdom and Singapore. The average number of recursive DNS servers from these countries that contacted our authorities was 22 — significantly smaller than the volume of recursive DNS servers within the US.

8 Discussion and Limitations

Pleiades has some limitations. For example, once a new DGA is discovered, Pleiades can build fairly accurate statistical models of how the domains generated by the DGA “look like”, but it is unable to learn or reconstruct the exact domain generation algorithm. Therefore, Pleiades will generate a certain number of false positives and false negatives. However, the results we presented in Table 1 show that Pleiades is able to construct a very accurate *DGA Classifier* module, which produces very few false positives and false negatives for $\alpha = 10$. At the same time, Table 3 shows that the *C&C Detection* module, which attributes a single active domain name to a given DGA, and also works fairly well in the ma-

Table 4: C&C Infrastructure for BankPatch.

IP addresses	CC	Owner
146.185.250.{89-92}	RU	Petersburg Int.
31.11.43.{25-26}	RO	SC EQUILIBRIUM
31.11.43.{191-194}	RO	SC EQUILIBRIUM
46.16.240.{11-15}	UA	iNet Colocation
62.122.73.{11-14,18}	UA	“Leksim” Ltd.
87.229.126.{11-16}	HU	Webenlet Kft.
94.63.240.{11-14}	RO	Com Frecatei
94.199.51.{25-18}	HU	NET23-AS 23VNET
94.61.247.{188-193}	RO	Vatra Luminoasa
88.80.13.{111-116}	SE	PRQ-AS PeRiQuito
109.163.226.{3-5}	RO	VOXILITY-AS
94.63.149.{105-106}	RO	SC CORAL IT
94.63.149.{171-175}	RO	SC CORAL IT
176.53.17.{211-212}	TR	Radore Hosting
176.53.17.{51-56}	TR	Radore Hosting
31.210.125.{5-8}	TR	Radore Hosting
31.131.4.{117-123}	UA	LEVEL7-AS IM
91.228.111.{26-29}	UA	LEVEL7-AS IM
94.177.51.{24-25}	UA	LEVEL7-AS IM
95.64.55.{15-16}	RO	NETSERV-AS
95.64.61.{51-54}	RO	NETSERV-AS
194.11.16.133	RU	PIN-AS Petersburg
46.161.10.{34-37}	RU	PIN-AS Petersburg
46.161.29.102	RU	PIN-AS Petersburg
95.215.{0-1}.29	RU	PIN-AS Petersburg
95.215.0.{91-94}	RU	PIN-AS Petersburg
124.109.3.{3-6}	TH	SERVENET-AS-TH-AP
213.163.91.{43-46}	NL	INTERACTIVE3D-AS
200.63.41.{25-28}	PA	Panamaserver.com

jority of cases. Unfortunately, there are some scenarios in which the HMM-based classification has difficulties. We believe this is because our HMM considers domain names simply to be sequences of individual characters. In our future work, we plan to experiment with 2-grams, whereby a domain name will be seen as a sequence of pairs of characters, which may achieve better classification accuracy for the harder to model DGAs.

For example, our HMM-based detector was unable to obtain high true positive rates on the Boonana DGA. The reason is that the Boonana DGA leverages third-level pseudo-random domain names under several second-level domains owned by *dynamic DNS* providers. During our evaluation, the hosts infected with Boonana contacted DGA-generated domain names under 59 different effective second-level domains. We believe that the high variability in the third-level domains and the high number of effective 2LDs used by the DGA make it harder to build a good HMM, thus causing a relatively low number of true positives. However, in a real-world deployment scenario, the true positive rate may be significantly increased by focusing on the dynamic DNS domains queried by the compromised hosts. For example, since we know that Boonana only uses dynamic DNS domains, we can filter out any other NXDomains, and avoid passing them to the HMM. In this scenario the

HMM would receive as an input only dynamic DNS domains, which typically represent a fraction of all active domains queried by each host, and consequently the absolute number of false positives can be significantly reduced.

As we mentioned in Section 3, detecting active DGA-generated C&C domains is valuable because their resolved IP addresses can be used to update a C&C IP blacklist. In turn, this IP blacklist can be used to block C&C communications at the network edge, thus providing a way to mitigate the botnet’s malicious activities. Clearly, for this strategy to be successful, the frequency with which the C&C IP addresses change should be lower than the rate with which new pseudo-random C&C domain names are generated by the DGA. This assumption holds for all practical cases of DGA-based malware we encountered. After all, the generation of pseudo-random domains mainly serves the purpose of making the take-down of loosely centralized botnets harder. However, one could imagine “hybrid” botnets that use DGA-generated domains to identify a set of peer IPs to bootstrap into a P2P-based C&C infrastructure. Alternatively, the DGA-generated C&C domains may be flux domains, namely domain names that point to a IP fluxing network. It is worth noting that such sophisticated “hybrid” botnets may be quite complex to develop, difficult to deploy, and hard to manage successfully.

Another potential limitation is due to the fact that Pleiades is not able to distinguish between different botnets whose bot-malware use the same DGA algorithm. In this case, while the two botnets may be controlled by different entities, Pleiades will attribute the compromised hosts within the monitored network to a single DGA-based botnet.

One limitation of our evaluation method is the exact enumeration of the number of infected hosts in the ISP network. Due to the location of our traffic monitoring sensors (below the recursive DNS server), we can only obtain a lower bound estimate on the number of infected hosts. This is because we have visibility of the IP addresses within the ISP that generate the DNS traffic, but lack additional information about the true number of hosts “behind” each IP. For example, an IP address that generates DNS traffic may very well be a NAT, firewall, DNS server or other type of complex device that behaves as a proxy (or relay point) for other devices. Also, according to the ISP, the DHCP churn rate is relatively low, and it is therefore unlikely that we counted the same internal host multiple times.

In the case of Zeus.v3, the DGA is used as a backup C&C discovery mechanism, in the event that the P2P component fails to establish a communication channel with the C&C. The notion of having a DGA component as a redundant C&C discovery strategy could be

used in the future by other malware. A large number of new DGAs may potentially have a negative impact on the supervised modules of Pleiades, and especially on the HMM-based C&C detection. In fact, a misclassification by the *DGA Classifier* due to the large number of classes among which we need to distinguish may misguide the selection of the right HMM to be used for C&C detection, thus causing an increase in false positives. In our future work we plan to estimate the impact of such misclassifications on the C&C detection accuracy, and investigate whether using auxiliary IP-based information (e.g., IP reputation) can significantly improve the accuracy in this scenario.

As the internals of our system become public, some botnets may attempt to evade both the DGA discovery and C&C detection process. As we have already discussed, it is in the malware authors’ best interest to create a high number of DGA-related NXDomains in order to make botnet take-over efforts harder. However, the malware could at the same time generate NXDomains not related with the C&C discovery mechanism in an effort to mislead our current implementation of Pleiades. These *noisy* NXDomains may be generated in two ways: (1) randomly, for example by employing a different DGA, or (2) by using one DGA with two different seeds, one of which is selected to generate noise. In case of (1), the probability that they will be clustered together is small. This means that these NXDomains will likely not be part of the final cluster correlation process and they will not be reported as new DGA-clusters. On the other hand, case (2) might cause problems during learning, especially to the HMM, because the noisy and “true” NXDomains may be intermixed in the same cluster, thus making it harder to learn an accurate model for the domain names.

9 Conclusion

In this paper, we presented a novel detection system, called Pleiades, that is able to accurately detect machines within a monitored network that are compromised with DGA-based bots. Pleiades monitors traffic below the local recursive DNS server and analyzes streams of unsuccessful DNS resolutions, instead of relying on manual reverse engineering of bot malware and their DGA algorithms. Using a multi-month evaluation phase, we showed that Pleiades can achieve very high detection accuracy. Moreover, over the fifteen months of the operational deployment in a major ISP, Pleiades was able to identify six DGAs that belong to known malware families and six new DGAs never reported before.

References

- [1] K. Aas and L. Eikvil. Text categorisation: A survey., 1999.
- [2] abuse.ch. ZeuS Gets More Sophisticated Using P2P Techniques. <http://www.abuse.ch/?p=3499>, 2011.
- [3] M. Antonakakis, R. Perdisci, D. Dagon, W. Lee, and N. Feamster. Building a dynamic reputation system for DNS. In *the Proceedings of 19th USENIX Security Symposium (USENIX Security '10)*, 2010.
- [4] M. Antonakakis, R. Perdisci, W. Lee, N. Vasiloglou, and D. Dagon. Detecting malware domains in the upper DNS hierarchy. In *the Proceedings of 20th USENIX Security Symposium (USENIX Security '11)*, 2011.
- [5] BankPatch. Trojan.Bankpatch.C. http://www.symantec.com/security_response/writeup.jsp?docid=2008-081817-1808-99&tabid=2, 2009.
- [6] L. Bilge, E. Kirda, C. Kruegel, and M. Balduzzi. EXPOSURE: Finding malicious domains using passive dns analysis. In *Proceedings of NDSS*, 2011.
- [7] R. Feldman and J. Sanger. *The text mining handbook: advanced approaches in analyzing unstructured data*. Cambridge Univ Pr, 2007.
- [8] R. Finones. Virus:Win32/Expiro.Z. <http://www.microsoft.com/security/portal/Threat/Encyclopedia/Entry.aspx?Name=Virus%3AWin32%2FExpiro.Z>, 2011.
- [9] Y. Freund and L. Mason. The alternating decision tree learning algorithm. In *Proceedings of the Sixteenth International Conference on Machine Learning, ICML '99*, 1999.
- [10] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. BotHunter: Detecting malware infection through IDS-driven dialog correlation. In *Proc. USENIX Security*, 2007.
- [11] M. Geide. Another trojan bamital pattern. <http://research.zscaler.com/2011/05/another-trojan-bamital-pattern.html>, 2011.
- [12] S. Golovanov and I. Soumenkov. TDL4 top bot. http://www.securelist.com/en/analysis/204792180/TDL4_Top_Bot, 2011.
- [13] G. Gu, R. Perdisci, J. Zhang, and W. Lee. Bot-Miner: clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *USENIX Security*, 2008.
- [14] G. Gu, J. Zhang, and W. Lee. BotSniffer: Detecting botnet command and control channels in network traffic. In *Network and Distributed System Security Symposium (NDSS)*, 2008.
- [15] J. Hermans. MozillaWiki TLD List. https://wiki.mozilla.org/TLD_List, 2006.
- [16] S. Krishnan and F. Monrose. Dns prefetching and its privacy implications: when good things go bad. In *Proceedings of the 3rd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more, LEET'10*, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.
- [17] L. I. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. Wiley-Interscience, 2004.
- [18] M. H. Ligh, S. Adair, B. Hartstein, and M. Richard. *Malware Analyst's Cookbook and DVD*, chapter 12. Wiley, 2010.
- [19] P. Mockapetris. Domain names - concepts and facilities. <http://www.ietf.org/rfc/rfc1034.txt>, 1987.
- [20] P. Mockapetris. Domain names - implementation and specification. <http://www.ietf.org/rfc/rfc1035.txt>, 1987.
- [21] M. Newman. *Networks: an introduction*. Oxford University Press, 2010.
- [22] A. Y. Ng, M. I. Jordan, and Y. Weiss. On spectral clustering: Analysis and an algorithm. In *Advances In Neural Information Processing Systems*, pages 849–856. MIT Press, 2001.
- [23] P. Porras, H. Saidi, and V. Yegneswaran. An analysis of conficker's logic and rendezvous points. <http://mtc.sri.com/Conficker/>, 2009.
- [24] D. Pelleg and A. W. Moore. X-means: Extending k-means with efficient estimation of the number of clusters. In *Proceedings of the Seventeenth International Conference on Machine Learning, ICML '00*, pages 727–734, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.
- [25] C. POLSKA. ZeuS P2P+DGA variant mapping out and understanding the threat.

- http://www.cert.pl/news/4711/langswitch_lang/en, 2012.
- [26] P. Porras. Inside risks: Reflections on conficker. *Communications of the ACM*, 52:23–24, October 2009.
- [27] P. Porras, H. Saidi, and V. Yegneswaran. Conficker C analysis. Technical report, SRI International, Menlo Park, CA, April 2009.
- [28] L. R. Rabiner. Readings in speech recognition. chapter A tutorial on hidden Markov models and selected applications in speech recognition. 1990.
- [29] P. Royal. Analysis of the kraken botnet. http://www.damballa.com/downloads/r_pubs/KrakenWhitepaper.pdf, 2008.
- [30] S. Shevchenko. Srizbi domain generator calculator. <http://blog.threatexpert.com/2008/11/srizbis-domain-calculator.html>, 2008.
- [31] S. Shevchenko. Domain name generator for murofet. <http://blog.threatexpert.com/2010/10/domain-name-generator-for-murofet.html>, 2010.
- [32] SOPHOS. Mal/Simda-C. <http://www.sophos.com/en-us/threat-center/threat-analyses/viruses-and-spyware/Mal-Simda-C/detailed-analysis.aspx>, 2012.
- [33] J. Stewart. Bobax trojan analysis. <http://www.secureworks.com/research/threats/bobax/>, 2004.
- [34] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: analysis of a botnet takeover. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 635–647, New York, NY, USA, 2009. ACM.
- [35] S. Stover, D. Dittrich, J. Hernandez, and S. Dietrich. Analysis of the storm and nugache trojans: P2P is here. In *USENIX ;login.*, vol. 32, no. 6, December 2007.
- [36] T.-F. Yen and M. K. Reiter. Are your hosts trading or plotting? Telling P2P file-sharing and bots apart. In *ICDCS*, 2010.
- [37] R. Villamarin-Salomon and J. Brustoloni. Identifying botnets using anomaly detection techniques applied to dns traffic. In *5th Consumer Communications and Networking Conference*, 2008.
- [38] Wikipedia. The storm botnet. http://en.wikipedia.org/wiki/Storm_botnet, 2010.
- [39] J. Williams. What we know (and learned) from the waledac takedown. <http://tinyurl.com/7apnn9b>, 2010.
- [40] J. Wolf. Technical details of srizbi's domain generation algorithm. <http://blog.fireeye.com/research/2008/11/technical-details-of-srizbis-domain-generation-algorithm.html>, 2008. Retrieved: April, 10 2010.
- [41] J. Wong. Trojan:Java/Boonana. <http://www.microsoft.com/security/portal/Threat/Encyclopedia/Entry.aspx?Name=Trojan%3AJava%2FBoonana>, 2011.
- [42] S. Yadav, A. K. K. Reddy, A. N. Reddy, and S. Ranjan. Detecting algorithmically generated malicious domain names. In *Proceedings of the 10th annual Conference on Internet Measurement*, IMC '10, pages 48–61, New York, NY, USA, 2010. ACM.
- [43] S. Yadav and A. N. Reddy. Winning with dns failures: Strategies for faster botnet detection. In *7th International ICST Conference on Security and Privacy in Communication Networks*, 2011.
- [44] T.-F. Yen and M. K. Reiter. Traffic aggregation for malware detection. In *Proc. International conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2008.
- [45] B. Zdrnja. Google Chrome and (weird) DNS requests. <http://isc.sans.edu/diary/Google+Chrome+and+weird+DNS+requests/10312>, 2011.
- [46] J. Zhang, R. Perdisci, W. Lee, U. Sarfraz, and X. Luo. Detecting stealthy P2P botnets using statistical traffic fingerprints. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Dependable Computing and Communication Symposium*, 2011.

PUBCRAWL: Protecting Users and Businesses from CRAWLers

Gregoire Jacob

University of California, Santa Barbara / Telecom SudParis

gregoire.jacob@gmail.com

Christopher Kruegel

University of California, Santa Barbara

chris@cs.ucsb.edu

Engin Kirda

Northeastern University

ek@ccs.neu.edu

Giovanni Vigna

University of California, Santa Barbara

vigna@cs.ucsb.edu

Abstract

Web crawlers are automated tools that browse the web to retrieve and analyze information. Although crawlers are useful tools that help users to find content on the web, they may also be malicious. Unfortunately, unauthorized (malicious) crawlers are increasingly becoming a threat for service providers because they typically collect information that attackers can abuse for spamming, phishing, or targeted attacks. In particular, social networking sites are frequent targets of malicious crawling, and there were recent cases of scraped data sold on the black market and used for blackmailing.

In this paper, we introduce PUBCRAWL, a novel approach for the detection and containment of crawlers. Our detection is based on the observation that crawler traffic significantly differs from user traffic, even when many users are hidden behind a single proxy. Moreover, we present the first technique for crawler campaign attribution that discovers synchronized traffic coming from multiple hosts. Finally, we introduce a containment strategy that leverages our detection results to efficiently block crawlers while minimizing the impact on legitimate users. Our experimental results in a large, well-known social networking site (receiving tens of millions of requests per day) demonstrate that PUBCRAWL can distinguish between crawlers and users with high accuracy. We have completed our technology transfer, and the social networking site is currently running PUBCRAWL in production.

1 Introduction

Web crawlers, also called spiders or robots, are tools that browse the web in an automated and systematic fashion. Their purpose is to retrieve and analyze information that is published on the web. Crawlers were originally developed by search engines to index web pages, but have since multiplied and diversified. Crawlers are now used as link checkers for web site verification, as scrapers to harvest content, or as site analyzers that process the collected data for analytical or archival purposes [9]. While many crawlers are legitimate and help users find relevant

content on the web, unfortunately, there are also crawlers that are deployed for malicious purposes.

As an example, cyber-criminals perform large-scale crawls on social networks to collect user profile information that can later be sold to online marketers or used for targeted attacks. In 2009, a hacker who had crawled the popular student network StudiVZ, blackmailed the company, threatening to sell the stolen data to gangs in Eastern Europe [23]. In 2010, Facebook sued an entrepreneur who crawled more than 200 million profiles, and who was planning to create a third-party search service with the data that he had collected [25]. In general, the problem of site scraping is not limited to social networks. Many sites who advertise goods, services, and prices online desire protection against competitors that use crawlers to spy on their inventory. In several court cases, airlines (e.g., American Airlines [2], Ryanair [20]) sued companies that scraped the airlines' sites to be able to offer price comparisons and flights to their customers. In other cases, attackers scraped content from victim sites, and then simply offered the cloned information under their own label.

Many websites explicitly forbid unauthorized scraping in their terms of services. Unfortunately, such terms are simply ignored by non-cooperating (malicious) crawlers. The *Robot Exclusion Protocol* faces a similar problem: Web sites can specify rules in the `robots.txt` file to limit crawler accesses to certain parts of their site [14], but a crawler has to voluntarily follow these restrictions.

Current detection techniques rely on simple crawler artifacts (e.g., spurious user agent strings, suspicious referrers) and simple traffic patterns (e.g., inter-arrival time, volume of traffic) to distinguish between human and crawler traffic. Unfortunately, better crawler implementations can remove revealing artifacts, and simple traffic patterns fail in the presence of proxy servers or large corporate gateways, which can serve hundreds of legitimate users from a single IP address. In response to the perceived lack of effective protection, several commercial anti-scraping services have emerged (e.g., *Dis-*

til.It, *SiteBlackBox*, *Sentor Assassin*). These services employ “patent pending heuristics” to defend against unwanted crawlers. Unfortunately, it is not clear from available descriptions how these services work in detail.

Many sites rely on CAPTCHAs [24] to prevent scrapers from accessing web content. CAPTCHAs use challenge-response tests that are easy to solve for humans but hard for computers. Some tests are known to be vulnerable to automated breaking techniques [4]. Nevertheless, well-designed CAPTCHAs offer a reasonable level of protection against automated attacks. Unfortunately, their excessive use brings along usability problems and severely decreases user satisfaction. Other prevention techniques are crawler traps. A crawler trap is a URL that lures crawlers into infinite loops, using, for example, symbolic links or sets of auto-generated pages [15]. Unfortunately, legitimate crawlers or users may also be misled by these traps. Traps and CAPTCHAs can only be one part of a successful defense strategy, and legitimate users and crawlers must be exposed as little as possible to them.

In this paper, we introduce a novel approach and a system called PUBCRAWL to detect crawlers and automatically configure a defense strategy. PUBCRAWL’s usefulness has been confirmed by a well-known, large social networking site we have been collaborating with, and it is now being used in production. Our detection does not rely on easy-to-detect artifacts or the lack of fidelity to web standards in crawler implementations. Instead, we leverage the key observation that crawlers are automated processes, and as such, their access patterns (web requests) result in different types of regularities and variations compared to those of real users. These regularities and variations form the basis for our detection.

For detection, we use both content-based and timing-based features to *passively* model the traffic from different sources. We extract content-based features from HTTP headers (e.g., referrers, cookies) and URLs (e.g., page revisits, access errors). These features are checked by heuristics to detect values betraying a crawling activity. For timing-based features, we analyze the time series produced by the stream of requests. We then use machine learning to train classifiers that can distinguish between crawler and user traffic. Our system is also able to identify crawling campaigns led by distributed crawlers by looking at the synchronization of their traffic.

The aforementioned features work well for detecting crawlers that produce a minimum volume of traffic. However, it is conceivable that some adversaries have access to a large botnet with hundreds of thousands of infected machines. In this case, each individual bot would only need to make a small number of requests to scrape the entire site, possibly staying below the minimal volume required by our models. An *active* response

to such attacks must be triggered, such as the injection of crawler traps or CAPTCHAs. An active response is only triggered when a single client sends more than a (small) number of requests. To minimize the impact of active responses on legitimate users, we leverage our detection results to distinguish between malicious crawlers and benign sources that produce a lot of traffic (e.g., proxy servers or corporate gateways). This allows us to automatically whitelist benign sources (whose IP addresses rarely change), minimizing the impact on legitimate users while denying access to unwanted crawlers.

For evaluation, we applied PUBCRAWL to the web server logs of the social networking site we were working with. To train the system, we examined 5 days worth of traffic, comprising 73 million requests from 813 average-volume sources (IP addresses). The logs were filtered to focus on sources whose traffic volume was not an obvious indicator of their type. To test the system, we examined a set of 62 million requests coming from 763 sources over 5 days. Our experiments demonstrated that more sophisticated crawlers are often hard to distinguish from real users, and hence, are difficult to detect using traditional techniques. Using our system, we were able to identify crawlers with high accuracy, including crawlers that were previously-unknown to the social networking site. We also identified interesting campaigns of distributed crawlers.

Section 2 gives an overview of the system whereas Sections 3 to 5 provide more technical details for each part. The configuration and evaluation of the system is finally addressed in Sections 6 to 8. Overall, this paper makes the following contributions:

- We present a novel technique to detect individual crawlers by time series analysis. To this end, we use auto-correlation and decomposition techniques to extract navigation patterns from the traffic of individual sources.
- We introduce the first technique to detect distributed crawlers (crawling campaigns). More precisely, our system can identify coordinated activity from multiple crawlers.
- We contain crawlers using active responses that we strategically emit according to detection results.
- We implemented our approach in a tool called PUBCRAWL, and performed the largest real-world crawler detection evaluation to date, using tens of millions of requests from a popular social network. PUBCRAWL distinguishes crawlers from users with high accuracy (even users behind a proxy).

2 System Overview

The goal of PUBCRAWL is to analyze the web traffic that is sent to a destination site, and to automatically classify

the originating sources of this traffic as either crawlers or users. The initial traffic analysis is passive and performed off-line, using log data that records web site requests from clients. The goal of this analysis is to build a knowledge base about traffic sources (IP addresses).

This knowledge base is then consulted to respond to web requests. Requests from known users or accepted crawlers (e.g., *Googlebot*) are served directly. Requests from unwanted crawlers, in contrast, are blocked. Of course, the knowledge base may not contain an entry for a source IP. In this case, a small number of requests is permitted until it exceeds a given threshold: the system then switches to active containment and, for example, injects traps or CAPTCHAs into the response.

While the system is active, the requests are recorded to refine the knowledge base: When PUBCRAWL identifies a previously-unknown source to be a user or a legitimate proxy, requests from this source are no longer subjected to active responses whereas unwanted crawlers are blacklisted. The key insight is that PUBCRAWL can successfully identify legitimate, high-volume sources, and these sources are very stable. This stability of high-volume sources and the large number of low-volume sources (tens of requests) ensure that only a small fraction of users will be subjected to active responses.

The architecture of PUBCRAWL is shown in Figure 1. The server log entries (i.e., the input) are first split into time windows of fixed length. Running over these windows, the system extracts two kinds of information: (i) HTTP header information, including URLs and (ii) timing information in the form of time series. The extracted information is then submitted to two detection modules (which detect individual crawlers) and an attribution module (which detects crawling campaigns). These three modules generate the knowledge base that is leveraged by the proactive containment module for real-time traffic. The different modules are described below:

Heuristic detection. For a given time window, the system analyzes the requests coming from distinct source IP addresses, and extracts different features related to HTTP header fields and URL elements. These features are checked with heuristics to detect suspicious values that could reveal an ongoing crawling activity (e.g., suspicious referrers, unhandled cookies, etc.).

Traffic shape detection. When crawlers correctly set the different HTTP fields and masquerade the user agent string, it becomes much more difficult to tell apart slow crawlers from busy proxies since they cannot be distinguished based on request volumes. Traffic shape detection addresses this problem. For a given source IP address, the system analyzes the requests (time stamps) over a fixed time window to build the associated time series. Figure 2 depicts time series representing crawler

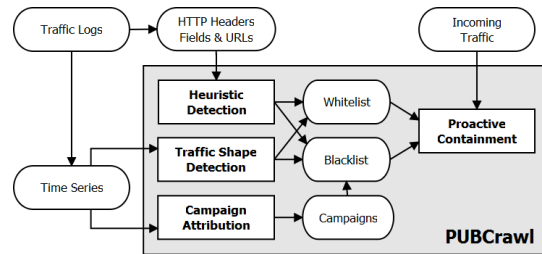


Figure 1: Architecture overview

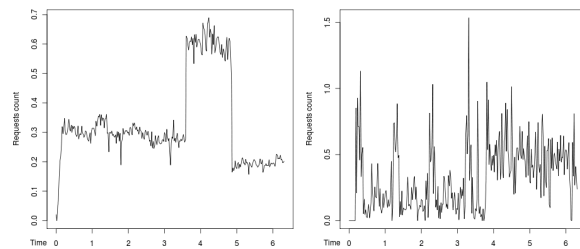


Figure 2: YahooSlurp and MSIE 7 time series

and user traffic. One can observe distinctive patterns that are specific to each class of traffic. Crawler traffic tends to exhibit more regularity and stability over time. User traffic, instead, tends to exhibit more local variations. However, over time, user traffic also shows repetitive patterns whose regularity is related to the “human time scale” (e.g., daily or weekly patterns).

To determine the long-term stability of a time series with respect to its quick variations, we leverage auto-correlation analysis techniques. Furthermore, to separate repetitive patterns from slow variations, we leverage decomposition analysis techniques. Decomposition separates a time series into a trend component that captures slow variations, a seasonal component that captures repetitive patterns, and a component that captures the remaining noise. We use the information extracted from these analyses as input to classifiers for detection. These classifiers are used to determine whether an unknown time series belongs to a crawler or a user.

Campaign attribution. Some malicious crawlers willingly reduce their volume of requests to remain stealthy. This comes at a cost for the attacker in terms of crawling efficiency (volume of retrieved data during a specific time span). To compensate for these limitations, malicious crawlers, just like legitimate ones, distribute their crawling activity over multiple sources. We denote a set of crawlers, distributed over multiple locations, and showing synchronized activity, as a *crawling campaign*.

Figure 3 presents two time series that correspond to the same crawler distributed over two hosts in different subnets. One can observe a high level of similarity between the two time series. The key insight of our approach is that these similarities can be used to identify the distributed crawlers that are part of a campaign.

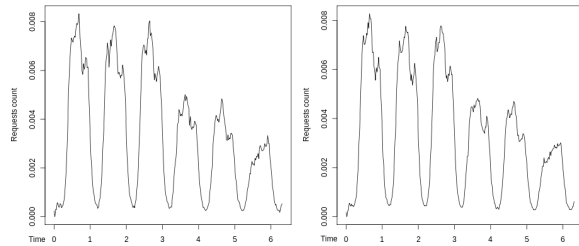


Figure 3: Distributed crawler (sources A and B)

Proactive containment. Our containment approach uses the detection results to establish a whitelist of legitimate crawlers and user sources that are allowed direct access to the site. We also compile a blacklist of unauthorized crawlers that need to be blocked. As mentioned previously, other sources are granted a small number of unrestricted accesses per day. For IPs that exceed this threshold, the system responds by inserting CAPTCHAs or crawler traps into response pages.

3 Crawler Detection Approach

In this section, we provide a more detailed description of the crawler detection process in PUBCRAWL.

3.1 Heuristic detection based on headers

The *heuristic detection* module processes, for each source, the HTTP header fields and URLs extracted from the requests in the traffic log. Request-based features have been used in the past by systems that aim to detect crawlers [10, 11, 17, 18, 21, 22]. We selected the following ones for our detection module:

- *High error rate:* The URL lists used to feed a crawler often contain entries that belong to invalid profile names. As a result, crawlers tend to show a higher rate of errors when accessing profile pages.
- *Low page revisit:* Crawlers tend to avoid revisiting the same pages. Users, on the other hand, tend to regularly revisit the same profiles in their network.
- *Suspicious referrer:* Many crawlers ignore the referrer field in a request, setting it to null instead. Advanced crawlers do handle the referrer field, but give themselves away by using referrers that point to the results of directory queries (listings).
- *Ignored cookies:* Many crawlers ignore cookies. As a result, a new session identifier cookie is issued for each request by these crawlers.

In addition to the previously-described heuristics, we propose a number of additional, novel features:

- *Unbalanced traffic:* When we see multiple user agent strings coming from one IP, we expect the requests to be somewhat balanced between these different user agents. If one agent is responsible for more than 99% of the requests, this is suspicious.

- *Low parameter usage:* Existing detectors mostly consider the length and depth of URLs. In our case, profile URLs show a similar length and the same level of depth. Instead, parameters can additionally be appended to URLs (e.g., language selection). Crawlers typically do not use these parameters.
- *Suspicious profile sequence:* Crawlers often access profiles in a sorted (alphabetic) order. This is because the pointers to profiles are often obtained from directory queries.

Heuristics results are combined into a final classification by a majority vote. That is, if a majority of heuristics are triggered, we flag the source as a crawler.

3.2 Time series extraction

Our traffic shape detection significantly differs from existing work on crawler detection as we do not divide the source traffic into sessions. Instead, we model traffic as counting processes from which some properties are extracted, replacing the simple timing features (e.g., mean and deviation of inter-arrival times) computed over traffic sessions as in [18, 21, 22].

Deriving the time series. We model the traffic from a source over a time window $[t_0, t_n]$ as a counting process (a particular kind of time series). A counting process $X(t) = \{X(t)\}_{t_0}^{t_n}$ counts the number of requests that arrive from a given source within n time intervals of fixed duration, where each interval is a fraction of the entire time window. Notice that the traffic logs must be split into windows of at least two days to capture patterns that repeat at the time scale of one day.

Because most statistical tests are sensitive to the total (absolute) numbers of requests, we normalize the time series. To this end, the amplitude of the series is first scaled to capture the ratio of requests per time interval to the total volume of requests produced by the source. The time frame of the series is then normalized by setting a common time origin: the start of all series is set to be the arrival time of the *first* observed request *among all* monitored sources. Formally, the normalized time series are extracted as follows: Let S be the set of monitored sources. Let R_s be the set of requests from a source $s \in S$. Then, its time series $X_s(t)$ is:

$$X_s(t) = \left\{ \frac{\text{Card}(\{r \in R_s \mid r.\text{arrival} \in [t, t + d[\})}{\text{Card}(\{r \in R_s \mid r.\text{arrival} \in [t_0, t_n[\})} \right\} \quad (1)$$

$$t_0 = \min_{s \in S} (\{\min_{r \in R_s} (\{r.\text{arrival}\})\}) \quad (2)$$

We chose $d = 30$ minutes as the length of each time interval, in order to smooth the shape of the time series. Shorter intervals made the series too sensitive to perturbations in the network communications that are often independent of the source. Longer intervals, instead, make it harder to capture interesting variations in the traffic.

3.3 Detection by traffic shape analysis

In this section, we present how we model the shape of time series to distinguish users from crawlers. Sections 3.3.1 and 3.3.2 discuss how characteristic features of the traffic are extracted using the auto-correlation and decomposition analyses. Section 3.3.3 describes how these features are used to train classifiers with the goal of identifying crawler traffic.

3.3.1 Auto-correlation analysis

To characterize the stability of the source traffic, we compute the *Sample Auto-Correlation Function (SAC)* of the source’s time series and analyze its shape [3, Chapt.9]. The *SAC* captures the dependency of values at different points in time on the values observed for the process at previous times (the time difference between the two compared values is called *lag*). This function is a good indicator for how the request counts vary over time. A strong auto-correlation at small lags indicates a stable (regular) process, which is typical for crawlers. Spikes of auto-correlation at higher lags indicate potential seasonal variations, as in the case of users (for example, a strong auto-correlation at a lag of one day indicates that traffic follows a regular, daily pattern).

For a given lag k , the auto-correlation coefficient r_k is computed as in Equation 3, where E denotes the mean and Var the variance. The *SAC Function* captures the auto-correlation coefficients at different lags.

$$r_k = \frac{E[(X(t) - E[X(t)]) \times (X(t+k) - E[X(t)])]}{Var[X(t)]} \quad (3)$$

To determine the significance of the auto-correlation coefficient at a given lag k , the coefficient is usually compared to the standard error. If the coefficient is larger than twice the standard error, it is statistically significant. In this case, we say that we observe a *spike* at lag k . A spike indicates that counts separated by a lag k are linearly dependent. We use the Moving Average (MA) model to compute the standard error at lag k [3]. Unlike other models, the MA model does not assume that the values of the time series are uncorrelated, random variables. This is important, as we expect request counts from a single source to be correlated.

Figure 4 presents the *SAC Functions* computed over the time series from Figure 2. The functions were plotted over 96 lags (time span of two days). The additional (red) lines correspond to the standard errors under the MA assumption. If we observe the shape of these *SACs*, the crawler *SAC* shows a strong auto-correlation at small lags, followed by a slow linear decay, but no consecutive spike. The user *SAC* shows a less significant auto-correlation at small lags, followed by a fast exponential decay. However, we observe spikes at lags multiple of 0.5, corresponding to a half-daily and daily seasonality.

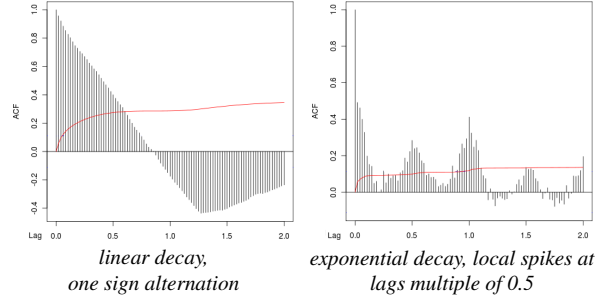


Figure 4: YahooSlurp and MSIE 7 SACs

Auto-correlation interpretation. Interpreting the shape of the *SAC* is traditionally a manual process, which is left to an analyst [3]. For our system, this process needs to be automated. To this end, we introduce three features to describe the properties of the *SAC*: speed of decay, sign alternation, and local spikes.

The *speed of decay* captures the short-term stability of the traffic. A slow decay indicates that the traffic is stable over longer periods whereas a fast decay indicates that there is little stability. The speed of decay feature can assume four values: linear decay, exponential decay, cut-off decay (coefficients reach a cliff and drop), no decay (coefficients comparable to random noise).

The *sign alternation* identifies how often the sign of the *SAC* changes. Its values are: no alternation, single alternation, oscillation, or erratic. No or single sign alternations are typical of crawlers, while user traffic potentially shows more alternations.

Local spikes reflect periodicity in the traffic. A local spike implies a repeated activity whose occurrences are separated by a fixed time difference (lag). This is typical for user traffic. This feature has two values: a discrete spikes count plus a Boolean value indicating if spikes are observed at interesting lags (half day, day).

Computing our features is a two-step process: First, we compute “runs” over the auto-correlation coefficients of the *SAC*. A run is a sequence of *zeroes* and *ones* for each lag k , where a *one* indicates that a particular property of interest holds. An auto-correlation coefficient is characterized by four properties: positive, significant, null, and larger than the previous value. Runs allow us to determine how often these properties change. This can be done by computing various statistics (e.g. mean, variance, length) over the related runs and their subruns (a sequence of consecutive, identical values).

In the second step, we apply a number of heuristics to the different runs. In particular, we compare the statistics computed for different runs with thresholds that indicate whether a certain property changes once, frequently, or never. The details of how we compute the runs, as well as the heuristics that we apply, are described in detail in Appendix A. The heuristics provide the actual feature values: speed of decay, sign alternation and local spikes.

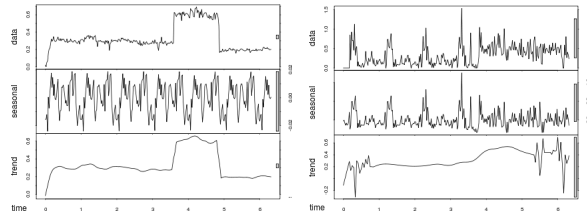


Figure 5: YahooSlurp and MSIE 7 decompositions

3.3.2 Decomposition analysis

The events that constitute a time series are the result of different factors that are not directly captured by the time series; only their impact is observed. Part of these factors slowly change over time. These factors generate slow shifts within the series that constitute its *trend* component $T(t)$. Another part of these factors repeat over time due to periodic events. These factors generate repetitive patterns within the series that constitute its *seasonal* component $S(t)$. Unexplained short-term, random effects constitute the remaining *noise* $R(t)$.

Decomposition aims to identify the three components of a time series such as $X(t) = T(t) + S(t) + R(t)$. The results of decomposition provide valuable insights into which component has the strongest influence on the series. The decomposition is achieved using the *Seasonal-Trend decomposition procedure based on LOESS (STL)*. *STL* moves sliding windows over the analyzed time series to compute smoothed representations of the series using the *locally weighted regression (LOESS)* [6, 7].

The width of the sliding windows is chosen specifically to extract certain frequencies. In our system, we set the window width for both the trend and the seasonal components to 6 hours. This width had to be comparable to the expected seasonality of 12 or 24 hours for user traffic. The shorter width permits the extraction of components that may slightly vary from day to day; a larger width would not tolerate such variation. Figure 5 shows the decomposition of the time series from Figure 2.

Trend variation. The trend components for crawlers are often very stable, and thus, close to a square signal. To distinguish stable from noisy “signals”, we apply the differentiation operator ∇ (with a lag 1 and a distance of 1) to the trend. For crawlers where the traffic is rather stable, the differentiated trend is very close to a null series, with the exception of spikes when an amplitude shift occurs. For users, the traffic shows quicker and more frequent variations, which results in a higher variation of the differentiated trend series. The variation of the differentiated trend is measured using the *Index of Dispersion for Counts* [12], $IDC[\nabla T(t)]$. Compared to other statistical indicators such as the variance, the IDC, as a ratio, offers the advantage of being normalized.

Season-to-trend ratio. Time series that correspond to users’ traffic often exhibit repetitive patterns. These pat-

Table 1: Features characterizing traffic time series

Origin	Feature name	Type
Auto-correlation analysis (SAC)	<i>coefficients at lags 1 and 2, decay, sign alternation, number of local spikes, daily correlation</i>	2 x Continuous, 2 x Enumerate(4), 1 x Discrete, 1 x Boolean
Decomposition analysis	<i>differentiated trend IDC, season over trend ratio</i>	1 x Continuous, 1 x Continuous

terns can repeat on a daily basis, weekly basis, or other frequencies that show some significance in terms of the human perception of time. Consequently, the seasonal component is more important for user traffic and likely prevails over the trend component. This is no longer true for crawler traffic. To capture this difference between user and crawler traffic, we compute the ratio between the seasonal and trend components as shown in Equation 4. The amplitude of the seasonality component is computed using the difference between its maximum and minimum values (as the minimum value might be negative). The amplitude of the trend component is measured using a quantile operation to remove outliers resulting from possible errors in the decomposition.

$$r_{s/t} = \frac{Max[S(t)] - Min[S(t)]}{Quantile[T(t), 0.95]} \quad (4)$$

3.3.3 Traffic classification

The auto-correlation and decomposition provide a set of features that describe important characteristics related to the traffic from a given source. These features are summarized in Table 1. The type column shows, for each feature, the domain of values: continuous or discrete numbers, Boolean values, or labels drawn from a set of n possibilities (written as “Enumerate(n)”).

Of course, no single feature alone is sufficient to unequivocally distinguish between crawler and user traffic. Thus, PUBCRAWL trains a combination of three well-known, simple classifiers. The first classifier is a naive Bayes classifier that was trained using the maximum posterior probability [19]. The conditional probabilities for continuous attributes were estimated using weighted local regression. The second classifier is an association rules classifier [5]. The third classifier is a support vector machine (SVM) that was trained using a non-linear kernel function (Gaussian Radial Basis Function - RBF). To construct an optimal hyperplane, we chose a C-SVM classification error function.

All three classifiers require a training phase. For this, we make use of a labeled training set that contains time series for both known crawlers and users. Each classifier is trained separately on the same training data. During the detection phase, each classifier is invoked in parallel. To determine whether an individual source is a crawler or a user, we use majority voting over the outputs of the three classifiers.

Algorithm 1

Require: A time series set $\chi = X_1(t), \dots, X_n(t)$

- 1: $C = \emptyset$
- 2: **for** $X_i(t)$ in χ **do**
- 3: $\tau = k / \text{Stdv}[X_i(t)]$
- 4: $C^* = \text{candidates}(C, \text{Vol}[X_i(t)], \text{Max}[X_i(t)], \text{Stdv}[X_i(t)])$
- 5: $s_m, c_m = \max_{c \in C^*} (\{\text{euclidean_dist}(X_i(t)), c.\text{medoid}\})$
- 6: **if** $s_m > \tau$ **then**
- 7: $c_m.\text{add_time_series}(X_i(t))$
- 8: **else**
- 9: $c_n = \text{new_cluster}(\text{medoid} = X_i(t))$
- 10: $C = C \cup c_n$
- 11: **end if**
- 12: **end for**
- 13: **return** clusters set C

4 Crawling Campaign Attribution

In the previous section, we introduced our approach to detect crawlers based on the analysis of traffic from individual sources. However, numerous crawlers do not operate independently, but work together in crawling campaigns, distributed over multiple locations. By identifying such campaigns, we can provide valuable forensic information to generate the list of blacklisted sources.

Campaign attribution is achieved by identifying crawlers that exhibit a strong similarity between their access patterns. To detect sources that exhibit similar patterns, we use clustering to group similar time series coming from detected crawlers. During the training period, we first determine a minimal intra-cluster similarity required for crawlers to belong to a common campaign. During detection, clustering results are used to identify hosts that exhibit similar activity, and hence, are likely part of a single, distributed crawler.

Time series similarity. A significant amount of literature exists on similarity measures for time series [16]. Most of this body of work aims at providing a similarity measure that is resilient to distortion, so that time series of similar shape can be clustered. Distortion in terms of request volume is already handled by the normalization that we apply when deriving the time series. On the other hand, resilience to time distortion is not desirable. The reason is that we want to detect sources that behave similarly, including the time domain. As a consequence, we leverage the (inverse) *Squared Euclidean Distance*. This metric is fast and known to provide good results [13].

Incremental clustering. PUBCRAWL uses an incremental clustering approach to find similar time series. Time series coming from detected crawlers are submitted one by one to our clustering algorithm described in Algorithm 1. For each new time series, the algorithm computes the similarity between this time series and the medoids of existing clusters (Line 5, Algorithm 1). When the maximal similarity, found for a given medoid, is above a threshold τ (Line 6, Algorithm 1), the time series is added to the associated cluster. Otherwise, the time series becomes the medoid for a new cluster that is

created. τ corresponds to the minimal intra-cluster similarity that the algorithm has to enforce. τ is computed during the learning phase, based on a labeled dataset that contains time series for distributed crawlers. Note that τ is not fixed but depends on the standard deviation to compensate for time series proving to be more “noisy”.

To speed up the process, for each incoming time series, the function *candidates* (Line 4, Algorithm 1) selects a subset of comparable clusters. These candidate clusters are chosen because their medoids have a volume of requests, an amplitude and a standard deviation that are comparable to the new time series. We found that this selection process significantly reduced the number of necessary computations (but also false positives).

Once all time series are processed, each cluster that contains a sufficient number of elements is considered to represent a crawling campaign. Sources that are part of this cluster are flagged as parts of a distributed crawler.

5 Crawler Proactive Containment

Existing techniques to detect crawlers, including our approach, often require a non-negligible amount of traffic before reaching a decision. To address attacks in which an adversary leverages a large number of bots for crawling, we require an additional containment mechanism. In PUBCRAWL, the detection modules are mainly used to produce two lists: A whitelist of IPs corresponding to authorized users (proxies) and legitimate crawlers, and a blacklist of IPs corresponding to suspicious crawlers. These lists are used to enforce an access policy for the real-time stream of requests.

Whenever the protected web site receives a request from a source in the whitelist, the request is granted. Sources on the blacklist are blocked. Other sources are considered unknown, and are treated as follows.

For each source, we check the number of requests that were generated within a given time interval (currently, one day). If this volume remains below a minimal threshold k_1 , the source is considered a user, and its access to the site is granted. If k_1 is chosen sufficiently small, the amount of leaked information remains small, even if an attacker has many machines at their disposal.

If this same volume is above a second threshold k_2 , we can use our models to make a meaningful decision and either whitelist or blacklist this source.

When the number of requests from a source is between k_1 and k_2 , unknown sources are exposed to active responses such as CAPTCHAs and crawler traps. By modifying k_1 and k_2 , a number of trade-offs can be made. When k_1 increases, fewer users are exposed to active responses, but it is easier for large-scale, distributed attackers to steal data. When k_2 increases, our system will be more precise in making decisions between crawlers and users but we expose more users to

active responses. In Section 6, we show that, for reasonable settings of the thresholds k_1 and k_2 , only a very small fraction of requests and IP addresses are subjected to active responses, while the amount of pages that even large botnets can scrape remains small.

In practice, PUBCRAWL will only be used for “anonymous” requests. These are requests from users who do not have an account on the site or have not logged in. When a user authenticates (logs in), subsequent requests will contain a cookie that grants direct access to the site (and authenticated requests are rate-limited on an individual basis). This is important to consider when discussing why IP addresses form the basis for our white- and blacklists. In fact, we are aware that making decisions based on IP addresses can be challenging; IP addresses are only a weak and imperfect mechanism to identify the source of requests. However, for anonymous requests, the IP address is the only reliable information that is available to the server (since the client completely controls the request content).

One problem with using IP addresses is that a malicious crawler (or bot) on an infected home computer might regularly acquire a new IP address through *dhcp*. Thus, the blacklist can become stale quickly. We address this problem by allowing each individual IP address only a small number k_1 of unrestricted accesses (before active containment is enabled). While each fresh IP address does allow a bot a new set of requests, IP addresses are not changing rapidly enough so that attackers can draw a significant advantage. Another problem is that the IP address of a whitelisted, legitimate proxy could change, subjecting the users behind it to unwanted, active containment. Our experimental results (in Section 6) show that this is typically not the case, and legitimate, high-traffic sources are relatively stable. Finally, it is possible that an attacker compromises a machine behind a whitelisted proxy and abuses it as a crawler. To protect against this case, our system enforces a maximum traffic volume after which the whitelist status is revoked and the IP address is treated as unknown.

To keep the access control lists up-to-date, PUBCRAWL continuously re-evaluates unknown sources and entries on the whitelist. Entries in the blacklist are expired after some days. Moreover, users can always authenticate to the site to bypass PUBCRAWL’s checks.

6 Evaluation

We implemented PUBCRAWL in Python, with an interface to R [1] for the time series analysis. The system was developed and evaluated in cooperation with a large, well-known social network. More precisely, we used ten days of real-world request data taken from the web server logs of the social network, and we received feedback from expert practitioners in assessing the quality of

our results. The evaluation yielded a favorable outcome, and our detection system is now integrated into the production environment of a large social networking site.

6.1 Dataset presentation

The ten days of web server logs contain all requests to public profile pages of the social networking site we used as a case study. Public profiles are accessible by anyone on the Internet without requiring the client to be logged in. The social network provider has experienced that almost all crawling activity to date comes from clients that are not logged into their system. The reason is that authenticated users are easy to track and to throttle. Handling large volumes of non-authenticated traffic from a single source is most difficult; this traffic might be the result of anonymous users surfing behind a proxy, or it might be the result of crawling. Making this distinction is not straightforward.

The log files contain eight fields for each request: *time*, *originating IP address*, *Class-C subnet*, *user-agent*, *target URL*, *server response*, *referrer*, *cookie*. The IP address and the Class-C subnet fields were encrypted to avoid privacy issues. Thus, we can only determine whether two requests originate from the same client, or from two clients that are part of the same /24 network. The remaining information is unmodified. This allows us to check for suspicious user agents, and to determine the profile names that are accessed. The server response can be used to determine whether the visited profile exists or not. In addition, the referrer indicates the previous website visited by the client. The cookie contains, in our case, a session identifier that is set by the social networking site to track individual clients.

Data prefiltering. Given that the social network is very popular, the log files are large – they contain tens of millions of requests per day that originate from millions of different IP addresses. As a result, we introduce a prefiltering process to reduce the data to a volume that is manageable by our time series analyses. To this end, we leverage the fact that, by looking at the volume of requests from a single source, certain clients can be immediately discarded: we can safely classify all sources that generate more than 500,000 requests per day as crawlers.

Sources that generate less than 1,000 requests per day are also put aside because our time-series-based techniques require a minimum number of data points to produce statistically meaningful results. These sources are handled by the active containment policy.

In our experiments, the prefiltering process reduced the number of requests that need to be analyzed by a factor of two. More importantly, however, the number of clients (IP addresses) that need to be considered is reduced by about four orders of magnitude.

Ground truth. Obtaining ground truth for any real-world data is difficult. We followed a semi-automated approach to establish the ground truth for our datasets.

For the initial labels, we used heuristic detection (Section 3.1), which represents the state-of-the-art in crawler detection. We then contacted the social network site, which had access to the actual (non-encrypted) IP addresses. Based on their feedback, we made readjustments to the initial ground truth labels. More precisely, we first marked sources as legitimate crawlers when they operated from IP ranges associated with popular search engines. In addition, IP addresses that belong to well-known companies were labeled as users. For borderline cases, if an IP address was originating traffic from users who successfully logged on to the site, we tagged this IP as a user. Overall, we observed that current heuristics often incorrectly classify high volume sources as crawlers.

In a next step, we performed an extensive manual analysis of the sources (by looking at the time series, checking user agent string values, ...). We made a second set of adjustments to the ground truth. In particular, we found a number of crawlers that were missed by heuristic detection. These crawlers were actively attempting to mimic the behavior of a browser: user agent strings from known web browsers, cookie and referrer management, and slow runs at night. Some examples of mimicry are discussed in Appendix B. These cases are very interesting because they underline the need for more robust crawler detection approaches such as PUBCRAWL.

Finally, we manually checked for similar time series, and correlated this similarity with user agent strings and class-C subnets. This information was used to build a reference clustering to evaluate the campaign attribution.

6.2 Training detection and attribution

For training, we used a first dataset S_0 , which contained five days worth of traffic recorded between the 1st and the 5th of August 2011. After prefiltering, this dataset consists of ~ 73 million requests generated by 813 IP addresses. Given this number of IP addresses, manual investigation for the ground truth was possible.

Heuristic detection. We used the training set to individually determine suitable thresholds for the detection heuristics. We verified the results of the configured heuristics over the training set S_0 with the ground truth. The results are given in Table 2. In this table, a true positive (TP) means a correctly identified crawler. A true negative (TN) is a correctly identified user. The results are split between heuristics over features from existing work and new features introduced in this paper. We can see that the new features greatly improve the detection rate when combined with existing ones. Still, the final detection rate of 75.54% illustrated the need for more robust features.

Table 2: Training: Accuracy for heuristic detection.

Rates	Former features	New features	Combined features
TP/FN	41.32%/58.68%	82.31%/17.69%	75.54%/24.46%
TN/FP	100.00%/0.00%	84.00%/16.00%	96.00%/04.00%

Table 3: Training: Accuracy for traffic shape detection.

Crawlers (TP/FN)	Bayes	Rules	SVM	Vote
Cross validation	98.39%	96.36%	98.55%	98.99%/01.01%
Two third split	97.45%	96.19%	95.11%	96.90%/03.10%
Users (TN/FP)	Bayes	Rules	SVM	Vote
Cross validation	79.09%	78.91%	81.91%	82.91%/17.09%
Two third split	78.84%	78.22%	80.44%	82.28%/17.72%

Table 4: Training: Accuracy for campaign attribution.

Precision	Recall	Accuracy
99.03%	85.54%	94.35%

Traffic shape detection. To train the classifiers for detection, we used S_0 that contained 709 crawler sources and 104 user sources. To determine the quality of the training over S_0 , we used both five-fold cross validation and a validation by splitting the data into two thirds for training and one third for testing. The results are shown in Table 3. The table shows that our system obtains a crawler detection rate above 96.9%. It also shows the benefit of voting, as the final output of classification is more accurate than each individual classifier.

Interestingly, the dataset was not evenly balanced between crawlers and users. The majority of sources that produce more than thousand requests per day are crawlers. However, the dataset also contains a non-trivial amount of user sources. Thus, it is not possible to simply block all IP addresses that send more than one thousand requests. In fact, since the user sources are typically proxies for a large user population, blocking these nodes would be particularly problematic. We thus verified the accuracy specifically for user sources in Table 3. Given the bias towards crawlers, the accuracy for users is slightly lower but remains at around 83%.

It must be noticed that traffic shape detection results show interesting improvements compared to heuristic detection that is close to the approach deployed in existing work. This approach produces more accurate results while using features that are more robust to evasion.

Campaign attribution. For campaign attribution, the clustering algorithm presented in Section 4 needs to be configured with a τ that defines the minimal, desired similarity within clusters. To determine this threshold, we ran a bisection clustering algorithm on the training dataset S_0 . The algorithm first assumes that all elements (time series) are part of a single, large cluster. Then, it iteratively splits clusters until each time series is part of a single cluster. We analyzed the entire cluster hierarchy to find the reference clusters as well as the necessary cut points to obtain them. The cut points of reference clusters indicated us the minimal similarity that we related to the deviation to determine that

$k = 350$ was the linear coefficient giving optimal values for τ . For the candidate selection, the following thresholds were chosen: *volume* = 25%, *amplitude* = 35%, *deviation* = 30%, so that these thresholds avoid additional false positives while creating no false negatives compared to the results without candidate selection.

To evaluate the quality of the campaign attribution, we ran our clustering algorithm on the 709 crawler sources from S_0 . We use the *precision* to measure how well the clustering algorithm distinguished between time series that are different, and the *recall* to measure how well our technique recognizes similar time series. To evaluate the successful attribution rate, we use the *accuracy* to measure how well the clustering results can be used to detect distributed crawlers and thus campaigns.

The clustering results are shown in Table 4. One can see that the algorithm offers a good precision. The recall is a bit lower: Closer examination revealed that a few large reference clusters were split into multiple clusters. For example, *Bingbot* had its 209 corresponding time series split into 5 subclusters. Fortunately, recall is less important in our attribution scenario. The reason is that split clusters do not prevent us to detect a campaign in most cases; instead, a campaign is simply split into several smaller ones.

6.3 Evaluating detection capabilities

For testing, the social networking site provided an additional dataset S_1 , which contained five extra days worth of traffic. After prefiltering, this dataset consisted of ~62 million requests generated by 763 IP addresses. Unlike the training, the testing was performed on site at the social networking site. Hence, the traffic logs could be analyzed with non-encrypted IPs.

Heuristic detection. We compared the results for the heuristic detection over the testing set S_1 with the ground truth we had (semi-automatically) established previously. As shown in Table 5, the detection rate slightly decreases to 71.60%, but remains comparable to the rate over the training set.

Traffic shape detection. To test the classifiers trained over S_0 , we deployed the traffic shape detection approach over the test set S_1 . The results for this experiment are presented in Table 6. According to the table, the global accuracy of 94.89% remains very close to the 95% of accuracy observed for the training set.

Since the goal of the detection module is to build whitelists and blacklists of sources, we computed individual results for the following four subsets: users (5%) and legitimate crawlers (65%) to be whitelisted, and unauthorized crawlers (7%) and masquerading crawlers (23%) to be blacklisted. Unauthorized crawlers are agents that can be recognized by their user agent string

Table 5: Testing: Accuracy for heuristic detection.

Rates	Former features	New features	Combined features
TP/FN	31.19%/68.81%	86.24%/14.76%	71.60%/28.40%
TN/FP	100.00%/0.00%	87.18%/12.82%	94.87%/05.13%

Table 6: Testing: Accuracy for traffic shape detection.

	Bayes	Rules	SVM	Vote
Global	93.05%	87.55%	94.36%	94.89%
Legitimate crawlers	92.54%	87.10%	97.18%	93.95%
Unauthorized crawlers	88.89%	96.27%	100.00%	100.00%
Masquerading crawlers	98.27%	86.71%	98.84%	98.84%
Crawlers (TP/FN)	93.66%	87.68%	97.79%	95.58%/04.42%
Users (TN/FP)	82.50%	85.00%	32.50%	82.50%/17.50%

Table 7: Testing: Accuracy for campaign attribution.

Precision	Recall	Accuracy
92.84%	80.63%	91.89%

but are not supposed to crawl the site. Masquerading crawlers are malicious crawlers trying to masquerade as real browsers to remain stealthy and to avoid detection.

We achieve perfect detection for unauthorized crawlers. In particular, the system was able to detect crawlers such as *ISA connectivity checker*, *Yandexbot*, *YooiBot*, or *Exabot*. Results are also very good for masquerading browsers, with a detection rate close to 99%. The detection rate slightly drops to 94% for legitimate crawlers such as *Baiduspider*, *Bingbot*, *Googlebot* or *Yahoo!slurp*. But 4% of these false negatives are related to *Google FeedFetcher*. In principle, the requests from *FeedFetcher* are triggered by user requests. As a result, its time series are individually recognized as user traffic.

By combining heuristic detection and traffic shape detection, the detection rates were not improved further. The reason is that the crawlers detected by heuristics were already included in the set of crawlers detected by traffic shape. This observations confirms our belief that traffic shape detection is stronger than heuristic detection based on HTML and URL features.

To gain a better understanding of our results, we asked for feedback from the social networking site. The network administrators confirmed that a large number of crawlers were previously unknown to them (and they subsequently white- or blacklisted the IPs). Since de-anonymized IP addresses were available to us, we could check the sources of these crawlers. Interestingly, several sources were proxies of universities, where crawler traffic was mixed with user activity. Because of the mix of user and crawler activity, the current detection techniques did not raise alerts. Note that such mix of activity must be taken into consideration for blacklisting (e.g., the university administrators can be warned that unauthorized crawling is coming from their network and asked to take appropriate measure). In such cases, it is a policy decision whether to blacklist the IP or not. Also, recall that requests from users who are logged-in is not affected.

Performance. To process the entire dataset S_1 , several instances of the modules for heuristic detection, traffic shape detection and campaign attribution were run over five parallel processes that required roughly 45 minutes to run. This time included loading the data into the database, generating the time series, and performing the different analyses. The experiments were run on a single server (with 4 cores and 24 GB of RAM).

6.4 Evaluation of campaign attribution

To evaluate the quality of the campaign attribution technique, we ran our clustering algorithm over the crawler sources from the de-anonymized testing set S_1 . The results of the experiment are shown in Table 7. One can see that the precision and recall have slightly dropped compared to the training set. The intra-cluster similarity threshold τ might not be optimal anymore. Nonetheless, the attribution accuracy remains at 91.89%, which is close to the 94% obtained during training.

Overall, we obtain 238 clusters from the 763 distinct source IPs. Looking at these clusters, we started to observe interesting campaigns when a cluster contained 3-4 or more elements (hosts). Table 8 provides a description of these campaigns. The first campaigns correspond to legitimate crawlers. Interestingly, the campaign associated to *Feedfetcher*, whose crawlers evaded traffic shape detection, is successfully identified. Even if the *Feedfetcher* traffic is similar to user traffic, *Google* distributes the requests (i.e., load-balances) over multiple IPs, explaining that their time series are synchronized. Table 8 also presents five campaigns showing suspicious user-agent strings, and five campaigns masquerading as legitimate browsers or search engine spiders. Another interesting case is the campaign where crawlers send requests as *Gecko/20100101 Firefox*. This campaign shows a significant number of clusters because it uses *rotating IP addresses* over short time periods. However, it is still detected because the active IP addresses were operating loosely synchronized. The social network operators showed a particular interest in this case, and they are now relying on our system to detect such threats that are difficult to detect otherwise.

6.5 Evaluation of proactive containment

The evaluation was performed over a dataset S_2 of non-filtered traffic. The dataset contained ~ 110 million requests from ~ 11 million IP sources.

Figure 6 shows the cumulative distribution function (CDF) for the fraction of IPs (y-axis) that send (at most) a certain number of requests (x-axis). One can see that most IPs (more than 98.7%) send only a single request per day. This is important because it means that most sources (IPs) will never be affected by active containment. Figure 7 shows the situation for requests instead

Table 8: Identified Crawling Campaigns.

Agent	#Clust.	#ClassC	#IP	Req/day
Legitimate crawlers				
<i>Bingbot</i>	5	11	211	6 million
<i>Googlebot</i>	5	2	42	4 million
+ <i>Feedfetcher</i>	4	4	65	-
<i>Yahooslurp</i>	4	9	71	500 thousand
<i>Baiduspider</i>	1	1	23	50 thousand
<i>Voilabot</i>	3	3	20	19 thousand
<i>Facebookexternalhit/1.1</i>	1	1	8	14 thousand
Crawlers with suspicious agent strings				
" "	2	16	22	330 thousand
<i>Python-urllib/1.17</i>	2	51	54	140 thousand
<i>Mozilla(compatible;ICS)</i>	1	10	10	70 thousand
<i>EventMachine HTTP Client</i>	1	3	3	3 thousand
<i>Gogospider</i>	1	2	3	2 thousand
Masquerading crawlers				
<i>Gecko/200805906 Firefox</i>	1	10	73	350 thousand
<i>Gecko/20100101 Firefox</i>	9	12	25	60 thousand
<i>MSIE6 NT5.2 TencentTraveler</i>	1	1	30	7 thousand
<i>Mozilla(compatible; Mac OS X)</i>	1	1	4	8 thousand
<i>googlebot(crawl@google.com)</i>	1	1	4	1 thousand

of IPs. That is, the figure shows the CDF for the fraction of total requests (y-axis) over the maximum number of requests per source (x-axis). One can see that roughly 45% of all request are sent by sources that send at most 100 requests. The graph highlights that a little over 40% of all requests are done by sources that make only a single request. One can also see that a significant amount of the total requests are incurred by a few heavy hitter IPs that make tens of thousands to millions of requests.

Containment impact. We use these two graphs to discuss the impact of the two containment thresholds: k_1 , below which sources have unrestricted access to the site, and k_2 , above which sources can be examined by our analysis (and hence, properly whitelisted or blacklisted).

We can see from Figure 6 that if we choose k_1 low enough, we can guarantee that only a tiny number of sources will be impacted. For example, by setting $k_1 = 100$, we see that 99.98% of the sources will not be impacted at all. If an attacker wants to take advantage of this unrestricted access, he would require 5,000 crawlers running in parallel to reach the crawling rate of a *single* crawler agent from *Googlebot*. Looking back at Table 8 for real-world examples, the lowest rate of requests we observed for a distributed crawler was for the *Gecko/20100101 Firefox* campaign using rotating IPs. Even in this case, the amount of requests per agent was above a few hundreds per day. Thus, we consider values for k_1 between 10 and 100.

To choose k_2 , we have a trade-off between the amount of traffic that will be impacted by active responses and the quality of our detection. We see the fraction of requests that are impacted by plotting k_1 and k_2 over Figure 7 and reading the difference over the y-axis. Table 9 lists the proportion of impacted traffic for various concrete settings of k_1 and k_2 . If we keep k_2 at 1,000 and choose 20 for k_1 , we expect active responses to impact less than 0.1% of all IP sources **and** only about 3.24%

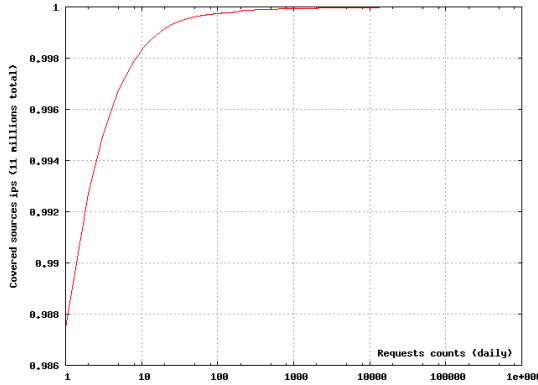


Figure 6: CDF of the source IPs over traffic volumes

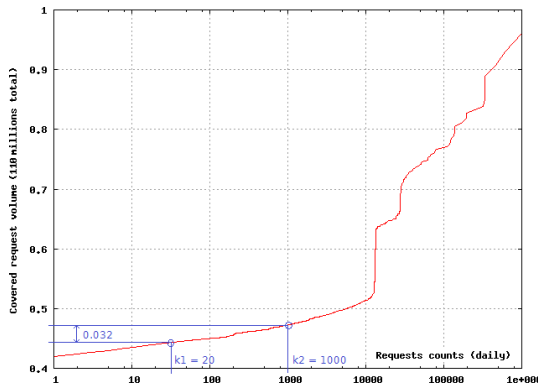


Figure 7: CDF of the requests over traffic volumes

Table 9: Requests (%) impacted by containment

k_1/k_2	100	500	1000	2000	5000	10000
10	1.49%	2.95%	3.75%	4.82%	6.27%	7.88%
15	1.20%	2.66%	3.46%	4.53%	5.98%	7.58%
20	0.99%	2.44%	3.24%	4.32%	5.77%	7.37%
25	0.81%	2.27%	3.07%	4.14%	5.59%	7.19%
50	0.37%	1.83%	2.63%	3.70%	5.15%	6.75%
100	0.00%	1.46%	2.26%	3.33%	4.78%	6.38%

of the overall requests. Of course, we expect that these 3.24% of requests do contain a non-trivial amount of traffic from stealthy crawlers. When k_1 is increased, the impact on legitimate users decreases. The downside is that large botnets can scrape larger parts of the site.

Sources stability. The whitelist approach for legitimate, high-volume sources (over k_2) works well only when IP sources remain stable for user proxies and legitimate crawlers. To verify this assumption, we studied the IP evolution between the training set S_0 and the testing set S_1 . Considering crawlers, 66.9% of IPs were both present in S_0 and S_1 . Looking at the stable IPs, they correspond either to legitimate crawlers (e.g., *Googlebot*, *Bingbot*) or large crawling campaigns (e.g., *Python*, *Firefox* from Table 8). Unstable crawler IPs correspond to unauthorized and masquerading crawlers.

Most importantly, looking at high-volume users (proxies), 81.8% of IPs were both present in S_0 and S_1 (about one month apart). Thus, whitelisting these sources would work well and, hence, we would not see

much negative effect for (non-authenticated) users when using active containment. Moreover, the 18.2% of non-stable IPs are mainly due to sources whose volume was close to the prefiltering threshold. These sources might have been stripped from S_0 or S_1 by prefiltering.

Overall, our results demonstrate that PUBCRAWL can thwart large-scale malicious crawlers and botnets while interfering with a small number of legitimate requests.

7 Limitations

Our system has proven to be useful in the real-world, and it is currently deployed by the social networking site as a part of their crawler mitigation efforts. However, as with many other areas in security, there is no silver bullet, and sophisticated attackers might try to bypass our system. Nevertheless, PUBCRAWL significantly increases the difficulty bar for the attackers.

Detection limitations. An attacker might try to thwart the heuristics-based and traffic-shape-based detection modules. The traffic shape detection has two main requirements: 1) a large-enough volume of requests is required for the time series to be statistically significant, 2) at least two days of traffic are required for the auto-correlation and decomposition analyses.

While traffic shape detection is well-suited for detecting crawlers of sufficient volume, because of requirement 1), it is not particularly well-suited to detect “slow” distributed crawlers that send a very small number of requests from hundreds of thousands of hosts. For this, campaign attribution is more appropriate. Because of requirement 2), it is not well suited either to detect “aggressive” (noisy) crawlers in real-time. For this, heuristic detection is more appropriate.

To address the problem of “slow” and “aggressive” crawlers, PUBCRAWL combines the detection modules with a containment strategy. Aggressive crawlers are initially slowed down by active responses, until they are detected and blacklisted. Slow crawlers can at most make k_1 requests before the active response component is activated. Moreover, since slow crawlers require a larger number of machines, the effect of the active response component is magnified (applied to each crawler).

Another way to avoid detection is traffic reshaping. That is, an attacker could try to engineer the crawler traffic so that it closely mimics the behavior of actual users. The attacker would first have 1) to craft valid HTTP traffic (headers) and 2) to design a stealthy visiting policy both in terms of *topology* and *timing*. In terms of topology, the attacker would have to craft a non-suspicious sequence of URLs to visit (based on order and revisit behavior). In terms of timing, he would have to craft the volume and distribution of requests over time so that the traffic shape remains similar to user traffic. Overall,

mimicking the behavior of users would require a non-trivial effort on behalf of the attacker to learn the properties of user traffic, especially given that only the social network has a global overview of the incoming traffic.

Attribution limitations. If an attacker wants to bypass our campaign attribution method, he has to ensure that *all* nodes of his distributed crawler behave differently. Sets of rotating IPs are already successfully detected. To break the synchronization between the nodes, simple time shifts would not be sufficient: Existing similarity measures for time series (e.g., dynamic time warping) can be used to recover from shifts. An attacker would have to completely break the synchronization between its different crawlers while shaping for each one a different traffic behavior (which needs to be similar to user traffic to avoid individual detection).

Containment limitations. If attackers do not succeed in whitelisting their crawlers, they can willingly maintain their traffic volume under the containment threshold k_1 . To crawl a good portion of a social network with millions of pages requires a significant crawling infrastructure, and building or renting botnets over long periods of time might be prohibitively expensive for most adversaries. Attackers can also increase their traffic volume until the blocking threshold k_2 . In this case, they would have to find a solution to automatically bypass active responses (resolve CAPTCHAs or identify crawler traps).

8 Related Work

We are not the first one to study the problem of detecting web crawlers. However, we are the first to propose a solution to distributed crawlers, and we are the first to have used an extensive, large-scale real-world dataset to evaluate and validate our approach.

Similar to [18, 21, 22], PUBCRAWL relies on machine learning techniques to extract characteristic properties of traffic that can help to distinguish crawlers from users. However, the features we use for the learning process are different. Compared to [11, 17, 21, 22], the similar features we extract from the HTTP headers and URLs are fed to heuristic detection. Our experiments demonstrate that these features are not reliable.

For traffic shape detection and its learning process, we used timing features instead. Compared to [18, 21, 22], the results of the auto-correlation and decomposition analyses prove to be more robust. That is, the extracted properties are harder to evade by attackers than the simple time and volume statistics used by previous approaches (e.g., the average or the variance of inter-arrival times between requests).

In our detection approach, most of the features extracted from the time series are designed to express the regularity of web traffic. In [8], the authors already

leveraged the notion of regularity of crawler traffic for detection. To extract the relevant information, the authors rely on Fast Fourier Transformations. However, both crawler and user traffics show regularities, but they do so at different levels. We thus use decomposition techniques to distinguish between different types of regularities: Crawler regularity can be observed within the trend component, whereas user regularity can be observed within the seasonal component.

Existing crawler detection approaches mainly remain deployed offline – just like our detection approach based on traffic shape. However, a significant novelty of our approach is that we integrate our detection process into a proactive containment strategy to protect from crawlers in real-time. This is similar to [18] where the authors address real-time containment. The containment approach they propose relies on the detection results from an incremental classification system where crawler models evolve over time. Instead, we chose a more realistic, practical white- and blacklisting approach where sources are blocked on a per IP basis.

An interesting contribution of our work is the formalization of the traffic by time series, which allows us to address the problem of crawling campaign attribution and distributed crawlers detection using clustering. We have also evaluated our tool on a much larger scale than previous work, which has used requests that are in the order of thousands. Finally, in our experiments, we observed and identified real-world evasion techniques that target some of the traffic features used in previous work. Hence, we provide evidence that attackers today are investing significant effort to evade some of the straightforward and well-known crawler detection techniques.

9 Conclusion

To defend against malicious crawling activities, many websites deploy heuristics-based techniques. As a reaction, crawlers have increased in sophistication. They now frequently mimic the behavior of browsers, as well as distribute their activity over multiple hosts.

This paper introduced PUBCRAWL, a novel approach for the detection and containment of crawlers. The key insight of our system is that the traffic shape of crawlers and users differ significantly so that they can be automatically identified using time series analysis. We also propose the first technique that is able to identify crawling campaigns by detecting the synchronized activity of distributed crawlers using time series clustering. Finally, we introduce an active containment mechanism that strategically uses active responses to maximize protection and minimize user annoyance.

Our experiments with a large, popular social networking site demonstrate that PUBCRAWL can distinguish users with accuracy and filter out crawlers. Our detec-

tion approach is currently deployed in production by the social networking site we collaborated with.

Acknowledgment

We would like to thank the people working at the social network with whom we collaborated, as well as Secure Business Austria for their support. This work was supported by the Office of Naval Research (ONR) under Grant N000140911042 and the National Science Foundation (NSF) under grants CNS-0845559, CNS-0905537 and CNS-1116777.

References

[1] The R Project for Statistical Computing. <http://www.r-project.org/>.

[2] 67th District Court, Tarrant County, Texas. Cause NO. 067-194022-02: American Airlines, Inc. vs. FareChase, Inc. <http://www.fornova.net/documents/AAFareChase.pdf>, 2003.

[3] B. L. Bowerman, R. T. O’Connell, and A. B. Koehler. *Forecasting, Time Series, and Regression – An applied approach – Fourth edition*. Thomson Brook/Cole, 2005.

[4] E. Burzstein, R. Bauxis, H. Paskov, D. Perito, C. Fabry, and J. Mitchell. The Failure of Noise-Based Non-Continuous Audio Captchas. In *IEEE Security and Privacy*, 2011.

[5] P. Clark and T. Niblett. The CN2 Induction Algorithm. *Machine Learning*, 3(4):261–283, 1989.

[6] R. B. Cleveland, W. S. Cleveland, J. E. McRae, and I. Terpenning. STL: a Seasonal-Trend Decomposition Procedure based on Loess. *Journal of Official Statistics*, 6(1):3–73, 1990.

[7] W. S. Cleveland and S. J. Devlin. Locally Weighted Regression: an Approach to Regression Analysis by Local Fitting. *J. Am. Stat. Assoc.*, 83:596–610, 1988.

[8] M. D. Dikaiakos, A. Stassopoulou, and L. Papageorgiou. An Investigation of Web Crawler Behavior: Characterization and Metrics. *Computer Networks*, 28:880–897, 2005.

[9] D. Doran and S. S. Gokhale. Discovering New Trends in Web Robot Traffic through Functional Classification. In *Proc. of the IEEE International Symposium on Networking Computing and Applications (NCA)*, pages 275–278, 2008.

[10] D. Doran and S. S. Gokhale. Web Robot Detection Techniques: Overview and Limitations. *Data Mining and Knowledge Discovery*, 22(1-2):183–210, 2011.

[11] W. Guo, S. Ju, and Y. Gu. Web Robot Detection Techniques based on Statistics of their Requested URL Resources. In *Proc. of the International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pages 302–306, 2005.

[12] R. Gusella. Characterizing the Variability of Arrival Processes with Indexes of Dispersion. *IEEE J. Sel. Areas Commun.*, 9(2):203–211, 1991.

[13] E. Keogh and S. Kasetty. On the Need for Time Series Data Mining Benchmarks: a Survey and Empirical Demonstration. In *Proc. of the ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 102–111, 2002.

[14] M. Koster. A Method for Web Robots Control. Technical report, RFC draft, 1996.

[15] M. Lamberton, E. Levy-Abegnoli, and P. Thubert. System and Method for Enabling a Web Site Robot Trap. United States Patent No. US 6,925,465 B2, 2005.

[16] T. W. Liao. Clustering of Time Series Data – a Survey. *Pattern Recognition*, 38(11):1857–1874, 2005.

[17] X. Lin, L. Quan, and H. Wu. An Automatic Scheme to Categorize User Sessions in Modern HTTP Traffic. In *Proc. of GLOBE-COM*, pages 1485–1490, 2008.

[18] A. G. Lourenço and O. O. Belo. Applying Clickstream Data Mining to Real-Time Web Crawler Detection and Containment using ClickTips Platform. In *Advances in Data Analysis, Proc. of the 30th Annual Conference of the Gesellschaft für Klassifikation e.V.*, pages 351–358. Springer, 2007.

[19] A. McCallum and K. Nigam. A Comparison of Event Models for Naive Bayes Text Classification. In *AAAI FICML Workshop on Learning for Text Categorization*, 1998.

[20] Pinsent Masons. Ryanair wins German court victory in screen-scraping injunction. http://www.theregister.co.uk/2008/07/11/ryanair_screen_scraping_victory/, 2008.

[21] A. Stassopoulou and M. D. Dikaiakos. Crawler detection: A bayesian approach. In *Proc. of the International Conference on Internet Surveillance and Protection (ICISP)*, 2006.

[22] P.-N. Tan and V. Kumar. Discovery of Web Robot Sessions based on their Navigational Patterns. *Data Mining and Knowledge Discovery*, 6(1):9–35, 2002.

[23] Tech Crunch. Hacker arrested for blackmailing StudiVZ and other social networks. <http://eu.techcrunch.com/2009/10/21/hacker-arrested-for-blackmailing-studivz-and-other-social-networks/>, 2009.

[24] L. von Ahn, M. Blum, N. Hopper, and J. Langford. The CAPTCHA Project. Technical report, Carnegie Mellon University, 2000.

[25] P. Warden. How I got sued by Facebook. <http://petewarden.typepad.com/searchbrowser/2010/04/how-i-got-sued-by-facebook.html>, 2010.

A Interpreting auto-correlation functions

This section describes how we compute the values of the three features describing the shape of the *Sample Auto-Correlation Function (SAC)*. We first compute *runs* over four SAC characteristics: trend (increase/decrease), sign (positive/negative), significance (spikes), and null (lags with null correlation). The *run* computations are shown in Equations 5-8 for the auto-correlation coefficient r_k that ranges over all k lags.

$$\text{Trend: } tr_k = \begin{cases} 1 & \text{if } k = 1 \text{ or } |r_k| \geq |r_{k-1}| \\ 0 & \text{else} \end{cases} \quad (5)$$

$$\text{Sign: } sr_k = \begin{cases} 1 & \text{if } r_k \geq 0 \\ 0 & \text{else} \end{cases} \quad (6)$$

$$\text{Significance: } pr_k = \begin{cases} 1 & \text{if } |r_k| \geq 2 \times s_k \\ 0 & \text{else} \end{cases} \quad (7)$$

$$\text{Null: } nr_k = \begin{cases} 1 & \text{if } |r_k| \leq 0.1 \times \max\{|r_k|\} \\ 0 & \text{else} \end{cases} \quad (8)$$

Based on the previously-computed runs (as well as amplitude values), we determine the three properties of the SAC: Table 10 lists the heuristics to determine the decay, Table 11 lists the heuristics to determine the sign alternation, and Table 12 lists the heuristics to identify local spikes.

B Crawlers mimicking user behavior

Table 13 shows examples of crawlers that we found in our dataset. The first example in the table represents an easy crawler to detect: The user-agent points to the *Python* programming framework, which is popular among crawlers, no

Table 10: Heuristics to determine the speed of decay of the SAC

Characteristic	Metric	Expected Value	Interpretation	Feature Value
Amplitude	difference between first and last lags	low	no overall decrease in the amplitude of the coefficients	<i>No decay (white noise)</i>
Trend runs	mean of the runs values	average	balanced proportion between increases and decreases	
	number of runs	high	high number of inflections points in the function	
Null runs	mean of the runs values	low	observed coefficients are not residuals	
Amplitude	difference between first and middle lags	above average	quick decrease in the amplitude of the coefficients	<i>Cut-off</i>
Spike runs	length of the first run	low	coefficients become insignificant after a short number of lags	
	length of the longest null valued run	high	the function contains long intervals of insignificant coefficients	
Null runs	mean of the runs values	above average	the function coefficients tend quickly towards 0	
	lag of the first positive run	low	the function converges towards 0 in the small lags	
Amplitude	difference between first and middle lags	below average	slow decrease in the amplitude of the coefficients	<i>Linear decay</i>
Trend runs	number of runs	low	low number of inflections points in the function	
Spike runs	length of the first run	above low	coefficients remain significant after a longer number of lags	
Spike runs	length of the first run	above low	coefficients remain significant after a longer number of lags	
Null runs	mean of the runs values	low	the function coefficients tend slowly towards 0	
-	-		all decaying functions that are neither cut off nor linear	<i>Exponential decay</i>

Table 11: Heuristics to determine the sign alternation property of the SAC

Characteristic	Metric	Expected Value	Interpretation	Feature Value
Sign runs	number of runs	equal 1	no sign change	<i>No alternation</i>
Sign runs	number of runs	equal 2	single sign change	<i>Single alternation</i>
Sign runs	number of runs	above average	important number of sign changes	<i>Oscillations</i>
	variance of the runs length	below average	sign changes occur at regular periods	
-	-		sign changes are unpredictable	<i>Erratic alternation</i>

Table 12: Heuristics to determine the local spikes of the SAC

Characteristic	Metric	Expected Values	Interpretation	Feature Value
Spike runs	lags of positive runs	above 1	first spike must be ignored	<i>Local spike (lag, length)</i>
	length of positive runs	low	short spikes centered around a lag	

referrer is set, the cookies transmitted by the server are ignored, the dispersion index is low, indicating regular traffic, and the error rate is too high to be generated by a user following links. The next examples introduce different evasion techniques by mimicry. Examples 2 to 7 use user-agent string masquerading. Examples 4 and 5 set their referrer to the result of a directory query, whereas Example 7 uses mainly profiles URL as referrers, just like during human browsing. Examples 6 to 8 handle cookies with different policies: Examples 6 and 7 reuse the same cookie for a fixed number of requests, whereas Example 8 reuses the same cookie for a fixed period of time. More advanced techniques of traffic shaping can also be found, where the timing and the targeting of requests is mod-

ified to look similar to user traffic. Example 6 interrupts its activity at night to avoid raising suspicion. Examples 6 and 7 show a high fraction of revisited pages, which is more typical for human behavior. These examples indicate that crawlers already attempt to bypass existing detection heuristics.

Table 13: Evasion techniques observed for different crawlers.

Source	Referrer	Cookie	IDC[X(t)]	Overlap	Errors	Traffic
Grey cells correspond to observed properties of the crawler traffic that are close to browser traffic, they might correspond to evasion attempts.						
IP: B2ED1877DE4DC83B3DF0ED8AFF68E6B0490B5107 User-agent string: Python-urllib/1.17	null referrer	no cookie reuse	517.79	00.0%	12.7%	
IP: 8DB9FE9C5D0796FF7E9BFB05201EE55A9052C586 User-agent string: Mozilla/4.0 (Windows; U; Windows NT 6.1; en-US; rv...)	null referrer	no cookie reuse	10.49	01.3%	01.4%	
IP: A30B14FA5261FA04C612F5E938FCB349C28CBF58 User-agent string: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)	null referrer	no cookie reuse	7502.97	00.0%	00.4%	
IP: 43DB44548E1AB95696703B92DBBD778CB44EC4E4 User-agent string: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)	directory query as referrer (all starting with same letter)	no cookie reuse	246.12	00.2%	00.2%	
IP: 62A7DF089C1700ACF7A2B1A1614418E97C4ED42B User-agent string: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV...)	directory query as referrer (all starting with same letter)	no cookie reuse	430.32	00.1%	00.3%	
IP: 3E680C6B9A92070AB608671486DC332AD2B7083F User-agent string: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.0.1)	null referrer	reuse cookie for next 3 requests	9669.75	11.7%	03.3%	
IP: 2DC89E853294C1F0797002C4211FE005E39BEA52 User-agent string: Mozilla/4.0 (compatible; MSIE 5.0b2; Windows NT...)	only 1.6% of null referrers, rest is all public profiles	reuse cookie for next 100 requests	4.35	29.3%	00.3%	
IP: 3CBEF4B3A90AA47F6517260D78371EDBFE09E9A7 User-agent string: ia_archiver (+http://www.alex.com/site/help/webma...)	null referrer	reuse cookies for requests in a 15 min interval	59.42	02.7%	01.2%	

Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner

Adam Doupe, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna
University of California, Santa Barbara
{adoupe, cavedon, chris, vigna}@cs.ucsb.edu

Abstract

Black-box web vulnerability scanners are a popular choice for finding security vulnerabilities in web applications in an automated fashion. These tools operate in a *point-and-shoot* manner, testing any web application—regardless of the server-side language—for common security vulnerabilities. Unfortunately, black-box tools suffer from a number of limitations, particularly when interacting with complex applications that have multiple actions that can change the application’s state. If a vulnerability analysis tool does not take into account changes in the web application’s state, it might overlook vulnerabilities or completely miss entire portions of the web application.

We propose a novel way of inferring the web application’s internal state machine *from the outside*—that is, by navigating through the web application, observing differences in output, and incrementally producing a model representing the web application’s state.

We utilize the inferred state machine to drive a black-box web application vulnerability scanner. Our scanner traverses a web application’s state machine to find and fuzz user-input vectors and discover security flaws. We implemented our technique in a prototype crawler and linked it to the fuzzing component from an open-source web vulnerability scanner.

We show that our state-aware black-box web vulnerability scanner is able to not only exercise more code of the web application, but also discover vulnerabilities that other vulnerability scanners miss.

1 Introduction

Web applications are the most popular way of delivering services via the Internet. A modern web application is composed of a back-end, server-side part (often written in Java or in interpreted languages such as PHP, Ruby, or Python) running on the provider’s server, and a client

part running in the user’s web browser (implemented in JavaScript and using HTML/CSS for presentation). The two parts often communicate via HTTP over the Internet using Asynchronous JavaScript and XML (AJAX) [20].

The complexity of modern web applications, along with the many different technologies used in various abstraction layers, are the root cause of vulnerabilities in web applications. In fact, the number of reported web application vulnerabilities is growing sharply [18,41].

The occurrence of vulnerabilities could be reduced by better education of web developers, or by the use of security-aware web application development frameworks [10,38], which enforce separation between structure and content of input and output data. In both cases, more effort and investment in training is required, and, therefore, cost and time-to-market constraints will keep pushing for the current fast-but-insecure development model.

A complementary approach for fighting security vulnerabilities is to discover and patch bugs before malicious attackers find and exploit them. One way is to use a white-box approach, employing static analysis of the source code [4,15,17,24,28]. There are several drawbacks to a white-box approach. First, the potential applications that can be analyzed is reduced to only those applications that use the target programming language. In addition, there is the problem of substantial false positives. Finally, the source code of the application itself may be unavailable.

The other approach to discovering security vulnerabilities in web applications is by observing the application’s output in response to a specific input. This method of analysis is called *black-box* testing, as the application is seen as a sealed machine with unobservable internals. Black-box approaches are able to perform large-scale analysis across a wide range of applications. While black-box approaches usually have fewer false positives than white-box approaches, black-box approaches suffer

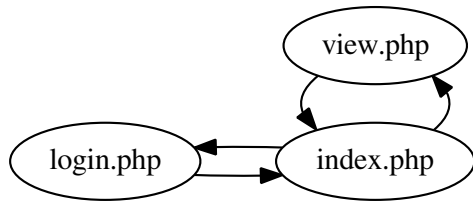


Figure 1: Navigation graph of a simple web application.

from a discoverability problem: They need to reach a page to find vulnerabilities on that page.

Classical black-box web vulnerability scanners crawl a web application to enumerate all reachable pages and then fuzz the input data (URL parameters, form values, cookies) to trigger vulnerabilities. However, this approach ignores a key aspect of modern web applications: Any request can change the state of the web application.

In the most general case, the state of the web application is any data (database, filesystem, time) that the web application uses to determine its output. Consider a forum that authenticates users, an e-commerce application where users add items to a cart, or a blog where visitors and administrators can leave comments. In all of these modern applications, the way a user interacts with the application determines the application's state.

Because a black-box web vulnerability scanner will never detect a vulnerability on a page that it does not see, scanners that ignore a web application's state will only explore and test a (likely small) fraction of the web application.

In this paper, we propose to improve the effectiveness of black-box web vulnerability scanners by increasing their capability to understand the web application's internal state. Our tool constructs a partial model of the web application's state machine in a fully-automated fashion. It then uses this model to fuzz the application in a state-aware manner, traversing more of the web application and thus discovering more vulnerabilities.

The main contributions of this paper are the following:

- A black-box technique to automatically learn a model of a web application's state.
- A novel vulnerability analysis technique that leverages the web application's state model to drive fuzzing.
- An evaluation of our technique, showing that both code coverage and effectiveness of vulnerability analysis are improved.

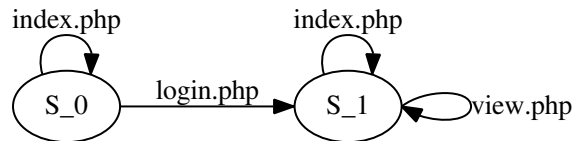


Figure 2: State machine of a simple web application.

2 Motivation

Crawling modern web applications means dealing with the web application's changing state. Previous work in detecting workflow violations [5, 11, 17, 30] focused on navigation, where a malicious user can access a page that is intended only for administrators. This unauthorized access is a violation of the developer's intended workflow of the application.

We wish to distinguish a navigation-based view of the web application, which is simply derived from crawling the web application, from the web application's internal state machine. To illustrate this important difference, we will use a small example.

Consider a simple web application that has only three pages, `index.php`, `login.php`, and `view.php`. The `view.php` page is only accessible after the `login.php` page is accessed. There is no logout functionality. A client accessing this web application might make a series of requests like the following:

```

<index.php, login.php, index.php, view.php,
index.php, view.php>
  
```

Analyzing this series of requests from a navigation perspective creates a navigation graph, shown in Figure 1. This graph shows which page is accessible from every other page, based on the navigation trace. However, the navigation graph does not represent the information that `view.php` is only accessible after accessing `login.php`, or that `index.php` has changed after requesting `login.php` (it includes the link to `view.php`).

What we are interested in is not how to navigate the web application, but how the requests we make influence the web application's internal state machine. The simple web application described previously has the internal state machine shown in Figure 2. The web application starts with the internal state `S_0`. Arrows from a state show how a request affects the web application's internal state machine. In this example, in the initial state, `index.php` does not change the state of the application, however, `login.php` causes the state to transition from `S_0` to `S_1`. In the new state `S_1`, both `index.php` and `view.php` do not change the state of the web application.

The state machine in Figure 2 contains important information about the web application. First, it shows that `login.php` permanently changes the web application's

state, and there is no way to recover from this change. Second, it shows that the `index.php` page is seen in two different states.

Now the question becomes: “How does knowledge of the web application’s state machine (or lack thereof) affect a black-box web vulnerability scanner?” The scanner’s goal is to find vulnerabilities in the application, and to do so it must fuzz as many execution paths of the server-side code as possible¹. Consider the simple application described in Figure 2. In order to fuzz as many code paths as possible, a black-box web vulnerability scanner must fuzz the `index.php` page in both states `S_0` and `S_1`, since the code execution of `index.php` can follow different code paths depending on the current state (more precisely, in state `S_1`, `index.php` includes a link to `view.php`, which is not present in `S_0`).

A black-box web vulnerability scanner can also use the web application’s state machine to handle requests that change state. For example, when fuzzing the `login.php` page of the sample application, a fuzzer will try to make several requests to the page, fuzzing different parameters. However, if the first request to `login.php` changes the state of the application, all further requests to `login.php` will no longer execute along the same code path as the first one. Thus, a scanner must have knowledge of the web application’s state machine to test if the state was changed, and if it was, what requests to make to return the application to the previous state before continuing the fuzzing process.

We have shown how a web application’s state machine can be leveraged to improve a black-box web vulnerability scanner. Our goal is to infer, in a black-box manner, as much of the web application’s state machine as possible. Using only the sequence of requests, along with the responses to those requests, we build a model of as much of the web application’s state machine as possible. In the following section, we describe, at a high level, how we infer the web application’s state machine. Then, in Section 4, we provide the details of our technique.

3 State-Aware Crawling

In this section, we describe our state-aware crawling approach. In Section 3.1, we describe web applications and define terms that we will use in the rest of the paper. Then, in Section 3.2, we describe the various facets of the state-aware crawling algorithm at a high level.

¹Hereinafter, we assume that the scanner relies on fuzzer-based techniques. However, any other automated vulnerability analysis technique would benefit from our state-aware approach.

3.1 Web Applications

Before we can describe our approach to inferring a web application’s state, we must first define the elements that come into play in our web application model.

A web application consists of a server component, which accepts HTTP requests. This server component can be written in any language, and could use many different means of storage (database, filesystem, memcache). After processing a request, the server sends back a response. This response encapsulates some content, typically HTML. The HTML content contains links and forms which describe how to make further requests.

Now that we have described a web application at a high level, we need to define specific terms related to web applications that we use in the rest of this paper.

- Request—The HTTP request made to the web application. Includes anything (typically in the form of HTTP headers) that is sent by the user to the web application: the HTTP Method, URL, Parameters (GET and POST), Cookies, and User-Agent.
- Response—The response sent by the server to the user. Includes the HTTP Response Code and the content (typically HTML).
- Page—The HTML page that is contained in the response from a web application.
- Link—Element of an HTML page that tells the browser how to create a subsequent request. This can be either an anchor or a form. An anchor always generates a GET request, but a form can generate either a POST or GET request, depending on the parameters of the form.
- State—Anything that influences the web application’s server-side code execution.

3.1.1 Web Application Model

We use a *symbolic Mealy machine* [7] to model the web application as a black-box. A Mealy machine is an automaton where the input to the automaton, along with the current state, determines the output (i.e., the page produced by the response) and the next state. A Mealy machine operates on a finite alphabet of input and output symbols, while a symbolic Mealy machine uses an infinite alphabet of input and output symbols.

This model of a web application works well because the input to a web application, along with the current state of the web application, determines the output and the next state. Consider a simple e-commerce web application with the state machine shown in Figure 3. In this state graph, all requests except for the ones leaving a state

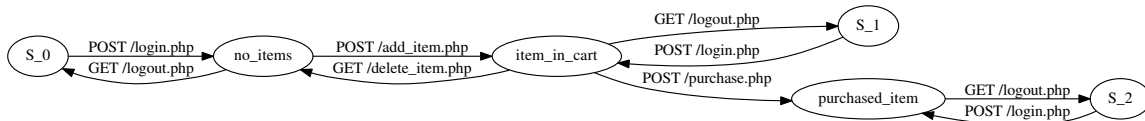


Figure 3: The state machine of a simple e-commerce application.

bring the application back to the same state. Therefore, this state graph does not show all the request that can be made to the application, only the subset of requests that change the state.

For instance, in the initial state `S_0`, there is only one request that will change the state of the application, namely `POST /login.php`. This change logs the user into the web application. From the state `no_items`, there are two requests that can change the state, `GET /logout.php` to return the user to state `S_0` and `POST /add_item.php` to add an item to the user's shopping cart.

Note that the graph shown in Figure 3 is not a strongly connected graph—that is, every state cannot be reached by every other state. In this example, purchasing an item is a permanent action, it irrecoverably changes the state (there is no link from `purchased_item` to `item_in_cart`). Another interesting aspect is that one request, `GET /logout.php`, leads to three different states. This is because once the web application's state has changed, logging out, and then back in, does not change the state of the cart.

3.2 Inferring the State Machine

Inferring a web application's state machine requires the ability to detect when the state of the web application has changed. Therefore, we start with a description of the state-change detection algorithm, then explain the other components that are required to infer the state machine.

The key insight of our state-change algorithm is the following: We detect that the state of the web application has changed when we make an identical request and get a different response. This is the only externally visible effect of a state-change: Providing the same input causes a different output.

Using this insight, our state-change detection algorithm works, at a high level, as follows: (1) Crawl the web application sequentially, making requests based on a link in the previous response. (2) Assume that the state stays the same, because there is no evidence to the contrary. (3) If we make a request identical to a previous request and get a different response, then we assume that some request since the last identical request changed the state of the web application.

The intuition here is that a Mealy machine will, when given the same input in the same state, produce the same output. Therefore, if we send the same request and get a different output, the state must have changed. By detecting the web application's state changes only using inputs and outputs, we are agnostic with respect to both *what* constitutes the state information and *where* the state information is located. In this way, we are more generic than approaches that only consider the database to hold the state of the application, when in fact, the local file system or even memory could hold part of the web application's state.

The state-change detection algorithm allows us to infer when the web application's state has changed, yet four other techniques are necessary to infer a state machine: the clustering of similar pages, the identification of state-changing requests, the collapsing of similar states, and navigating.

Clustering similar pages. We want to group together pages that are similar, for two reasons: To handle infinite sections of web applications that are generated from the same code (e.g., the pages of a calendar) and to detect when a response has changed.

Before we can cluster pages, we model them using the links (anchors and forms) present on the page. The intuition here is that the links describe *how* the user can interact with the web application. Therefore, changes to what a user can do (new or missing links) indicate when the state of the web application has changed. Also, infinite sections of a web application will share the same link structure and will cluster together.

With our page model, we cluster pages together based on their link structure. Pages that are in different clusters are considered different. The details of this approach are described in Section 4.1.

Determining the state-changing request. The state-change detection algorithm only says that the state has changed, however we need to determine *which* request actually changed the state. When we detect a state change, we have a temporal list of requests with identical requests at the start and end. One of the requests in this list changed the state. We use a heuristic to determine which request changed the state. This heuristic favors newer requests over older requests, `POST` requests over `GET` requests, and requests that have previously changed

the state over those that have never changed the state. The details are described in Section 4.2.

Collapsing similar states. The state-change detection algorithm detects only when the state has changed, however, we need to understand if we returned to a previous state. This is necessary because if we detect a state change, we want to know if this is a state we have previously seen or a brand new state. We reduce this problem to a graph coloring problem, where the nodes are the states and an edge between two nodes means that the states cannot be the same. We add edges to this graph by using the requests and responses, along with rules to determine when two states cannot be the same. After the graph is colored, states that are the same color are collapsed into the same state. Details of this state-merging technique are provided in Section 4.3.

Navigating. We have two strategies for crawling the web application.

First, we always try to pick a link in the last response. The rationale behind choosing a link in the last response is that we emulate a user browsing the web application. In this way, we are able to handle multi-step processes, such as previewing a comment before it is committed.

Second, for each state, we make requests that are the least likely to change the state of the web application. The intuition here is that we want to first see as much of a state as possible, without accidentally changing the state, in case the state change is permanent. Full details of how we crawl the web application are provided in Section 4.4

4 Technical Details

Inferring a web application’s state machine requires concretely defining aspects such as page clustering or navigation. However, we wish to stress that this is one implementation of the state machine inference algorithm and it may not be optimal.

4.1 Clustering Similar Pages

Our reason for grouping similar pages together is twofold: Prevent infinite scanning of the website by grouping the “infinite” areas together and detect when the state has changed by comparing page responses in an efficient manner.

4.1.1 Page Model

The output of a web application is usually an HTML document (it can actually be any arbitrary content, but we only consider HTML content and HTTP redirects). An HTML page is composed of navigational information (anchors and forms) and user-readable content. For

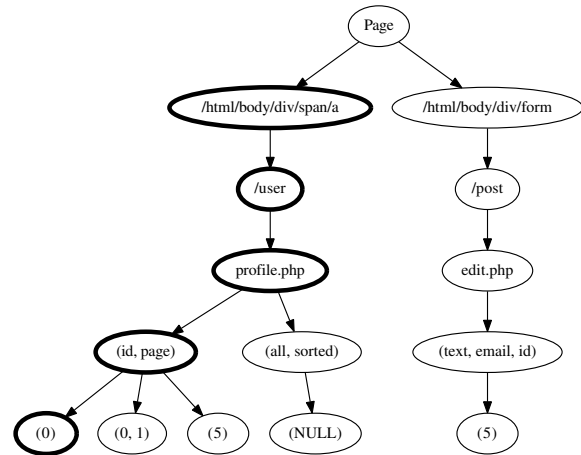


Figure 4: Representation of a page’s link vectors stored in a prefix tree. There are five links present on this tree, as evidenced by the number of leaf nodes.

our state-change detection algorithm, we are not interested in changes to the content, but rather to changes in the navigation structure. We focus on navigation changes because the links on a page define how a user can interact with the application, thus, when the links change, the web application’s state has changed.

Therefore, we model a page by composing all the anchors and forms. First, every anchor and form is transformed into a vector constructed as follows:

$$\langle dompath, action, params, values \rangle$$

where:

- *dompath* is the DOM (*Document Object Model*) path of the HTML link (anchor or form);
- *action* is a list where each element is from the `href` (for anchors) or `action` (for forms) attribute split by ‘/’;
- *params* is the (potentially empty) set of parameter names of the form or anchor;
- *values* is the set of values assigned to the parameters listed in *params*.

For instance, an anchor tag with the `href` attribute of `/user/profile.php?id=0&page` might have the following link vector:

$$\langle /html/body/div/span/a, /user, profile.php, (id, page), (0) \rangle$$

All link vectors of a page are then stored in a prefix tree. This prefix tree is the model of the page. A prefix tree for a simple page with five links is shown in Figure 4. The link vector previously described is highlighted in bold in Figure 4.

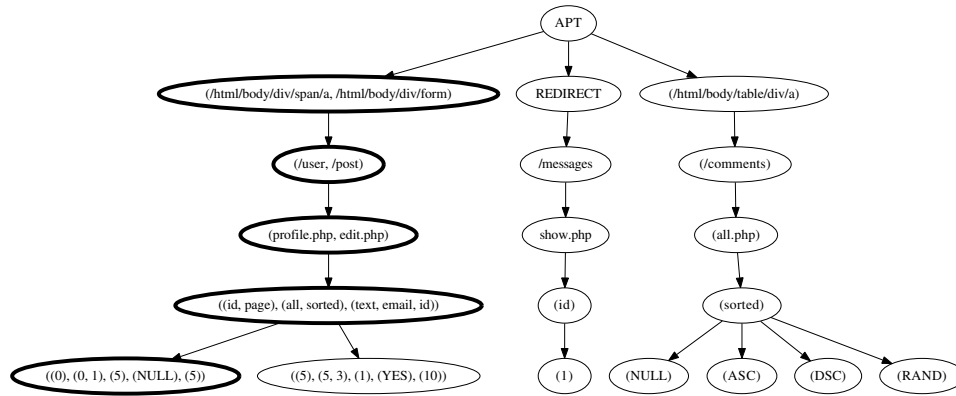


Figure 5: Abstract Page Tree. Every page’s link vector is stored in this prefix tree. There are seven pages in this tree. The page link vector from Figure 4 is highlighted in bold.

HTTP redirects are handled as a special case, where the only element is a special *redirect* element having the target URL as the value of the *location* attribute.

4.1.2 Page Clustering

To cluster pages, we use a simple but efficient algorithm. As described in the previous section, the model of a page is a prefix tree representing all the links contained in the page.

These prefix trees are translated into vectors, where every element of this vector is the set of all nodes of a given level of the prefix tree, starting from the root. At this point, all pages are represented by a *page link vector*. For example, Figure 4 has the following page link vector:

```
((/html/body/div/span/a, /html/body/div/form),
 (/user, /post),
 (profile.php, edit.php),
 ((id, page), (all, sorted), (text, email, id)),
 ((0), (0, 1), (5), (NULL), (5)))
```

The page link vectors for all pages are then stored in another prefix tree, called the *Abstract Page Tree (APT)*. In this way, pages are mapped to a leaf of the tree. Pages which are mapped to the same leaf have identical page link vectors and are considered to be the same page. Figure 5 shows an APT with seven pages. The page from Figure 4 is bold in Figure 5.

However, we want to cluster together pages whose page link vectors do not match exactly, but are similar (e.g., shopping cart pages with a different number of elements in the cart). A measure of the similarity between two pages is how many elements from the beginning of their link vectors are the same between the two pages. From the APT perspective, the higher the number of ancestors two pages (leaves) share, the closer they are.

Therefore, we create clusters of similar pages by selecting a node in the APT and merging into one cluster, called an *Abstract Page*, all the leaves in the corresponding subtree. The criteria for deciding whether to cluster a subtree of depth n from the root is the following:

- The number of leaves is greater than the median number of leaves of all its siblings (including itself); in this way, we cluster only subtrees which have a larger-than-usual number of leaves.
- There are at least $f(n)$ leaves in the subtree, where $f(n)$ is inversely related to n . The intuition is that the fewer ancestors a subtree has in common (the higher on the prefix tree it is), the more pages it must have to cluster them together. We have found that the function $f(n) = 8(1 + \frac{1}{n+1})$ works well by experimental analysis on a large corpus of web pages.
- The pages share the same *domepath* and the first element of the *action* list of the page link vector; in this way, all the pages that are clustered together share the same link structure with potentially different parameters and values.

4.2 Determine the State-Changing Request

When a state change is detected, we must determine which request actually changed the web application’s state. Recall that we detect a state change when we make a request that is identical to a previous request, yet has different output. At this point, we have a list of all the requests made between the latest request R and the request R' closest in time to R such that R is identical to R' . We use a heuristic to determine which request in this list changed the web application’s state, choosing the request i between R' and R which maximizes the function:

$$\text{score}(n_{i,transition}, n_{i,seen}, distance_i)$$

where:

- $n_{i,transition}$ is the number of times the request caused a state transition;
- $n_{i,seen}$ is the number of times the request has been made;
- $distance_i$ is how many requests have been made between request R and request i .

The function *score* is defined as:

$$score(n_{i,transition}, n_{i,seen}, distance_i) = 1 - \left(1 - \frac{n_{i,transition}+1}{n_{i,seen}+1}\right)^2 + \frac{BOOST_i}{distance_i+1}$$

$BOOST_i$ is .2 for POST requests and .1 for GET requests.

We construct the *score* function to capture two properties of web applications:

1. A POST request is more likely to change the state than a GET request. This is suggested by the HTTP specification, and *score* captures this intuition with $BOOST_i$.
2. Resistant to errors. Because we cannot prove that the selected request changed the state, we need to be resistant to errors. That is why *score* contains the ratio of $n_{i,transition}$ to $n_{i,seen}$. In this way, if we accidentally choose the wrong state-changing request once, but then, later, make that request many times without changing the state, we are less likely to choose it as a state-changing request.

4.3 Collapsing Similar States

Running the state detection algorithm on a series of requests and responses will tell us when the state has changed. At this point, we consider each state unique. This initial state assignment, though, is not optimal, because even if we encounter a state that we have seen in the past, we are marking it as new. For example, in the case of a sequence of login and logout actions, we are actually flipping between two states, instead of entering a new state at every login/logout. Therefore, we need to minimize the number of different states and collapse states that are actually the same.

The problem of state allocation can be seen as a graph-coloring problem on a non-planar graph [27]. Let each state be a node in the graph G . Let two nodes a and b be connected by an edge (meaning that the states cannot be the same) if either of the following conditions holds:

1. If a request R was made when the web application was in states a and b and results in pages in different clusters. The intuition is that two states cannot be

the same if we make an identical request in each state yet receive a different response.

2. The two states a and b have no pages in common. The idea is to err on the conservative side, thus we require that two states share a page before collapsing the states into one.

After adding the edges to the graph by following the previous rules, G is colored. States assigned the same color are considered the same state.

To color the nodes of G , we employ a custom greedy algorithm. Every node has a unique identifier, which is the incremental number of the state as we see it in the request-response list. The nodes are ordered by identifier, and we assign the color to each node in a sequential way, using the highest color available (i.e., not used by its neighbors), or a new color if none is available.

This way of coloring the nodes works very well for state allocation because it takes into account the temporal locality of states: In particular, we attempt to assign the highest available color because it is more likely for a state to be the same as a recently seen state rather than one seen at the beginning of crawling.

There is one final rule that we need to add after the graph is colored. This rule captures an observation about transitioning between states: If a request, R , transitions the web application from state a_1 to state b , yet, later when the web application is in state a_2 , R transitions the web application to state c , then a_1 and a_2 cannot be the same state. Therefore, we add an edge from a_1 to a_2 and redo the graph coloring.

We continue enforcing this rule until no additional edges are added. The algorithm is guaranteed to converge because only new edges are added at every step, and no edges are ever removed.

At the end of the iteration, the graph coloring output will determine the final state allocation—all nodes with the same color represent the same state (even if seen at different stages during the web application crawling process).

4.4 Navigating

Typical black-box web vulnerability scanners make concurrent HTTP requests to a web application to increase performance. However, as we have shown, an HTTP request can influence the web application's state, and, in this case, all other requests would occur in the new state. Also, some actions require a multi-step, sequential process, such as adding items to a shopping cart before purchasing them. Finally, a user of the web application does not browse a web application in this parallel fashion, thus, developers assume that the users will browse sequentially.

```
def fuzz_state_changing( fuzz_request ):
    make_request( fuzz_request )
    if state_has_changed():
        if state_is_reversible():
            make_requests_to_revert_state()
            if not back_in_previous_state():
                reset_and_put_in_previous_state()
        else:
            reset_and_put_in_previous_state()
```

Listing 1: Pseudocode for fuzzing state-changing request.

Our scanner navigates a web application by mimicking a user browsing the web application sequentially. Browsing sequentially not only allows us to follow the developer’s intended path through the web application, but it enables us to detect *which* requests changed the web application’s state.

Thus, a state-aware crawler must navigate the application sequentially. No concurrent requests are made, and only anchors and forms present in the last visited page are used to determine the next request. In the case of a page with no outgoing links we go back to the initial page.

Whenever the latest page does not contain unvisited links, the crawler will choose a path from the current page towards another page already seen that contains links that have not yet been visited. If there is no path from the current page to anywhere, we go back to the initial page. The criteria for choosing this path is based on the following intuitions:

- We want to explore as much of the current state as possible before changing the state, therefore we select links that are less likely to cause a state transition.
- When going from the current page to a page with an unvisited link, we will repeat requests. Therefore, we should choose a path that contains links that we have visited infrequently. This give us more information about the current state.

The exact algorithm we employ is Dijkstra Shortest Path Algorithm [14] with custom edge length. This edge length increases with the number of times we have previously visited that link. Finally, the edge length increases with how likely the link is to cause a state change.

5 State-Aware Fuzzing

After we crawl the web application, our system has inferred, as much as possible, the web application’s state machine. We use the state machine information, along with the list of request–responses made by the crawler, to

drive a state-aware fuzzing of the web application, looking for security vulnerabilities.

To fuzz the application in a state-aware manner, we need the ability to reset the web application to the initial state (the state when we started crawling). We do not use this ability when crawling, only when fuzzing. It is necessary to reset the application when we are fuzzing an irreversible state-changing request. Using the reset functionality, we are able to recover from these irreversible state changes.

Adding the ability to reset the web application does not break the black-box model of the web application. Resetting requires no knowledge of the web application, and can be easily performed by running the web application in a virtual machine.

Our state-aware fuzzing starts by resetting the web application to the initial state. Then we go through the requests that the crawler made, starting with the initial request. If the request does not change the state, then we fuzz the request as a typical black-box scanner. However, if the request is state-changing, we follow the algorithm shown in Listing 1. The algorithm is simple: We make the request, and if the state has changed, traverse the inferred state machine to find a series of requests to transition the web application to the previous state. If this does not exist, or does not work, then we reset the web application to the initial state, and make all the previous requests that the crawler made. This ensures that the web application is in the proper state before continuing to fuzz.

Our state-aware fuzzing approach can use *any* fuzzing component. In our implementation, we used the fuzzing plugins of an open-source scanner, w3af [37]. The fuzzing plugins take an HTTP request and generate variations on that request looking for different vulnerabilities. Our state-aware fuzzing makes those requests while checking that the state does not unintentionally change.

6 Evaluation

As shown in previous research [16], fairly evaluating black-box web vulnerability scanners is difficult. The most important, at least to end users, metric for comparing black-box web vulnerability scanners is true vulnerabilities discovered. Comparing two scanners that discover different vulnerabilities is nearly impossible.

There are two other metrics that we use to evaluate black-box web vulnerability scanners:

- False Positives. The number of spurious vulnerabilities that a black-box web vulnerability scanner reports. This measures the accuracy of the scanner. False positives are a serious problem for the end user of the scanner—if the false positives are

Application	Description	Version	Lines of Code
Gallery	Photo hosting.	3.0.2	26,622
PhpBB v2	Discussion forum.	2.0.4	16,034
PhpBB v3	Discussion forum.	3.0.10	110,186
SCARF	Stanford conference and research forum.	2007-02-27	798
Vanilla Forums	Discussion forum.	2.0.17.10	43,880
WackoPicko v2	Intentionally vulnerable web application.	2.0	900
WordPress v2	Blogging platform.	2.0	17,995
WordPress v3	Blogging platform.	3.2.1	71,698

Table 1: Applications that we ran the crawlers against to measure vulnerabilities discovered and code coverage.

high, the user must manually inspect each vulnerability reported to determine the validity. This requires a security-conscious user to evaluate the reports. Moreover, false positives erode the user's trust in the tool and make the user less likely to use it in the future.

- **Code Coverage.** The percentage of the web application's code that the black-box web vulnerability scanner executes while it crawls and fuzzes the application. This measures how effective the scanner is in exercising the functionality of the web application. Moreover, code coverage is an excellent metric for another reason: A black-box web vulnerability scanner, by nature, cannot find a vulnerability along a code path that it does not execute. Therefore, greater code coverage means that a scanner has the potential to discover more vulnerabilities. Note that this is orthogonal to fuzzing capability: A fuzzer—no matter how effective—will never be able to discover a vulnerability on a code path that it does not execute.

We use both the metrics previously described in our evaluation. However, our main focus is on code coverage. This is because a scanner with greater code coverage will be able to discover more vulnerabilities in the web application.

However, code coverage is not a perfect metric. Evaluating raw code coverage percentage numbers can be misleading. Ten percent code coverage of an application could be horrible or excellent depending on how much functionality the application exposes. Some code may be intended only for installation, may be only for administrators, or is simply dead code and cannot be executed. Therefore, comparing code coverage normalized to a baseline is more informative, and we use this in our evaluation.

6.1 Experiments

We evaluated our approach by running our state-aware-scanner along with three other vulnerability scanners

Scanner	Description	Language	Version
wget	GNU command-line website downloader.	C	1.12
w3af	Web Application Attack and Audit Framework.	Python	1.0-stable
skipfish	Open-source, high-performance vulnerability scanner.	C	2.03b
state-aware-scanner	Our state-aware vulnerability scanner.	Python	1.0

Table 2: Black-box web vulnerability scanners that we compared.

against eight web applications. These web applications range in size, complexity, and functionality. In the rest of this section, we describe the web applications, the black-box web vulnerability scanners, and the methodology we used to validate our approach.

6.1.1 Web Applications

Table 1 provides an overview of the web applications used with a short description, a version number, and lines of executable PHP code for each application. Because our approach assumes that the web application's state changes only via requests from the user, we made slight code modifications to three web applications to reduce the influence of external, non-user driven, forces, such as time. Please refer to Appendix A for a detailed description of each application and what was changed.

6.1.2 Black-Box Web Vulnerability Scanners

This section describes the black-box web vulnerability scanners that were compared against our approach, along with the configuration or settings that were used. Table 2 contains a short description of each scanner, the scanner's programming language, and the version number. Appendix B shows the exact configuration that was used for each scanner.

wget is a free and open-source application that is used to download files from a web application. While not a vulnerability scanner, wget is a crawler that will make all possible GET requests it can find. Thus, it provides an excellent baseline because vulnerability scanners make POST requests as well as GET requests and should discover more of the application than wget.

wget is launched with the following options: recursive, download everything, and ignore robots.txt.

w3af is an open-source black-box web vulnerability scanner which has numerous fuzzing modules. We enabled the blindSqli, eval, localFileInclude, osCommanding, remoteFileInclude, sqli, and xss fuzzing plugins.

skipfish is an open-source black-box web vulnerability scanner whose focus is on high speed and high performance. Skipfish epitomizes the “shotgun” approach, and boasts about making more than 2,000 requests per second to a web application on a LAN. Skipfish also attempts to guess, via a dictionary or brute-force, directory names. We disabled this behavior to be fair to the other scanners, because we do not want to test the ability to guess a hidden directory, but how a scanner crawls a web application.

state-aware-scanner is our state-aware black-box vulnerability scanner. We use HtmlUnit [19] to issue the HTTP requests and render the HTML responses. After crawling and building the state-graph, we utilize the fuzzing plugins from w3af to generate fuzzing requests. Thus, any improvement in code coverage of our crawler over w3af is due to our state-aware crawling, since the fuzzing components are identical.

6.1.3 Methodology

We ran each black-box web vulnerability scanner against a distinct, yet identical, copy of each web application. We ran all tests on our local cloud [34].

Gallery, WordPress v2, and WordPress v3 do not require an account to interact with the website, thus each scanner is simply told to scan the test application.

For the remaining applications (PhpBB v2, PhpBB v3, SCARF, Vanilla Forums, and WackoPicko v2), it is difficult to fairly determine how much information to give the scanners. Our approach only requires a username/password for the application, and by its nature will discover the requests that log the user out, and recover from them. However, other scanners do not have this capability.

Thus, it is reasonable to test all scanners with the same level of information that we give our scanner. However, the other scanners lack the ability to provide a username and password. Therefore, we did the next best thing: For those applications that require a user account, we log into the application and save the cookie file. We then instruct

the scanner to use this cookie file while scanning the web application.

While we could do more for the scanners, like preventing them from issuing the logout request for each application, we believe that our approach strikes a fair compromise and allows each scanner to decide how to crawl the site. Preventing the scanners from logging out of the application also limits the amount of the application they will see, as they will never see the web application from a guest’s perspective.

6.2 Results

Table 3 shows the results of each of the black-box web vulnerability scanners against each web application. The column “% over Baseline” displays the percentage of code coverage improvement of the scanner against the wget baseline, while the column “Vulnerabilities” shows total number of reported vulnerabilities, true positives, unique true positives among the scanners, and false positives.

The prototype implementation of our state-aware-scanner had the best code coverage for every application. This verifies the validity of our algorithm: Understanding state is necessary to better exercise a web application.

Figure 6 visually displays the code coverage percent improvement over wget. The most important thing to take from these results is the improvement state-aware-scanner has over w3af. Because we use the fuzzing component of w3af, the only difference is in our state-aware crawling. The results show that this gives state-aware-scanner an increase in code coverage from as little as half a percent to 140.71 percent.

Our crawler discovered three unique vulnerabilities, one each in PhpBB v2, SCARF, and WackoPicko v2. The SCARF vulnerability is simply a XSS injection on the comment form. w3af logged itself out before fuzzing the comment page. skipfish filed the vulnerable page under “Response varies randomly, skipping checks.” However, the content of this page does not vary randomly, it varies because skipfish is altering it. This *random* categorization also prevents skipfish from detecting the simple XSS vulnerability on WackoPicko v2’s guestbook. This result shows that a scanner needs to understand the web application’s internal state to properly decide *why* a page’s content is changing.

Skipfish was able to discover 15 vulnerabilities in Vanilla Forums. This is impressive, however, 14 stem from a XSS injection via the referer header on an error page. Thus, even though these 14 vulnerabilities are on different pages, it is the same root cause.

Surprisingly, our scanner produced less false positives than w3af. All of w3af’s false positives were due to faulty timing detection of SQL injection and OS com-

Scanner	Application	% over Baseline	Vulnerabilities			
			Reported	True	Unique	False
state-aware-scanner	Gallery	16.20%	0	0	0	0
w3af	Gallery	15.77%	3	0	0	3
skipfish	Gallery	10.96%	0	0	0	0
wget	Gallery	0%				
state-aware-scanner	PhpBB v2	38.34%	4	3	1	1
skipfish	PhpBB v2	5.10%	3	2	0	1
w3af	PhpBB v2	1.04%	5	1	0	4
wget	PhpBB v2	0%				
state-aware-scanner	PhpBB v3	115.45%	0	0	0	0
skipfish	PhpBB v3	60.21%	2	0	0	2
w3af	PhpBB v3	16.16%	0	0	0	0
wget	PhpBB v3	0%				
state-aware-scanner	SCARF	67.03%	1	1	1	0
skipfish	SCARF	55.66%	0	0	0	0
w3af	SCARF	21.55%	0	0	0	0
wget	SCARF	0%				
state-aware-scanner	Vanilla Forums	30.89%	0	0	0	0
w3af	Vanilla Forums	1.06%	0	0	0	0
wget	Vanilla Forums	0%				
skipfish	Vanilla Forums	-2.32%	17	15	2	2
state-aware-scanner	WackoPicko v2	241.86%	5	5	1	0
skipfish	WackoPicko v2	194.77%	4	3	1	1
w3af	WackoPicko v2	101.15%	5	5	1	0
wget	WackoPicko v2	0%				
state-aware-scanner	WordPress v2	14.49%	0	0	0	0
w3af	WordPress v2	12.49%	0	0	0	0
wget	WordPress v2	0%				
skipfish	WordPress v2	-18.34%	1	0	0	1
state-aware-scanner	WordPress v3	9.84%	0	0	0	0
w3af	WordPress v3	9.23%	3	0	0	3
skipfish	WordPress v3	3.89%	1	0	0	1
wget	WordPress v3	0%				

Table 3: Results of each of the black-box web vulnerability scanners against each application. The table is sorted by the percent increase in code coverage over the baseline scanner, wget.

manding. We believe that using HtmlUnit prevented our scanner from detecting these spurious vulnerabilities, even though we use the same fuzzing component as w3af.

Finally, our approach inferred the state machines of the evaluated applications. These state machines are very complex in the large applications. This complexity is because modern, large, application have many actions which modify the state. For instance, in WackoPicko v2, a user can log in, add items to their cart, comment on pictures, delete items from their cart, log out of the application, register as a new user, comment as this new user, upload a picture, and purchase items. All of these actions interact to form a complex state machine. The state machine our scanner inferred captures this complex series of state changes. The inferred WackoPicko v2 state machine is presented in Figure 7.

7 Limitations

Although dynamic page generation via JavaScript is supported by our crawler as allowed by the HtmlUnit framework [19], proper AJAX support is not implemented. This means that our prototype executes JavaScript when the page loads, but does not execute AJAX calls when clicking on links.

Nevertheless, our approach could be extended to handle AJAX requests. In fact, any interaction with the web application always contains a request and response, however the content of the response is no longer an HTML page. Thus, we could extend our notion of a “page” to typical response content of AJAX calls, such as JSON or XML. Another way to handle AJAX would be to follow a Crawljax [33] approach and covert the dynamic AJAX calls into static pages.

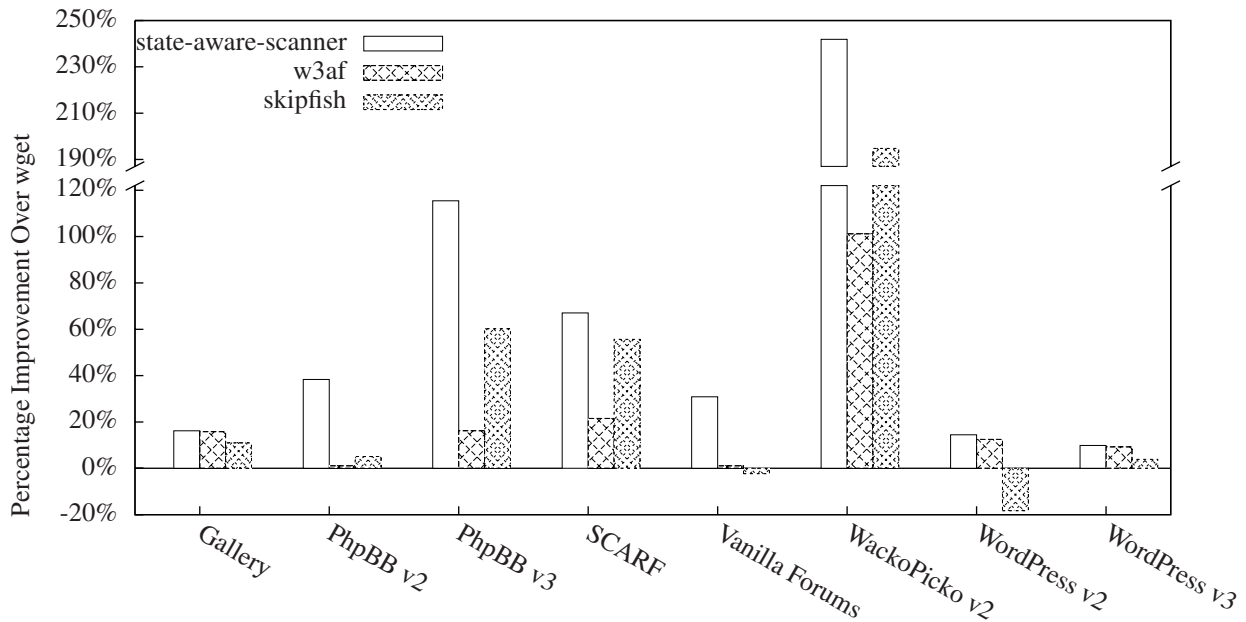


Figure 6: Visual representation of the percentage increase of code coverage over the baseline scanner, wget. Important to note is the gain our scanner, state-aware-scanner, has over w3af, because the only difference is our state-aware crawling. The y-axis range is broken to reduce the distortion of the WackoPicko v2 results.

Another limitation of our approach is that our scanner cannot be used against a web application being accessed by other users (i.e., a public web application), because the other users may influence the state of the application (e.g., add a comment on a guestbook) and confuse our state change detection algorithm.

8 Related Work

Automatic or semi-automatic web application vulnerability scanning has been a hot topic in research for many years because of its relevance and its complexity.

Huang et al. developed a tool (WAVES) for assessing web application security with which we share many points [24]. Similarly to us, they have a scanner for finding the entry points in the web application by mimicking the behavior of a web browser. They employ a learning mechanism to sensibly fill web form fields and allow deep crawling of pages behind forms. Attempts to discover vulnerabilities are carried out by submitting the same form multiple times with valid, invalid, and faulty inputs, and comparing the result pages. Differently from WAVES, we are using the knowledge gathered by the first-phase scanner to help the fuzzer detect the effect of a given input. Moreover, our first-phase scanner aims not only at finding relevant entry-points, but rather at building a complete state-aware navigational map of the web application.

A number of tools have been developed to try to automatically discover vulnerabilities in web applications, produced as academic prototypes [4, 17, 22, 25, 28, 29, 31], as open-source projects [8, 9, 37], or as commercial products [1, 23, 26, 35].

Multiple projects [6, 16, 42, 43] tackled the task of evaluating the effectiveness of popular black-box scanners (in some cases also called *point-and-shoot* scanners). The common theme in their results is a relevant discrepancy in vulnerabilities found across scanners, along with low accuracy. Authors of these evaluations acknowledge the difficulties and challenges of the task [21, 43]. In particular, we highlighted how more deep crawling and reverse engineering capabilities of web applications are needed in black-box scanners, and we also developed the *WackoPicko* web application which contains known vulnerabilities [16]. Similarly, Bau et al. investigated the presence of room for research in this area, and found improvement is needed, in particular for detecting second-order XSS and SQL injection attacks [6].

Reverse engineering of web applications has not been widely explored in the literature, to our knowledge. Some approaches [13] perform static analysis on the code to create UML diagrams of the application.

Static analysis, in fact, is the technique mostly employed for automatic vulnerability detection, often combined with dynamic analysis.

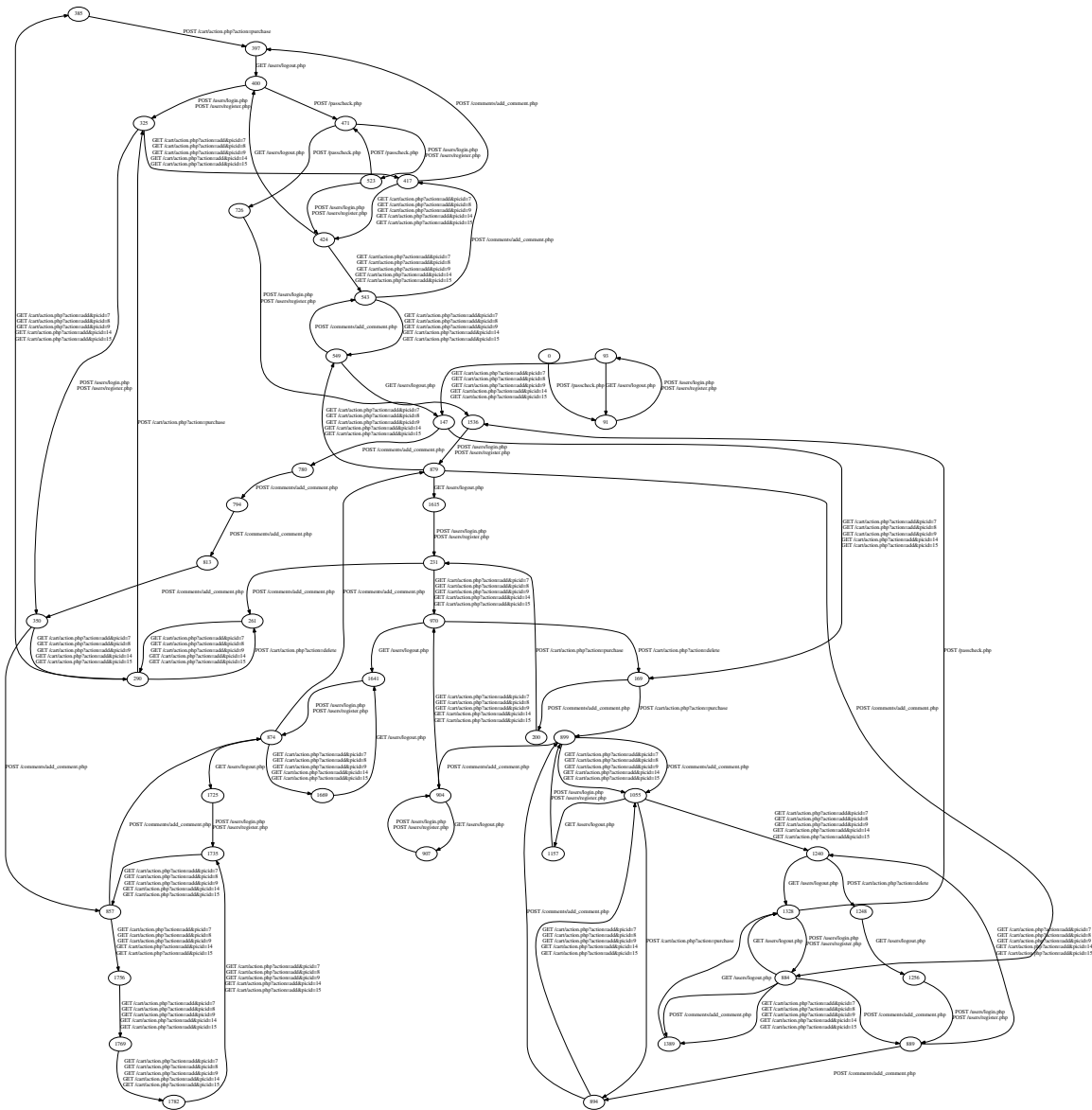


Figure 7: State machine that state-aware-scanner inferred for WackoPicko v2.

Halfond et al. developed a traditional black-box vulnerability scanner, but improved its result by leveraging a static analysis technique to better identify input vectors [22].

Pixy [28] employed static analysis with taint propagation in order to detect SQL injection, XSS and shell command injection, while *Saner* [4] used sound static analysis to detect failures in sanitization routines. *Saner* also takes advantage of a second phase of dynamic analysis to reduce false positives. Similarly, *WebSSARI* [25] also employed static analysis for detecting injection vulnerabilities, but, in addition, it proposed a technique for runtime instrumentation of the web application through the insertion of proper sanitization routines.

Felmetzger et al. investigated an approach for detecting a different type of vulnerability (some categories of logic flaws) by combining execution traces and symbolic model checking [17]. Similar approaches are also used for generic bug finding (in fact, vulnerabilities could be considered a subset of the general bug category). Csallner et al. employ dynamic traces for bug finding and for dynamic verification of the alerts generated by the static analysis phase [12]. Artzi et al., on the other hand, use symbolic execution and model checking for finding general bugs in web applications [3].

On a completely separate track, efforts to improve web application security push the developers toward writing secure code in the first place. Security experts are ty-

ing to achieve this goal by either educating the developers [40] or designing frameworks which prohibit the use of bad programming practices and enforce some security constraints in the code. Robertson and Vigna developed a strongly-typed framework which statically enforces separation between structure and content of a web page, preventing XSS and SQL injection [38]. Also Chong et al. designed their language for developers to build web applications with strong confidentiality and integrity guarantees, by means of compile-time and run-time checks [10].

Alternatively, consequences of vulnerabilities in web applications can be mitigated by trying to prevent the attacks before they reach some potentially vulnerable code, like, for example, in the already mentioned *Web-SSARI* [25] work. A different approach for blocking attacks is followed by Scott and Sharp, who developed a language for specifying a security policy for the web application; a gateway will then enforce these policies [39].

Another interesting research track deals with the problem of how to explore web pages behind forms, also called the *hidden web* [36]. McAllister et al. monitor user interactions with a web application to collect sensible values for HTML form submission and generate test cases that can be replayed to increase code coverage [32]. Although not targeted to security goals, the work of Raghavan and Garcia-Molina is relevant for our project for their contribution in classification of different types of dynamic content and for their novel approach for automatically filling forms by deducing the domain of form fields [36]. Raghavan and Garcia-Molina carried out further research in this direction, by reconstructing complex and hierarchical query interfaces exposed by web applications.

Moreover, Amalfitano et al. started tackling the problem of reverse engineering the state machine of client-side AJAX code, which will help in finding the web application server-side entry points and in better understanding complex and hierarchical query interfaces [2].

Finally, we need to mention the work by Berg et al. in reversing state machines into a *Symbolic Mealy Machine* (SMM) model [7]. Their approach for reversing machines cannot be directly applied to our case because of the infeasibility of fully exploring all pages for all the states, even for a small subset of the possible states. Nevertheless, the model they propose for a SMM fits our needs.

9 Conclusion

We have described a novel approach to inferring, as much as possible, a web application's internal state machine. We leveraged the state machine to drive the state-aware fuzzing of web applications. Using this approach,

our crawler is able to crawl—and thus fuzz—more of the web application than a classical state-agnostic crawler. We believe our approach to detecting state change by differences in output for an identical response is valid and should be adopted by all black-box tools that wish to understand the web application's internal state machine.

Acknowledgements

This work was supported by the Office of Naval Research (ONR) under Grant N000141210165, the National Science Foundation (NSF) under grant CNS-1116967, and by Secure Business Austria.

References

- [1] ACUNETIX. Acunetix Web Vulnerability Scanner. <http://www.acunetix.com/>.
- [2] AMALFITANO, D., FASOLINO, A., AND TRAMONTANA, P. Reverse Engineering Finite State Machines from Rich Internet Applications. In *2008 15th Working Conference on Reverse Engineering* (2008), IEEE, pp. 69–73.
- [3] ARTZI, S., KIEZUN, A., DOLBY, J., TIP, F., DIG, D., PARADKAR, A., AND ERNST, M. Finding bugs in web applications using dynamic test generation and explicit state model checking. *IEEE Transactions on Software Engineering* (2010).
- [4] BALZAROTTI, D., COVA, M., FELMETSGER, V., JOVANOVIĆ, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *2008 IEEE Symposium on Security and Privacy* (2008), IEEE, pp. 387–401.
- [5] BALZAROTTI, D., COVA, M., FELMETSGER, V., AND VIGNA, G. Multi-module Vulnerability Analysis of Web-based Applications. In *Proceedings of the ACM conference on Computer and Communications Security (CCS)* (2007), pp. 25–35.
- [6] BAU, J., BURSSTEIN, E., GUPTA, D., AND MITCHELL, J. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 332–345.
- [7] BERG, T., JONSSON, B., AND RAFFELT, H. Regular Inference for State Machines using Domains with Equality Tests. In *Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering* (2008), Springer-Verlag, pp. 317–331.
- [8] BYRNE, D. Grendel-Scan. <http://www.grendel-scan.com/>.
- [9] CHINOTEC TECHNOLOGIES. Paros. <http://www.parosproxy.org/>.
- [10] CHONG, S., VIKRAM, K., AND MYERS, A. SIF: Enforcing confidentiality and integrity in web applications. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium* (2007), USENIX Association, p. 1.
- [11] COVA, M., BALZAROTTI, D., FELMETSGER, V., AND VIGNA, G. Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID 2007)* (2007), pp. 63–86.
- [12] CSALLNER, C., SMARAGDAKIS, Y., AND XIE, T. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 17, 2 (2008), 1–37.

- [13] DI LUCCA, G., FASOLINO, A., PACE, F., TRAMONTANA, P., AND DE CARLINI, U. WARE: a tool for the reverse engineering of Web applications. In *Sixth European Conference on Software Maintenance and Reengineering, 2002. Proceedings* (2002), pp. 241–250.
- [14] DIJKSTRA, E. W. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik 1* (1959), 269–271.
- [15] DOUPÉ, A., BOE, B., KRUEGEL, C., AND VIGNA, G. Fear the EAR: Discovering and Mitigating Execution After Redirect Vulnerabilities. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011)* (Chicago, IL, October 2011).
- [16] DOUPÉ, A., COVA, M., AND VIGNA, G. Why Johnny Can't Pentest: An Analysis of Black-box Web Vulnerability Scanners. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2010)* (2010), Springer, pp. 111–131.
- [17] FELMETSGER, V., CAVEDON, L., KRUEGEL, C., AND VIGNA, G. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *Proceedings of the USENIX Security Symposium* (Washington, DC, August 2010).
- [18] FOSSI, M. Symantec Global Internet Security Threat Report. Tech. rep., Symantec, April 2009. Volume XIV.
- [19] GARGOYLE SOFTWARE INC. HtmlUnit. <http://htmlunit.sourceforge.net/>.
- [20] GARRETT, J. J. Ajax: A New Approach to Web Applications. <http://www.adaptivepath.com/ideas/essays/archives/000385.php>, Feb. 2005.
- [21] GROSSMAN, J. Challenges of Automated Web Application Scanning. Blackhat Windows 2004, 2004.
- [22] HALFOND, W., CHOUDHARY, S., AND ORSO, A. Penetration testing with improved input vector identification. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on* (2009), IEEE, pp. 346–355.
- [23] HP. WebInspect. <https://download.hpsmartupdate.com/webinspect/>.
- [24] HUANG, Y.-W., HUANG, S.-K., LIN, T.-P., AND TSAI, C.-H. Web application security assessment by fault injection and behavior monitoring. In *Proceedings of the 12th international conference on World Wide Web* (New York, NY, USA, 2003), WWW '03, ACM, pp. 148–159.
- [25] HUANG, Y.-W., YU, F., HANG, C., TSAI, C.-H., LEE, D.-T., AND KUO, S.-Y. Securing web application code by static analysis and runtime protection. In *WWW '04: Proceedings of the 13th international conference on World Wide Web* (New York, NY, USA, 2004), ACM, pp. 40–52.
- [26] IBM. AppScan. <http://www-01.ibm.com/software/awdtools/appscan/>.
- [27] JENSEN, T. R., AND TOFT, B. *Graph Coloring Problems*. Wiley-Interscience Series on Discrete Mathematics and Optimization. Wiley, 1994.
- [28] JOVANOVIĆ, N., KRUEGEL, C., AND KIRDA, E. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security* 18, 5 (2010), 861–907.
- [29] KALS, S., KIRDA, E., KRUEGEL, C., AND JOVANOVIĆ, N. Secubat: a Web Vulnerability Scanner. In *Proceedings of the 15th international conference on World Wide Web* (2006), ACM, pp. 247–256.
- [30] LI, X., AND XUE, Y. BLOCK: A Black-box Approach for Detection of State Violation Attacks Towards Web Applications. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC 2011)* (Orlando, FL, December 2011).
- [31] LI, X., YAN, W., AND XUE, Y. SENTINEL: Securing Database from Logic Flaws in Web Applications. In *CODASPY* (2012), pp. 25–36.
- [32] MCALLISTER, S., KIRDA, E., AND KRUEGEL, C. Leveraging User Interactions for In-Depth Testing of Web Applications. In *Recent Advances in Intrusion Detection* (2008), Springer, pp. 191–210.
- [33] MESBAH, A., BOZDAG, E., AND VAN DEURSEN, A. Crawling AJAX by Inferring User Interface State Changes. In *Web Engineering, 2008. ICWE '08. Eighth International Conference on* (july 2008), pp. 122–134.
- [34] NURMI, D., WOLSKI, R., GRZEGORCZYK, C., OBERTELLI, G., SOMAN, S., YOUSEFF, L., AND ZAGORODNOV, D. The Eucalyptus Open-Source Cloud-Computing System. In *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on* (may 2009), pp. 124–131.
- [35] PORTSWIGGER. Burp Proxy. <http://www.portswigger.net/burp/>.
- [36] RAGHAVAN, S., AND GARCIA-MOLINA, H. Crawling the hidden web. In *Proceedings of the International Conference on Very Large Data Bases* (2001), Citeseer, pp. 129–138.
- [37] RIANCHO, A. w3af – Web Application Attack and Audit Framework. <http://w3af.sourceforge.net/>.
- [38] ROBERTSON, W., AND VIGNA, G. Static Enforcement of Web Application Integrity Through Strong Typing. In *Proceedings of the USENIX Security Symposium* (Montreal, Quebec CA, September 2009).
- [39] SCOTT, D., AND SHARP, R. Abstracting application-level web security. In *Proceedings of the 11th international conference on World Wide Web* (2002), ACM, pp. 396–407.
- [40] SPI DYNAMICS. Complete Web Application Security: Phase 1 – Building Web Application Security into Your Development Process. SPI Dynamics Whitepaper, 2002.
- [41] STEVE, C., AND MARTIN, R. Vulnerability Type Distributions in CVE. *Mitre report*, May (2007).
- [42] SUTO, L. Analyzing the Accuracy and Time Costs of Web Application Security Scanners, 2010.
- [43] VIEIRA, M., ANTUNES, N., AND MADEIRA, H. Using Web Security Scanners to Detect Vulnerabilities in Web Services. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on* (2009), IEEE, pp. 566–571.

A Web Applications

This section describes the web applications along with the functionality against which we ran the black-box web vulnerability scanner.

Gallery is an open-source photo hosting application. The administrator can upload photos and organize them into albums. Guests can then view and comment on the uploaded photos. Gallery has AJAX functionality but gracefully degrades (is fully functional) without JavaScript. No modifications were made to the application.

PhpBB v2 is an open-source forum software. It allows registered users to perform many actions such as create new threads, comment on threads, and message other users. Version 2 is notorious for the amount of security vulnerabilities it contains [6], and we included it for this reason. We modified it to remove the “recently online” section on pages, because this section is based on time.

PhpBB v3 is the latest version of the popular open-source forum software. It is a complete rewrite from Version 2, but retains much of the same functionality. Similar to PhpBB v2, we removed the “recently online” section, because it is time-based.

SCARF, the Stanford Conference And Research Forum, is an open-source conference management system. The administrator can upload papers, and registered users can comment on the uploaded papers. We included this application because it was used by previous research [5, 11, 30, 31]. No modifications were made to this application.

Vanilla Forums is an open-source forum software similar in functionality to PhpBB. Registered users can create new threads, comment on threads, bookmark interesting threads, and send a message to another user. Vanilla Forums is unique in our test set in that it uses the path to pass parameters in a URL, whereas all other applications pass parameters using the query part of the URL. For instance, a specific user’s profile is GET /profile/scanner1, while a discussion thread is located at GET /discussion/1/how-to-scan. Vanilla Forums also makes extensive use of AJAX, and does not gracefully degrade without JavaScript. For instance, with JavaScript disabled, posting a comment returns a JSON object that contains the success or failure of the comment posting, instead of an HTML response. We modified Vanilla Forums by setting an XSRF token that it used to a constant value.

WackoPicko v2 is an open-source intentionally vulnerable web application which was originally created to evaluate many black-box web vulnerability scanners [16]. A registered user can upload pictures, comment on other user’s pictures, and purchase another user’s picture. Ver-

sion 2 contains minor tweaks from the original paper, but no additional functionality.

WordPress v2 is an open-source blogging platform. An administrator can create blog posts, where guests can leave comments. No changes were made to this application.

WordPress v3 is an up-to-date version of the open-source blogging platform. Just like the previous version, administrators can create blog posts, while a guest can comment on blog posts. No changes were made to this application.

B Scanner Configuration

The following describes the exact settings that were used to run each of the evaluated scanners.

- wget is run in the following way:

```
wget -rp -w 0 --waitretry=0 -nd
--delete-after --execute robots=off
```
- w3af settings:

```
misc-settings
set maxThreads 0
back
plugins
discovery webSpider
audit blindSqli, eval,
localFileInclude, osCommanding,
remoteFileInclude, sqli, xss
```
- skipfish is run in the following way:

```
skipfish -u -LV -W /dev/null -m 10
```

Aurasium: Practical Policy Enforcement for Android Applications

Rubin Xu
Computer Laboratory
University of Cambridge
Cambridge, UK
Rubin.Xu@cl.cam.ac.uk

Hassen Saïdi
Computer Science Laboratory
SRI International
Menlo Park, USA
hassen.saidi@sri.com

Ross Anderson
Computer Laboratory
University of Cambridge
Cambridge, UK
Ross.Anderson@cl.cam.ac.uk

Abstract

The increasing popularity of Google's mobile platform Android makes it the prime target of the latest surge in mobile malware. Most research on enhancing the platform's security and privacy controls requires extensive modification to the operating system, which has significant usability issues and hinders efforts for widespread adoption. We develop a novel solution called Aurasium that bypasses the need to modify the Android OS while providing much of the security and privacy that users desire. We automatically repackage arbitrary applications to attach user-level sandboxing and policy enforcement code, which closely watches the application's behavior for security and privacy violations such as attempts to retrieve a user's sensitive information, send SMS covertly to premium numbers, or access malicious IP addresses. Aurasium can also detect and prevent cases of privilege escalation attacks. Experiments show that we can apply this solution to a large sample of benign and malicious applications with a near 100 percent success rate, without significant performance and space overhead. Aurasium has been tested on three versions of the Android OS, and is freely available.

1 Introduction

Google's Android OS is undoubtedly the fastest growing mobile operating system in the world. In July 2011, Nielsen placed the market share of Android in the U.S. at 38 percent of all active U.S. smartphones [9]. Weeks later, for the period ending in August, Nielsen found that Android has risen to 43 percent. More important, among those who bought their phones in June, July, or August, Google had a formidable 56 percent market share. This unprecedented growth in popularity, together with the openness of its application ecosystem, has attracted malicious entities to aggressively target Android. Attacks on Android by malware writers have jumped by 76 percent over the past three months according to a report by

MacAfee [29], making it the most assaulted mobile operating system during that period. While much of the initial wave of Android malware consisted of trojans that masquerade as legitimate applications and leak a user's personal information or send SMS messages to premium numbers, recent malware samples indicate an escalation in the capability and stealth of Android malware. In particular, attempts are made to gain root access on the device through escalation of privilege [37] to establish a stealthy permanent presence on the device or to bypass Android permission checks.

Fighting malware and securing Android-powered devices has focused on three major directions. The first one consists of statically [20] and dynamically [12, 36] analyzing application code to detect malicious activities before the application is loaded onto the user's device. The second consists of modifying the Android OS to insert monitoring modules at key interfaces to allow the interception of malicious activity as it occurs on the device [19, 27, 17, 33, 13]. The third approach consists of using virtualization to implement rigorous separation of domains ranging from lightweight isolation of applications on the device [35] to running multiple instances of Android on the same device through the use of a hypervisor [26, 30, 11].

Two fundamental and intertwined problems plague these approaches. The first is that the definition of malicious behavior in an Android application is hard to ascertain. Access to privacy- and security-relevant parts of Android's API is controlled by an install-time application permission system. Android users are informed about what data and resources an application will have access to, and user consent is required before the application can be installed. These explicit permissions are declared in the application package. Install-time permissions provide users with control over their privacy, but are often coarse-grained. A permission granted at install time is granted as long as the application is installed on the device. While an application might legitimately re-

quest access to the Internet, it is not clear what connections it may establish with remote servers that may be malicious. Similarly, an application might legitimately require sending SMS messages. Once the SMS permission is granted, there are no checks to prevent the application from sending SMS messages to premium numbers without user consent. In fact, the mere request for SMS permission by an application can be deemed malicious according to a recent Android applications analysis [24], where it is suggested that 82 percent of malicious applications require permissions to access SMS. A recent survey [18] exposes many of the problems [22, 14] associated with application components interactions, delegation of permission, and permission escalation attacks due to poor or missing security policy specifications by developers. This prompted early work [21] on security policy extension for Android.

The second problem is that any approach so far that attempts to enhance the platform's security and privacy controls based on policy extensions requires extensive modification to the operating system. This has significant usability issues and hinders any efforts for widespread adoption. There exists numerous tablet and phone models with different hardware configurations, each running a different Android OS version with its own customizations and device drivers. This phenomenon, also known as the infamous Android version *fragmentation problem* [16] demonstrates that it is difficult to provide a custom-built Android for all possible devices in the wild. And it is even more difficult to ask a normal user to apply the source patch of some security framework and compile the Android source tree for that user's own device. These issues will prevent many OS-based Android security projects from being widely adopted by the normal users. Alternatively, it is equally difficult to bring together Google, the phone manufacturers, and the cellular providers to introduce security extensions at the level of the consumer market, due to misaligned incentives from different parties.

Our Approach We aim at addressing these challenges by providing a novel, simple, effective, robust, and deployable technology called Aurasium. Conceptually, we want Aurasium to be an application-hardening service: a user obtains arbitrary Android applications from potentially untrusted places, but instead of installing the application as is, pushes the application through the Aurasium black box and gets a hardened version. The user then installs this hardened version on the phone, assured by Aurasium that all of the application's interactions are closely monitored for malicious activities, and policies protecting the user's privacy and security are actively enforced.

Aurasium does not need to modify the Android OS

at all; instead, it enforces flexible security and privacy policies to arbitrary applications by repackaging to attach sandboxing code to the application itself, which performs monitoring and policy enforcement. The repackaged application package (APK) can be installed on a user's phone and will enforce at runtime any defined policy without altering the original APK's functionalities. Aurasium exploits Android's unique application architecture of mixed Java and native code execution to achieve robust sandboxing. In particular, Aurasium introduces libc interposition code to the target application, wrapping around the Dalvik virtual machine (VM) under which the application's Java code runs. The target application is also modified such that the interposition hooks get placed each time the application starts.

Aurasium is able to interpose almost all types of interactions between the application and the OS, enabling much more fine-grained policy enforcement than Android's built-in permission system. For instance, whenever an application attempts to access a remote site on the Internet, the IP of the remote server is checked against an IP blacklist. Whenever an application attempts to send an SMS message, Aurasium checks whether the number is a premium number. Whenever an application tries to access private information such as the International Mobile Equipment Identity (IMEI), the International Mobile Subscriber Identity (IMSI), stored SMS messages, contact information, or services such as camera, voice recorder, or location, a policy check is performed to allow or disallow the access. Aurasium also monitors I/O operations such as write and read. We evaluated Aurasium against a large number of real-world Android applications and achieved over 99 percent success rate. Repackaging an arbitrary application using Aurasium is fast, requiring an average of 10 seconds.

Our main contributions are that

- We have built an automated system to repackage arbitrary APKs where arbitrary policies protecting privacy and ensuring security can be enforced.
- We have developed a set of policies that take advantage of advances in malware intelligence such as IP blacklisting.
- We provide a way of protecting users from malicious applications without making any changes to the underlying Android architecture. This makes Aurasium a technology that can be widely deployed.
- Aurasium is a robust technology that was tested on three versions of Android. It has low memory and runtime overhead and, unlike other approaches, is more portable across the different OS versions.

The paper is organized as follows: Section 2 provides the some background information on the architecture of Android and then goes through details about the architecture, enforceable policies and deployment methods of Aurasium. In Section 3 we evaluate Aurasium with respect to its robustness in repackaging applications, as well as the overhead introduced by the repackaging process. Section 4 describes threat models against Aurasium and mitigation techniques. Related work and conclusions are discussed in Section 5 and Section 6, respectively.

2 Aurasium

2.1 Android

Android, the open source mobile operating system developed by the Open Handset Alliance led by Google, is gaining increasing popularity and market share among smartphones. Built on top of a Linux 2.6 kernel, Android introduces a unique application architecture designed to ensure performance, security, and application portability. Rigorous compartmentalization of installed applications is enforced through traditional Linux permissions. Additional permission labels are assigned to applications during install time to control the application’s access to security and privacy-sensitive functionalities of the OS, forming a mandatory access-control scheme.

Android employs an inter-process communication (IPC) mechanism called Binder [6] extensively for interactions between applications as well as for application-OS interfaces. Binder is established by a kernel driver and exposed as a special device node on which individual applications operate. Logically, the IPC works on the principle of thread migration. A thread invoking an IPC call with Binder appears as if it migrates into the target process and executes the code there, hopping back when the result is available. All the hard work such as taking care of argument marshalling, tracking object references across processes, and recursions of IPC calls is handled by Binder itself.

Android applications are mainly implemented in Java, with the compiled class files further converted into Dalvik bytecode, running on the proprietary register-based Dalvik VM. It is similar to the JVM, but designed for a resource-constrained environment with a higher code density and smaller footprint. Applications are tightly coupled with a large and function-rich Android framework library (c.f. J2SE). Also, applications are free to include compiled native code as standalone Linux shared object (.so) files. The interaction between an application’s Java and native code is well defined by the Java Native Interface (JNI) specification and supported by Android’s Native Development Kit (NDK). In reality, the complexity of using native code means that only a

small number of applications employ native code for the most performance-critical tasks.

2.2 System Design

Aurasium is made up of two major components: the repackaging mechanism that inserts instrumentation code into arbitrary Android applications and the monitoring code that intercepts an application’s interactions with the system and enforces various security policies. The repackaging process makes use of existing open source tools augmented with our own glue logic to re-engineer Android applications. The monitoring code employs user-level sandboxing and interposition to intercept the application’s interaction with the OS. Aurasium is also able to reconstruct the high-level IPC communication from the low-level system call data, which allows it to monitor virtually all of Android’s APIs.

2.2.1 Application-OS Interaction

Under the hood, some of Android’s OS APIs are handled by the kernel directly, while others are implemented at user-mode system services and are callable via inter-process communication methods. However, in almost all scenarios the application does not need to distinguish between the two, as these APIs have already been fully encapsulated in the framework library and the applications just need to interact with the framework through well-documented interfaces. Figure 1 shows in detail the layers of the framework library in individual applications’ address space.

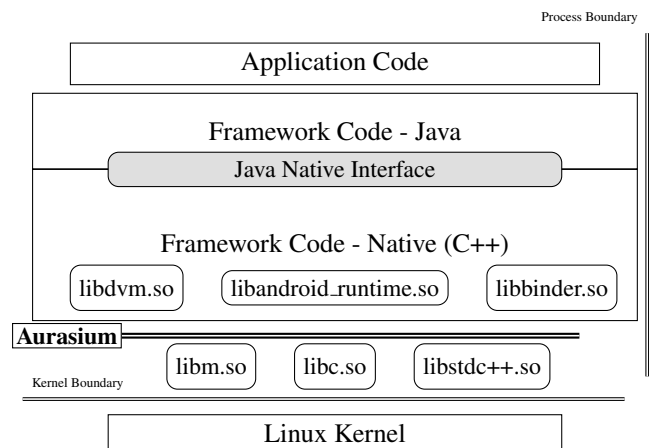


Figure 1: Android Application and Framework Structure

The top level of the framework is written in Java and is the well-documented part of the framework with which applications interact. This hides away the cumbersome

details from the application's point of view, but in order to realize the required operations it will hand over the request to the low-level part of the framework implemented in native code. The native layer of the framework consists of a few shared objects that do the real work, such as communicating with the Dalvik VM or establishing the mechanism for IPC communication. If we dive lower, we find that these shared objects are in fact also relying on shared libraries at even lower levels. There, we find Android's standard C libraries called *Bionic libc*. The Bionic libc will initiate appropriate system calls into the kernel that completes the required operation.

For example, if the application wants to download a file from the Internet, it has multiple ways to do so, ranging from fully managed `URLConnection` class to low-level `Socket` access. No matter what framework APIs the application decides to use, they will all land on the `connect()` method in the `OSNetworkSystem` Java class in order to create the underlying TCP socket. This `connect()` method in turn transfers control to `libnativehelper.so`, one of the shared objects in the native layer of the framework, which again delegates the request to the `connect()` method in `libc.so`. The socket is finally created by libc issuing a system call into the Linux kernel.

No matter how complex the upper layer framework library may be, it will always have to go through appropriate functions in the Bionic libc library in order to interact with the OS itself. This gives a reliable choke point at which the application's interactions with the OS can be examined and modified. The next section explains how function calls from the framework into libc can be interposed neatly.

2.2.2 Efficient Interposition

Similar to the traditional Linux model, shared objects in Android are relocatable ELF files that are mapped into the process's address space when loaded. To save memory and avoid code duplication, all shared objects shipped with Android are dynamically linked against the Bionic libc library. Because a shared object like libc can be loaded into arbitrary memory address, dynamic linking is used to resolve the address of unknown symbols at load time. For an ELF file that is dynamically linked to some shared object, its call sites to the shared object functions are actually jump instructions to some stub function in the ELF's procedure linkage table (PLT). This stub function then performs a memory load on some entry in the ELF's global offset table (GOT) in order to retrieve the real target address of this function call to which it then branches. In other words, the ELF's global offset table contains an array of function pointers of all dynamically linked external functions referenced by its code.

During dynamic linking this table is filled with appropriate function pointers; this is controlled by the metadata stored in the ELF file, such as which GOT entry maps to which function in which shared object.

This level of indirection introduced by dynamic linking can be exploited to implement the required interposition mechanism neatly: it is sufficient to go through every loaded ELF file and overwrite its GOT entries with pointers to our monitoring functions. This is equivalent to doing the dynamic linking again but substituting function pointers of interposition routines¹.

Because Java code is incapable of directly modifying process memory, we implemented our interposition routines in C++ and compiled them to native code. All the detour functions are also implemented in C++ and they will preprocess the relevant function call arguments before feeding them to Aurasium's policy logic. We try to minimize the amount of native code because it is generally difficult to write and test. As a result most of the policy logic is implemented in Java, which also means it can take advantage of many helper functions in the standard Android framework. However, in the preprocessing step of the IPC calls we make an effort to reconstruct the inter-process communication parameters as well as high-level Java objects out of marshalled byte streams in our native code. It turns out that despite the system changes between Android 2.2, 2.3 and 3.x, the IPC protocol remains largely unaffected² and hence our interposition code is able to run on all major Android versions reliably.

With all these facilities in place, Aurasium is capable of intercepting virtually all framework APIs and enforcing many classes of security and privacy policies on them. It remains to be discussed what policies we currently implement (Section 2.3) and how reliable Aurasium's sandboxing mechanism is (Section 4). But before that, let us explain how we repackage an Android application such that Aurasium's sandboxing code is inserted.

2.2.3 APK Repackaging

Android applications are distributed as a single file called an Android Application Package (APK) (Figure 2). An APK file is merely a Java JAR archive containing the compiled manifest file `AndroidManifest.xml`, the application's code in the form of dex bytecode, compiled XML resources such as window layout and string constant tables, and other resources like images, sound and native libraries. It also includes its own signature in a form identical to the standard Java JAR file signatures.

¹We did not consider other advanced dynamic linking techniques such as lazy linking here because they are not adopted in the current Android OS. They can be dealt with similarly.

²An exception is the introduction of `Strict Mode` from version

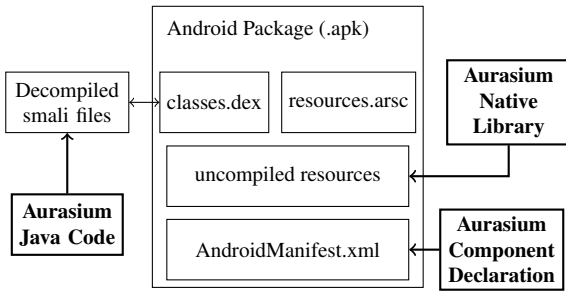


Figure 2: Android Application Package

Because the Aurasium code contains both a native library for low-level interposition and high-level Java code that executes the policy logic, we need a way of inserting both into the target APK. Adding a native library is trivial as native libraries are standalone Linux shared object (.so) files and are stored as is. Adding Java code is slightly tricky because Android requires all the application’s compiled bytecode to reside in a single file called `classes.dex`. To insert Aurasium’s Java code into an existing application, we have to take the original `classes.dex`, disassemble it back to a collection of individual classes, add Aurasium’s classes, and then re-assemble everything back to create the new `classes.dex`.

There exist open source projects that can perform such task. For example, `smali` [7], an assembler/disassembler for dex files, and `android-apktool` [1], which is an integrated solution that can process not only code but also compiled resources in APK files.³ In Aurasium we adopt `apktool` in our repackaging process. In the decode phase, `apktool` takes in an APK file, disassembles its dex file, and produces a directory such that each bytecode file maps to a single Java class, and its path corresponds to the package hierarchy, together with all other resources in the original APK file. Aurasium’s Java code is then merged into the directory and `apktool` is engaged again to assemble the bytecode back into a new `classes.dex` file. Together with other resources, a new APK file is finally produced.

In reality, before producing the final APK file there is one more thing to do: merely merging Aurasium code into the target application does not automatically imply that it will run. We need to make sure that Aurasium code is invoked somehow, preferably before any of the original application code, so that the application does not execute any of its code before Aurasium’s sandboxing is established. One option would be to modify the application’s entry point so that it points to Aurasium. This turns

out to be not as easy as one might expect. Android applications often possess many possible entry points, in the sense that every public application component including activity, service, broadcast receiver, and content provider can be invoked directly and hence they all act as entry points.

In Aurasium we take a different approach: The Android SDK allows an application to specify an `Application` class in its manifest file which will be instantiated by the runtime whenever the application is about to start. By declaring Aurasium as this `Application` class, Aurasium runs automatically before any other parts of the application. There is a small caveat that the original application may have already defined such `Application` class. In this case, we trace the inheritance of this class until we find the root base class. This class will have to be inherited directly from `Application`, and we modify its definition (which is in the decompiled bytecode form) such that it inherits from Aurasium’s `Application` class instead. This allows Aurasium to be instantiated as before, and being the root class ensures that Aurasium gets run before the application’s `Application` class is instantiated.

Figure 2 illustrates the composition of an APK and the various Aurasium modules added at repackaging time.

2.2.4 Application Signing

The last thing to worry about is that when an application is modified and repackaged, its signature is inevitably destroyed and there is no way to re-sign the application under its original public key. We believe this is a problem, but manageable. Every Android application is indeed required to have a valid signature, but signatures in Android work more like a proof of authorship, in the sense that applications signed by the same certificate are believed to come from the same author, hence they are trusted by each other and enjoy certain flexibilities within Android’s security architecture, e.g., signature permission. Application updates are also required to be signed with the same certificate as the original application. Other than that, signatures impose few other restrictions, and developers often use self-signed certificates for their applications.

This observation means that Aurasium can just re-sign the repackaged application using a new self-signed certificate. To preserve the authorship relation, Aurasium performs the re-signing step using a parallel set of randomly generated Aurasium certificates, maintaining a one-to-one mapping between this set to arbitrary developer certificates. In other words, whenever Aurasium is about to re-sign an application, it first verifies the validity of the old signature. If it passes, then Aurasium will proceed to sign the application with its own

2.3 Gingerbread.

³`apktool` is actually built on top of a fork of `smali`.

certificate that corresponds to the application's original certificate, or a newly generated one if this application has not been encountered earlier. In this way, the equivalence classes of authorship among applications are still maintained by Aurasium's re-signing procedure. Problems can still arise if Aurasium re-signs only a partial set of applications in the cases of application updates or applications intending to cooperate with their siblings. We consider these cases non-severe, with one reason being that Aurasium is more likely to be applied to a standalone application from a non-trusted source where application updates and application cooperation are not common.

Because all private keys of the generated certificates need to be stored⁴ for future queries, the re-signing process contains highly confidential information and, hence, requires careful protection. It should be (physically) separated from Aurasium's other services and perceived as an oracle with minimal interfaces to allow re-signing an already-signed application. For higher assurance, hardware security modules could be used.

2.2.5 Aurasium's Security Manager

Aurasium-wrapped applications are self-contained in the sense that the policy logic and the relevant user interface are included in the repackaged application bundle, and so are remembered user decisions stored locally in the application's data directory. Alternatively, Aurasium Security Manager (ASM) can also be installed, enabling central handling of policy decisions of all repackaged application on the device. Depending on the enforced policies at repackaging time, an application queries the ASM for a policy decision via IPC mechanisms with intents describing the sensitive operation it is about to perform, and the ASM either prompts the user for consent, uses a remembered user decision recorded earlier, or automatically makes a decision without user interaction by enforcing a predefined policy embedded at repackaging time. The policy logic in individual applications prefers delegating policy decisions to the ASM, and will fall back to local decisions only if a genuine ASM instance is not detected on the device.

Using ASM for central policy decision management has one major advantage: policy logic can be controlled globally, and it can also be improved by updating the ASM instance on the device. For example, IP address blacklisting and whitelisting can be managed and kept up to date by ASM. Repackaged applications are able to take advantage of better policy logics once ASM is updated, even after they have been repackaged and deployed to users' devices. There is a tradeoff between the flexibility of ASM and the efficiency of repackaged ap-

⁴Alternatively, these new certificates can be generated from the original certificate under a master key.

plication, though. In extreme cases, a repackaged application can proxy every IPC call to ASM, but this would be vastly inefficient. In our implementation ASM is consulted only with high-level summaries of potential sensitive operations, the set of which is fixed at repackaging time.

2.3 Policies

Now that we have demonstrated the ability to repackage arbitrary applications with Aurasium to insert monitoring code, we discuss various security policies that leverage this technique. It is important to point out that these are just some examples that we implemented as a proof of concept so far. Aurasium itself provides a flexible framework under which many more potent policies are possible.

We are interested primarily in enforcing some security policy that protects the device from untrusted applications. This includes not only attempts by the application to access sensitive information, leaking to the outside world or modifying it, but also attempts by the application to escalate privilege and to gain root access on the device by running suspicious system calls and loading native libraries. Aurasium's architecture and design allow us to implement many of the already-proposed policies such as dynamically constraining permissions [33], or setting up default dummy IMEI and IMSI numbers as well as phone numbers, as in [27].

The following subsections describe a set of policies that are easily checkable by Aurasium. The enforcement of these policies is supported by Aurasium intercepting the following functions:

- `ioctl()`

This is the main API through which all IPCs are sent. By interposing this and reconstructing the high-level IPC communication, Aurasium is able to monitor most Android system APIs and enforce the privacy and SMS policies, and modifying the IPC arguments and results on the fly. In certain cases such as content providers, Aurasium replaces the returned `Cursor` object with a wrapper class to allow finer control over the returned data.

- `getaddrinfo()` and `connect()`

These functions are responsible for DNS resolving and socket connection. Intercepting them allows Aurasium to control the application's Internet access.

- `dlopen()`, `fork()` and `execvp()`

The loading of native code from Java and execution of external programs are provided by these functions, which Aurasium monitors.

- `read()`, `write()`

These functions reflect access to the file system. Intercepting them allows Aurasium to control private and shared files accesses.

- `open()` and reflection APIs in `libdvm.so`⁵

These functions are intercepted to prevent malicious applications from circumventing Aurasium’s sandboxing. Because Aurasium may stores policy decisions in the application’s local directory, it must prevent the application from tampering with the decision file. `open()` is hooked such that whenever it is invoked on the decision file it will check the JNI call stack and allow only Aurasium code to successfully open the file. The various reflection APIs are also guarded to prevent malicious applications from modifying Aurasium’s Java-based policy logic by reflection.

2.3.1 Privacy Policy

The most obvious set of policies that can be defined relates to users’ privacy. These policies protect the private data of the user such as the IMEI, IMSI, phone number, location, stored SMS messages, phone conversations, and contact list. These policies can be checked by monitoring access to the system services provided by the Framework APIs. While many APIs are available to access system services, they all translate to a single call to the `ioctl()` system call. By monitoring calls to `ioctl()`, and parsing the data that is transmitted in the call, we are able to determine which service is being accessed and alert the user.

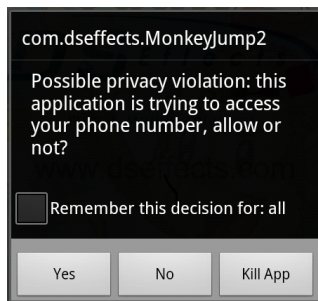


Figure 3: Enforcement of Privacy Policies: Access to Phone Number

Figure 3 illustrates how Aurasium intercepts a request made by an application to access the user’s phone number. Aurasium displays a warning message and prompts

⁵`Dalvik_java_lang_reflect_Method_invokeNative()`, `Dalvik_java_lang_reflect_Field_setField()` and `Dalvik_java_lang_reflect_Field_setPrimitiveField()`

the user to either accept the requested access or deny it. The user can also make Aurasium store that user’s answer to the request so that the same request never prompts the user for approval again and the cached answer is used instead. Finally, the user has the option to terminate the application.

Aurasium is capable of intercepting requests for the IMEI (Figure 4) and IMSI identifiers. Both the IMEI and IMSI numbers are often abused by applications to track users and devices for analytics and advertisement purposes, but are also used by malware to identify victims.

Similar policies are also implemented for accessing device location and contact list. In all of the above cases, if the user denies an request for the private information, Aurasium will provide shadow data to the application instead, similar to the approach in [27];

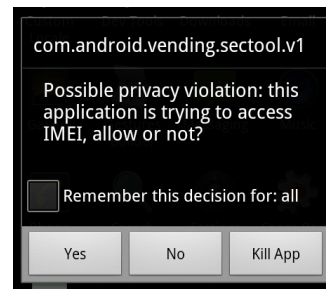


Figure 4: Enforcement of Privacy Policies: Access to IMEI from repackaged Android Market Security Tool malware.

2.3.2 Preventing SMS Abuse

Figure 5 illustrates how Aurasium intercepts SMS messages sent to a premium number, which is initiated by the malicious application `AndroidOS.FakePlayer` [2] found in the wild. Aurasium displays the destination number as well as the SMS’s content, so users can make informed decision on whether to allow the operation or not. In this case, the malware is most likely to attempt to subscribe to some premium service covertly. We also observed malware `NickySpy` [3] leaking device IMEI via SMS in another test run. We believe automatic classification on SMS number and content is possible to further reduce user intervention.

2.3.3 Network Policy

Similarly to the privacy policies, we enforce a set of network policies that regulate how an application is allowed to interact with the network. Since the Android permission scheme allows unrestricted access to the Internet when an application is installed, we enforce finer-grained

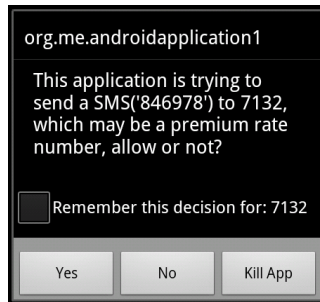


Figure 5: Enforcement of SMS Sending

policies that are expressed as a combination of the following:

- restrict the application to only a particular web domain or set of IP addresses
- restrict the application from connecting to a remote IP address known to be malicious

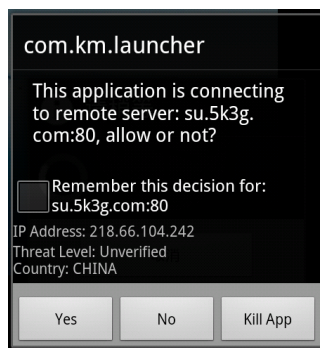


Figure 6: Enforcement of Network Policies: Access to an IP address with an unverified level of maliciousness

We use an IP blacklisting provided by the Bothunter network monitoring tool [4] to harvest information about malicious IP addresses. For each connection, the service retrieves information about the remote location, and the warning presented to the user indicates the level of maliciousness of the remote location (Figure 6). We also display the geo-location of the remote IP. It would be possible to include more threat intelligence from various diverse sources.

2.3.4 Privilege Escalation Policy

In addition to the privacy policy and the network policy, we implement a policy that warns the user when a suspicious `execvp` is invoked. Aurasium intervenes whenever the application tries to execute external ELF binaries.

Knowing the attack signatures based on suspicious executables can prevent certain types of escalation of privilege attacks. Figure 7 illustrates an interception of the `su` command. The Aurasium warning indicates that the application is trying to gain root access on a potentially rooted phone by executing the `su` command.

In another scenario, Aurasium warns the user when the application is about to load a native library. Malicious native code can interfere with Aurasium and potentially break out of its sandbox, which we discuss further in section 4.



Figure 7: Enforcement of Privilege Escalation Policy

2.3.5 Automatic Embedding of policies

Our implementation allows us to naturally compare the behavior of an application against a policy expressed not as a single event such as a single access to private data, to a system service or a single invocation of a system call, but as a sequence of such events. We plan on automatically embedding into an application code an arbitrary user-defined policy expressible in an automaton.

2.4 Deployment Models

Driven by the need for deployable mobile security solutions for Android and other platforms, we support multiple deployment models for Aurasium. The unrestricted and open nature of the Android Market allows us to provide Aurasium hardened and repackaged applications to users directly. Here, we discuss several deployment models for Aurasium that users can directly use without modifying the Android OS on their phones.

2.4.1 Web Interface

We have a web interface⁶ that allows users to upload arbitrary applications and download the Aurasium repackaged and hardened version. Aurasium can be employed to repackage any APKs that the user possesses.

⁶www.aurasium.com

2.4.2 Cooperation with Application Markets

We are exploring collaborations with Android markets run by mobile service providers to deploy Aurasium. Subscribers to the mobile service who get their applications from the official Android market supported by the mobile provider will have all their applications packaged with Aurasium for protection.

2.4.3 Deployment in the Cloud

Another deployment model consists of writing a custom download application that runs on a user's phone so that whenever a user browses an Android market and wishes to download an application, the application is pulled and sent to the Aurasium cloud service where the application is repackaged and then downloaded to the user's phone. This may be more accessible as users no longer need to interact with Aurasium's web interface manually.

2.4.4 Phone Deployment

Similarly to the cloud service, we plan on porting the repackaging tool to the Android phone itself. That is, we will be able to repackage an application on the device itself.

2.4.5 Corporate Environment

Many corporations have security concerns about mobile devices in their infrastructure and Aurasium can help to establish the desired security and privacy policies on applications to be installed on these devices. These Android devices should be configured to allow installing only Aurasium-protected applications (by means of APK signatures for example), while the applications can be provided by some methods described above, such as an internal repackaging service or a transparent repackaging proxy between the application market and the device.

3 Evaluation

We have evaluated Aurasium on a collection of Android applications to ensure that the application repackaging succeeds and that our added code does not impede the original functionality of the application. We have conducted a broad evaluation that includes a large number of benign applications as well as malware collection. Our evaluation was conducted on a Samsung Nexus S phone running Android 2.3.6 "Gingerbread".

3.1 Setting Up An Evaluation Framework

Aurasium consists of scripts that implement the repackaging process described in Figure 2. It transforms each

APK file in the corpus to the corresponding hardened repackaged application. We scripted to load the application onto the Nexus S phone, start the application automatically, and capture the logs generated by Aurasium. Android Monkey [8] is used to randomly exercise the user interface (UI) of the application.

Monkey is a program running on Android that feeds the application with pseudo-random streams of user events such as clicks and touches, as well as a number of system-level events. We use Monkey to stress-test the repackaged applications in a random yet repeatable manner. The captured logs allow us to determine whether the application has started and is being executed normally or whether it crashes due to our repackaging process. As a random fuzzer, Monkey is fundamentally unable to exercise all execution paths of an application. But in our setup, running random testing over a large number of independent applications proves useful, covering most of Aurasium's policy logic and revealing several bugs.

3.2 Repackaging Evaluation

We first performed an evaluation to determine how many APK files can successfully be repackaged by Aurasium. Table 1 shows a breakdown of the Android APK files corpus on which we ran our evaluation. We applied Aurasium to 3491 applications crawled from a third-party application store⁷ and 1260 known malicious applications [39]. Table 1 shows the success rate of repackaging for each category of applications.

Type of App	#of Apps	Repackaging Success Rate
App store corpus	3491	99.6%(3476)
Malware corpus	1260	99.8%(1258)

Table 1: Repackaging Evaluation Results

We have a near 100% success rate in repackaging arbitrary applications. Our failures to repackage an application are due to bugs in `apktool` in disassembling and reassembling the hardened APK file. We are working on improving `apktool` to achieve a 100% success rate.

3.3 Runtime Robustness

As we pointed out earlier, Aurasium is able to run on all major Android versions (2.2, 2.3, 3.x) without any problem. We performed the robustness evaluation on a Samsung Nexus S phone running Android 2.3.6 (which is among the most widely used Android distributions

⁷<http://lisvid.com>

2.3.3 – 2.3.7 [10]). For each hardened application we use Monkey to exercise the application’s functionalities by injecting 500 random UI events. These hardened applications are built with a debug version of Aurasium that will output a log message when Aurasium successfully intercepts an API invocation. Out of 3476 successfully repackaged application, we performed tests on 3189 standalone runnable applications⁸ on the device. We were able to start all of the applications in the sense that Aurasium successfully reported the interception of the first API invocation for all of them.

3.4 Performance Evaluation

We take two Android benchmark applications from the official market and apply Aurasium to them in order to check if Aurasium introduces significant performance overhead to a real-world application. In both cases, the benchmark scores turn out to be largely unaffected by Aurasium (Table 2).

Benchmark App	without Aurasium	with Aurasium
AnTuTu Benchmark	2900 <i>Pts</i>	2892 <i>Pts</i>
BenchmarkPi	1280 <i>ms</i>	1293 <i>ms</i>

Table 2: Performance on Benchmark Applications

Aurasium introduces the most overhead when the application performs API invocations, which is not the most important test factor of these benchmarks. So we synthesized an artificial application that performs a large number of API invocations, in order to find Aurasium’s performance overhead in the worst cases. Because these APIs all involve IPC with remote system services, they are expected to induce the most overhead as Aurasium needs to fully parse the Binder communication. Results in Table 3 show that Aurasium introduces an overhead of 14% to 35% in three cases, which we believe is acceptable as IPC-based APIs are not frequently used by normal applications to become the performance bottleneck. In objective testing we did not feel any lagging when playing with an Aurasium-hardened application.

3.5 Size Overhead

We evaluated application size after being repackaged with Aurasium code, as shown in Figure 8. On average, Aurasium increases the application size by only 52

⁸The rest are applications that do not have a main launchable Activity, and applications that fail to install due to clashes with pre-installed version.

200 API Invocations	Without Aurasium	With Aurasium	Overhead
Get Device Info	106 <i>ms</i>	143 <i>ms</i>	35%
Get Last Location	41 <i>ms</i>	55 <i>ms</i>	34%
Query Contact List	1270 <i>ms</i>	1340 <i>ms</i>	14%

Table 3: Performance on Synthesized Application

Kb, which is a very small overhead for the majority of applications.

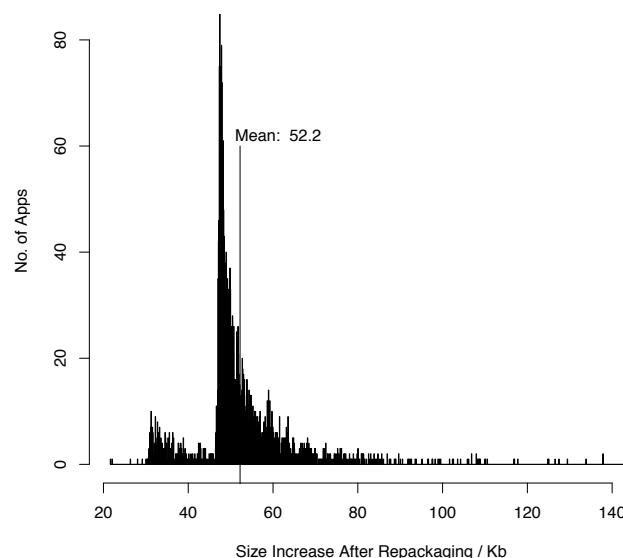


Figure 8: Application Size Increase After Repackaging.

3.6 Policies Enforcement

We observe the various behaviors intercepted from the 3031 runnable applications that were previously repackaged and run on the Nexus S device under Monkey. Table 4 shows a breakdown of the application corpus into permission requested in the manifest file of the applications. It also shows which applications actually make use of the permission to access the requested service.

Permission	Requested	Accessed
Internet Permission	2686	1305
GPS Permission	846	132
Phone State Permission	1243	378

Table 4: Permission Requested and Permissions Used

Due to the random fuzzing nature of our evaluation,

the accessed permission is most likely to be an underestimate. We also observed that 226 applications included native code libraries in their application bundle.

4 Attack Surfaces

Because fundamentally Aurasium code runs in the same process context as the application's code, there is no strong barrier between the application and Aurasium. Hence, it is non-trivial to argue that Aurasium can reliably sandbox arbitrary Android applications. We describe possible ways that a malicious application can break out of Aurasium's policy enforcement mechanism and discuss possible mitigation against them.

4.1 Native Code

Aurasium relies on being able to intercept calls to Bionic libc functions by means of rewriting function pointers in a module's global offset table. This is robust against arbitrary Java code, but a malicious application can employ native code to bypass Aurasium completely either by restoring the global offset table entries, by making relevant system calls using its own libc implementation rather than going through the monitored libc, or by tampering with the code or private data of Aurasium. However, because Android runtime requires applications to bootstrap as Java classes, the first load of native code in even malicious applications has to go through a well-defined and fixed pathway as defined by JNI. This gives us an upper hand in dealing with potential untrusted native code: because of the way our repackaging process works, Aurasium is guaranteed to start before the application's code and hence be able to intercept the application's first attempt to load alien native code (invocation of `dlopen()` function in libc). As a result, Aurasium is guaranteed to detect any potential circumvention attempts by a malicious application.

What can Aurasium do with such an attempt? Silently denying the load of all native code is not satisfactory because it will guarantee an application crash and some legitimate applications use native code. Even though Aurasium has the power to switch off the unknown native code, the collateral damage caused by false positives would be too severe.

If Aurasium is to give binary decisions on whether or not to load some unknown native code, then it reduces to the arms race between malware and antivirus software that we have seen for years. Aurasium tries to classify native code in Android applications, while malware authors craft and obfuscate it to avoid being detected. It is better not to go down the same road; and a much neater approach would be letting the native code run, but not with unlimited power.

Previous work [28, 40, 34] on securely executing untrusted native code provides useful directions for example, by using dynamic binary translation. In our scenario we are required to restrict the application's native code from writing to guarded memory locations (to prevent tampering with Aurasium and the libc interposition mechanism), using special machine instructions (to initiate system calls without going through libc), and performing arbitrary control flow transfer into libc. Due to time constraints we have not implemented such facilities in Aurasium. Currently, Aurasium prompts the user for a decision, and informs the user that if the load is allowed then Aurasium can be rendered ineffective from this point onwards. We consider this problem a high priority for future work.

Unlike the filtering-based hybrid sandboxes that are prone to the 'time of check/time of use' race conditions [25, 38], Aurasium's sandboxing mechanism is delegation based and hence much easier to defend against this class of attack.

4.2 Java Code

A possible attack on Aurasium would be using Java's reflection mechanism to interfere with the operation of Aurasium. Because currently Aurasium's policy enforcement logic is implemented in Java, a malicious application can use reflection to modify Aurasium's internal data structures and hence affect its correct behavior. We prevent such attacks by hooking into the reflection APIs in `libdvm.so` and preventing reflection access to Aurasium's internal classes.

Note that dynamically loaded Java code (via `DexClassLoader`) poses no threat to Aurasium, as the code is still executed by the same Dalvik VM instance and hence cannot escape Aurasium's sandbox. Native Java methods map to a dynamically loaded binary shared object library and are subject to the constraints discussed in the previous section, which basically means that attempts of using them will always be properly flagged by Aurasium.

4.3 Red Pill

Currently Aurasium is not designed to be stealthy. The existence of obvious traces such as changed application signature, the existence of Aurasium native library and Java classes allow applications to find out easily whether it is running under Aurasium or not. A malicious application can then refuse to run under Aurasium, forcing the user to use the more dangerous vanilla version. A legitimate application may also verify its own integrity (via application signature) to prevent malicious repackaging by malware writers. Due to Aurasium's control over the application's execution, it is possible to clean

up these traces for example by spoofing signature access to `PackageManager`, but fundamentally this is an arms race and a determined adversary will win.

5 Related Work

With the growing popularity of Android and the growing malware threat it is facing, many approaches to securing Android have been proposed recently. Many of the traditional security approaches adopted in desktops have been migrated to mobile phones in general and Android in particular. Probably the most standard approach is to use signature-based malware detection, which is in its infancy when it comes to mobile platforms. This approach is ineffective against zero-day attacks, and there is little reason to believe that it will be more successful in the mobile setting. Program analysis and behavioral analysis have been more successfully applied in the context of Android.

Static Analysis Static analysis of Android application package files is relatively more straightforward than static analysis of malware prevalent on desktops in general. Obfuscation techniques [41] used in today's malware are primarily aimed at impeding static analysis. Without effective ways to deobfuscate native binaries, static analysis will always suffer major drawbacks. Because of the prevalence of malware on x86 Windows machines, little effort has been focusing on reverse engineering ARM binaries. Static analysis of Java code is much more attainable through decompilation of the Dalvik bytecode. The DED [20] and dex2jar [5] are two decompilers that aim at achieving translation from Dalvik bytecode to Java bytecode.

Dynamic Analysis Despite its limitations, dynamic analysis remains the preferred approach among researchers and antivirus companies to profile malware and extract its distinctive features. The lack of automated ways to explore all the state space is often a hindering factor. Techniques such as multipath exploration [31] can be useful. However, the ability of mobile malware to load arbitrary libraries might limit the effectiveness of such techniques. The honeynet project offers a virtual machine for profiling Android Applications [36] similar to profiling desktop malware. Stowaway [23] is a tool that detects overprivilege in compiled Android applications. Testing is used on the Android API in order to build the permission map that is necessary for detecting overprivilege, and static analysis is used to determine which calls an application invokes.

Monitoring The bulk of research related to securing Android has been focused on security policy extension and enforcement for Android starting with [21]. TaintDroid [19] taints private data to detect leakage of users' private information modifying both Binder and the Dalvik VM, but extends only partially to native code. Quire [17] uses provenance to track permissions across application boundaries through the IPC call chain to prevent permission escalation of privilege attacks. Crepe [15] allows access to system services requested through install-time permission only in a certain context at runtime. Similarly, Apex [33] uses user-defined runtime constraints to regulate applications' access to system services. AppFence [27] blocks application access to data from imperious applications that demand information that is unnecessary to perform their advertised functionality, and covertly substitute shadow data in place. Airmid [32] uses cooperation between in-network sensors and smart devices to identify the provenance of malicious traffic.

Virtualization Recent approaches to Android security have focused on bringing virtualization technology to Android devices. The ability to run multiple version of the Android OS on the same physical device allows for strong separation and isolation but comes at a higher performance cost. L4Android [30] is an open source project derived from the L4Linux project. L4Android combines both the L4Linux and Google modifications of the Linux kernel and thus enables running Android on top of a microkernel. To address the performance issues when using virtualization, Cells in [11], is a lightweight virtualization architecture where multiple phones run on the same device. It is possible to run multiple versions of Android on a bare metal hypervisor and ensure strong isolation where shared security-critical device drivers run in individual virtual machines, which is demonstrated by [26]. Finally, logical domain separation, where two single domains are considered and isolation is enforced as a dataflow property between the logical domains without running each domain as a separate virtual machine, can also be employed [35].

6 Conclusion and Future Work

We have presented Aurasium, a robust and effective technology that protects users of the widely used Android OS from malicious and untrusted applications. Unlike many of the security solutions proposed so far, Aurasium does not require rooting and device reflashing.

Aurasium allows us to take full control of the execution of an application. This allows us to enforce arbitrary policies at runtime. By using the Aurasium security

manager (ASM), we are able to not only apply policies at the individual application level but across multiple applications simultaneously. This allows us to effectively orchestrate the execution of various applications on the device and mediate their access to critical resources and user's private data. This allows us to also detect attempts by multiple applications to collaborate and implement a malicious logic. With its overall low overhead and high repackaging success rate, it is possible to imagine Aurasium implementing an effective isolation and separation at the application layer without the need of complex virtualization technology.

Even though Aurasium currently only treats applications as black boxes and focuses on its external behavior, the idea of enforcing policy at per-application level by repackaging applications to attach side-by-side monitoring code is very powerful. By carefully instrumenting the application's Dalvik VM instance on the fly, it is even possible to apply more advanced dynamic analysis such as information flow and taint analysis, and we leave this as a possible direction of future work. We also plan on expanding our investigation of the potential threat models against Aurasium and provide practical ways to mitigate them, especially in the case of executing untrusted native code.

7 Acknowledgments

This material is based on work supported by the Army Research Office under Cyber-TA Grant No. W911NF-06-1-0316 and by the National Science Foundation Grant No. CNS-0716612.

References

- [1] Android apktool: A tool for reengineering Android apk files. code.google.com/p/android-apktool/.
- [2] Android.OS/Fakeplayer. www.f-secure.com/v-descs/trojan_androidos_fakeplayer_a.shtml.
- [3] Android.OS/NickiSpy. www.maikmorgenstern.de/wordpress/?tag=androidnickispy.
- [4] Bothhunter community threat intelligence feed. <http://www.bothhunter.net>.
- [5] dex2jar: A tool for converting Android's .dex format to Java's .class format. code.google.com/p/dex2jar/.
- [6] OpenBinder. www.angryredplanet.com/~hackbod/openbinder/docs/html/.
- [7] smali: An assembler/disassembler for Android's dex format. code.google.com/p/smali/.
- [8] UI/Application exerciser Monkey. developer.android.com/guide/developing/tools/monkey.html.
- [9] In U.S. market, new smartphone buyers increasingly embracing Android. blog.nielsen.com/nielsenwire/online_mobile/, sep 2011.
- [10] ANDROID OPEN SOURCE PROJECT. Platform versions. developer.android.com/resources/dashboard/platform-versions.html.
- [11] ANDRUS, J., DALL, C., HOF, A. V., LAADAN, O., AND NIEH, J. Cells: a virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 173–187.
- [12] BLÄSING, T., SCHMIDT, A.-D., BATYUK, L., CAMTEPE, S. A., AND ALBAYRAK, S. An Android application sandbox system for suspicious software detection. In *5th International Conference on Malicious and Unwanted Software (MALWARE'2010)* (Nancy, France, France, 2010).
- [13] BURGUERA, I., ZURUTUZA, U., AND NADJIM-TEHRANI, S. Crowdroid: behavior-based malware detection system for Android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile devices* (New York, NY, USA, 2011), SPSM '11, ACM, pp. 15–26.
- [14] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2011), MobiSys '11, ACM, pp. 239–252.
- [15] CONTI, M., NGUYEN, V. T. N., AND CRISPO, B. Crepe: context-related policy enforcement for Android. In *Proceedings of the 13th International Conference on Information Security* (Berlin, Heidelberg, 2011), ISC'10, Springer-Verlag, pp. 331–345.
- [16] DEGUSTA, M. Android orphans: Visualizing a sad history of support. theunderstatement.com/post/11982112928/android-orphans-visualizing-a-sad-history-of-support.
- [17] DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WAL-LACH, D. S. Quire: lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX Conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 23–23.
- [18] ENCK, W. Defending users against smartphone apps: Techniques and future directions. In *Proceedings of the 7th International Conference on Information Systems Security* (Kolkata, India, dec 2011), ICISS.
- [19] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–6.
- [20] ENCK, W., OCTEAU, D., MCDANIEL, P., AND CHAUDHURI, S. A study of Android application security. In *Proceedings of the 20th USENIX conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 21–21.
- [21] ENCK, W., ONGTANG, M., AND MCDANIEL, P. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 235–245.
- [22] ENCK, W., ONGTANG, M., AND MCDANIEL, P. Understanding Android security. *IEEE Security and Privacy* 7 (January 2009), 50–57.
- [23] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 627–638.

- [24] FELT, A. P., FINIFTER, M., CHIN, E., HANNA, S., AND WAGNER, D. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (Oct. 2011), SPSM '11, ACM, pp. 3–14.
- [25] GARFINKEL, T., PFAFF, B., AND ROSENBLUM, M. Ostia: A delegating architecture for secure system call interposition. In *Proceedings of the Network and Distributed Systems Security Symposium* (February 2004).
- [26] GUDETH, K., PIRRETTI, M., HOEPER, K., AND BUSKEY, R. Delivering secure applications on commercial mobile devices: the case for bare metal hypervisors. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile devices* (New York, NY, USA, 2011), SPSM '11, ACM, pp. 33–38.
- [27] HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 639–652.
- [28] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. P. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium* (Berkeley, CA, USA, 2002), USENIX Association, pp. 191–206.
- [29] LABS, M. McAfee threats report: Second quarter 2011. www.mcafee.com/us/resources/reports/rp-quarterly-threat-q2-2011.pdf, aug 2011.
- [30] LANGE, M., LIEBERGELD, S., LACKORZYNSKI, A., WARG, A., AND PETER, M. L4Android: a generic operating system framework for secure smartphones. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile devices* (New York, NY, USA, 2011), SPSM '11, ACM, pp. 39–50.
- [31] MOSER, A., KRUEGEL, C., AND KIRDA, E. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2007), SP '07, IEEE Computer Society, pp. 231–245.
- [32] NADJI, Y., GIFFIN, J., AND TRAYNOR, P. Automated remote repair for mobile malware. In *Proceedings of the 2011 Annual Computer Security Applications Conference* (Washington, DC, USA, 2011), ACSAC '10, ACM.
- [33] NAUMAN, M., KHAN, S., AND ZHANG, X. Apex: extending Android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2010), ASIACCS '10, ACM, pp. 328–332.
- [34] SCOTT, K., KUMAR, N., VELUSAMY, S., CHILDERS, B., DAVIDSON, J. W., AND SOFFA, M. L. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the international symposium on Code Generation and Optimization: feedback-directed and runtime optimization* (Washington, DC, USA, 2003), CGO '03, IEEE Computer Society, pp. 36–47.
- [35] SVEN, B., LUCAS, D., ALEXANDRA, D., STEPHAN, H., AHMAD-REZA, S., AND BHARGAVA, S. Practical and lightweight domain isolation on Android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile devices* (New York, NY, USA, 2011), SPSM '11, ACM, pp. 51–62.
- [36] THE HONEYNET PROJECT. Android reverse engineering virtual machine. www.honeynet.org/node/783.
- [37] VIDAS, T., VOTIPKA, D., AND CHRISTIN, N. All your droid are belong to us: a survey of current Android attacks. In *Proceedings of the 5th USENIX Workshop On Offensive Technologies* (Berkeley, CA, USA, 2011), WOOT'11, USENIX Association, pp. 10–10.
- [38] WATSON, R. N. M. Exploiting concurrency vulnerabilities in system call wrappers. In *Proceedings of the first USENIX Workshop On Offensive Technologies* (Berkeley, CA, USA, 2007), USENIX Association, pp. 2:1–2:8.
- [39] YAJIN, Z., AND XUXIAN, J. Dissecting android malware: Characterization and evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy* (may 2012).
- [40] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: a sandbox for portable, untrusted x86 native code. *Commun. ACM* 53 (January 2010), 91–99.
- [41] YOU, I., AND YIM, K. Malware obfuscation techniques: A brief survey. In *Proceedings of the 2010 International Conference on Broadband, Wireless Computing, Communication and Applications* (Washington, DC, USA, 2010), BWCCA '10, IEEE Computer Society, pp. 297–300.

AdSplit: Separating smartphone advertising from applications

Shashi Shekhar
shashi.shekhar@rice.edu

Michael Dietz
mdietz@rice.edu

Dan S. Wallach
dwallach@rice.edu

Abstract

A wide variety of smartphone applications today rely on third-party advertising services, which provide libraries that are linked into the hosting application. This situation is undesirable for both the application author and the advertiser. Advertising libraries require their own permissions, resulting in additional permission requests to users. Likewise, a malicious application could simulate the behavior of the advertising library, forging the user's interaction and stealing money from the advertiser. This paper describes AdSplit, where we extended Android to allow an application and its advertising to run as separate processes, under separate user-ids, eliminating the need for applications to request permissions on behalf of their advertising libraries, and providing services to validate the legitimacy of clicks, locally and remotely. AdSplit automatically recompiles apps to extract their ad services, and we measure minimal runtime overhead. AdSplit also supports a system resource that allows advertisements to display their content in an embedded HTML widget, without requiring any native code.

1 Introduction

The smartphone and tablet markets are growing in leaps and bounds, helped in no small part by the availability of specialized third-party applications (“apps”). Whether on the iPhone or Android platforms, apps often come in two flavors: a free version, with embedded advertising, and a pay version without. Both models have been successful in the marketplace. To pick one example, the popular Angry Birds game at one point brought in roughly equal revenue from paid downloads on Apple iOS devices and from advertising-supported free downloads on Android devices [10]. They now offer advertising-supported free downloads on both platforms.

We cannot predict whether free or paid apps will dominate in the years to come, but advertising-supported applications will certainly remain prominent. Already, a

cottage industry of companies offer advertising services for smartphone application developers.

Today, these services are simply pre-compiled code libraries, linked and shipped together with the application. This means that a remote advertising server has no way to validate a request it receives from a user legitimately clicking on an advertisement. A malicious application could easily forge these messages, generating revenue for its developer while hiding the advertisement displays in their entirety. To create a clear trust boundary, advertisers would benefit from running separately from their host applications.

In Android, applications must request permission at install time for any sensitive privileges they wish to exercise. Such privileges include access to the Internet, access to coarse or fine location information, or even access to see what other apps are installed on the phone. Advertisers want this information to better profile users and thus target ads at them; in return, advertisers may pay more money to their hosting applications' developers. Consequently, many applications which require no particular permissions, by themselves, suffer *permission bloat*—being forced to request the privileges required by their advertising libraries in addition to any of their own needed privileges. Since users might be scared away by detailed permission requests, application developers would also benefit if ads could be hosted in separate applications, which might then make their own privilege requests or be given a suitable one-size-fits-all policy.

Finally, separating applications from their advertisements creates better fault isolation. If the ad system fails or runs slowly, the host application should be able to carry on without inconveniencing the user. Addressing these needs requires developing a suitable software architecture, with OS assistance to make it robust.

The rest of the paper is organized as follows: in Section 2 we present a survey of thousands of Android applications, and estimate the degree of permission bloat caused by advertisement libraries. Section 3 discusses

the design objectives of AdSplit and how we can borrow ideas from how web advertisements are secured. Section 4 describes our Android-based implementation, and Section 5 quantifies its performance. Section 6 provides details about a simple binary rewriter to adapt legacy apps to use our system. Section 7 considers how we might eliminate native code libraries for advertisements and go with a more web-like architecture. Finally, Section 8 discusses a variety of policy issues.

2 App analysis

The need to monetize freely distributed smartphone applications has given rise to many different ad provider networks and libraries. The companies competing for business in the mobile ad world range from established web ad providers like Google's AdMob to a variety of dedicated smartphone advertising firms.

With so many options for serving mobile ads, many app developers choose to include multiple ad libraries. Additionally, there is a new trend of *advertisement aggregators* that have the aggregator choose which ad library to use in order to maximize profits for the developer.

While we're not particularly interested in advertising market share, we want to understand how these ad libraries behave. What permissions do they require? And how many apps would be operating with fewer permissions, if only their advertisement systems didn't require them? To address these questions, we downloaded approximately 10,000 free apps from the Android Market and the Amazon App Store and analyzed them.

How many ad libraries? Fig 1 shows the distribution of the number of advertisement libraries used by apps in our sample. Of the apps that use advertisements, about 35% include two or more advertising libraries.

Permissions required. We found that some ad libraries need more permissions than those mentioned in the documentation; also, the set of permissions may change with the version of the ad library. Table 1 shows some of the required and optional permission sets for a number of popular Android ad libraries. The permissions listed as optional are not required to use the ad library but may be requested in order to improve the quality of advertisements; for example, some ad libraries will use location information to customize ads. A developer using such a library has the choice of including location-targeted ads or not. Presumably, better targeted ads will bring greater revenue to the application developer.

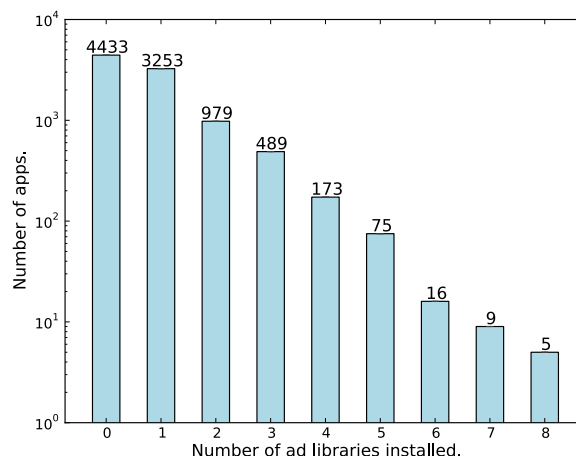


Figure 1: Number of apps with ad libraries installed.

Permission bloat. In Android, an application requests a set of permissions at the time it's installed. Those permissions must suffice for all of the app's needs and for the needs of its advertising library. We decided to measure how many of the permissions requested are used *exclusively* by the advertising library (i.e., if the advertising library were removed, the permission would be unnecessary).

This analysis required decompiling our apps into dex format [3] using the android-apktool [23]. For each app, we then extracted a list of all API calls made. Since advertising libraries have package names that are easy to distinguish, it's straightforward to separate their API calls from the main application. To map the list of API calls to the necessary permissions, we use the data gathered by Felt et al. [18]. This allows us to compute the minimal set of permissions required by an application, with and without its advertisement libraries. We then compare this against the formal list of permissions that each app requests from the system.

There may be cases where an app speculatively attempts to use an API call that requires a permission that was never granted, or there may be dead code that exercises a permission, but will never actually run. Our analysis will err on the side of believing that an application requires a permission that, in fact, it never uses. This means that our estimates of permission bloat are strictly a lower bound on the actual volume of permissions that are requested only to support the needs of the advertising libraries.

Our results, shown in Fig. 2, are quite striking. 15% of apps requesting Internet permissions are doing it for the sole benefit of their advertising libraries. 26% of apps requesting coarse location permissions are doing it for the sole benefit of their advertising libraries. 47% of apps

	Internet	NetworkState	ReadPhoneState	WriteExternalStorage	CoarseLocation	CallPhone
Ad Library						
AdMob [22]	✓	✓			○	
Greystripe [25]	✓	✓	✓			
Millennial Media [36]	✓	✓	✓	✓		
InMobi [29]	✓	○			○	○
MobClix [38]	✓	○	✓			
TapJoy [53]	✓	✓	✓	✓		
JumpTap [32]	✓	✓	✓		○	

✓ (required), ○ (optional)

Table 1: Different advertising libraries require different permissions.

requesting permission to get a list of the tasks running on the phone (the ad libraries use this to check if the application hosting the advertisement is in foreground) are doing it for the sole benefit of their advertising libraries. These results suggest that any architecture that separates advertisements from applications will be able to significantly reduce permission bloat. (In concurrent work to our own, Grace et al. [24] performed a static analysis of 100 thousand Android apps and found advertisement libraries uploading sensitive information to remote ad servers. They also found that some advertisement libraries were fetching and dynamically executing code from remote ad servers.)

3 Design objectives

Advertisement services have been around since the very beginnings of the web. Consequently, these services have adapted to use a wide variety of technologies that should be able to influence our AdSplit design.

3.1 Advertisement security on the web

Fundamentally, a web page with a third-party advertisement falls under the rubric of a *mashup*, where multiple web servers are involved in the presentation of a single web page.

Many web pages isolate advertisements from content by placing ads in an *iframe* [55]. The content hosted in an *iframe* is isolated from the hosting web-page and browsers allow only specific cross frame in-

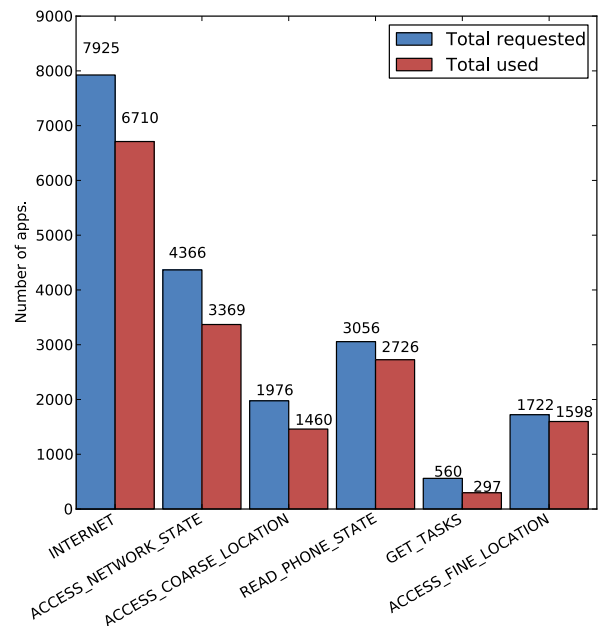


Figure 2: Distribution of types of permissions reduced when advertisements are separated from applications.

teractions [6, 40], protecting the advertisement against intrusions from the host page (although there have been plenty of attacks [51, 47, 50]). Another valuable property of the *iframe* is that it allows an external web server to distinguish between requests coming from the advertisement from requests that might be forged. Standard web security mechanisms assist with this; browsers enforce the *same origin policy*, restricting the host web page from making arbitrary connections to the advertiser. Defenses against cross site request forgery, like the Origin header [5], further aid advertisers in detecting fraudulent clicks.

Adapting these ideas to a smartphone requires significant design changes. Most notably, it's common for Android applications to request the privilege to make arbitrary Internet connections. There is nothing equivalent to the same origin policy, and consequently no way for a remote server to have sufficient context, from any given click request it receives, to determine whether that click is legitimate or fraudulent. This requires AdSplit to include several new mechanisms.

3.2 Adapting these ideas to AdSplit

The first and most prominent design decision of AdSplit is to separate a host application from its advertisements. This separation has a number of ramifications:

Specification for advertisements. Currently the ad libraries are compiled and linked with their corresponding host application. If advertisements are separate, then the host activities must contain the description of which advertisements to use. We introduced a method by which the host activity can specify the particular ad libraries to be used.

Permission separation. AdSplit allows advertisements and host applications to have distinct and independent permission sets.

Process separation. AdSplit advertisements run in separate processes, isolated from the host application.

Lifecycle management. Advertisements only need to run when the host application is running, otherwise they can be safely killed; similarly once the host application starts running, the associated advertisement process must also start running. Our system manages the lifecycle of advertisements.

Screen sharing. Advertisements are displayed inside host activity, so if advertisements are separated there should be a way to share screen real estate between advertisements and host application. AdSplit includes a mechanism for sharing screen real estate.

Authenticated user input. Advertisements generate revenue for their host applications; this revenue is typically dependent on the amount of user interaction with the advertisement. The host application can try to forge user input and generate fraudulent revenue, hence the advertisements should have a way to determine that any input events received from host application are genuine. AdSplit includes a method by which advertising applications can validate user input, validate that they are being displayed on-screen, and pass that verification, in an unforgeable fashion, to their remote server.

In the next section, we will describe how AdSplit achieves these design objectives.

4 Implementation

While many aspects of our design should be applicable to any smartphone operating system, we built our system on Android, and there are a number of relevant Android features that are important to describe.

4.1 Background

Android applications present themselves to the user as one or more *activities*, which are roughly analogous to windows in a traditional window system. Activities in

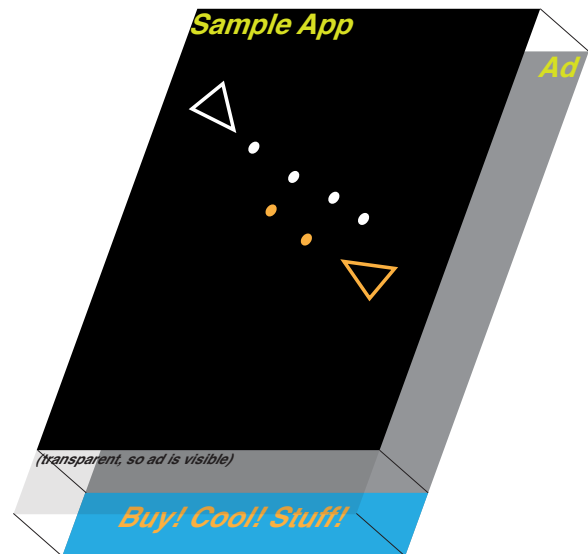


Figure 3: Screen sharing between host and advertisement apps.

Android are maintained on a stack, simplifying the user interface and enabling the “back” button to work consistently across applications. This switching between activities as well as other related functions to activity lifecycle are performed by the ActivityManager service.

When an activity is started, the ActivityManager creates appropriate data structures for the activity, schedules the creation of a process for activity, and puts activity-related information on a stack. There is a separate WindowManager that manages the z-order of windows and maintains their association with activities. The ActivityManager informs the WindowManager about changes to activity configuration. Since we want to factor out the advertising code into a separate process / activity, this will require a variety of changes to ensure that the user experience is unchanged.

An app using AdSplit will require the collaboration of three major components: the host activity, the advertisement activity, and the advertisement service. The host activity is the app that the user wants to run, whether a game, a utility, or whatever else. It then “hosts” the advertisement activity, which displays the advertisement. There is a one-to-one mapping between host activity and advertisement activity instances. The Unix processes behind these activities have distinct user-ids and distinct permissions granted to them. To coordinate these two activities, we have a central advertisement service. The ad service is responsible for delivering UI events to the ad activity. It also verifies that the ad activity is being properly displayed and that the UI clicks aren’t forged. (More on the verification task in Section 4.4.)

AdSplit builds on QUIRE [13], which prototyped a fea-

ture shown in Fig. 3, allowing the host and advertisement activities to share the screen together. First the window for advertisement activity is layered just below the host activity window. The host activity window contains transparent regions where advertisement will be displayed. Standard Android features allow the advertisement activity to verify that the user can actually see the ads.

4.2 Advertisement pairing

In AdSplit, we wish to take existing Android applications and separate out their advertising to follow the model described above. We first must explain the variety of different ways in which an existing application might arrange for an advertisement to be displayed. We will use Google's AdMob system as a running example. Other advertisement systems behave similarly, at least with respect to displaying banner ads. (For simplicity of discussion, we ignore full-screen interstitial ads.)

With current Android applications, if a developer wants to include an advertisement from AdMob in an activity of her application, she imports the AdMob library, and then either declares an AdMob.AdView in the XML layout, or she generates an instance of AdMob.AdView and inserts it in directly into the view hierarchy. This works without issue since all AdMob classes are loaded alongside the hosting application; they are separated only by having different package names.

Once we separate advertisements from applications, neither of these techniques will work, since the code isn't there any more. We first need a new mechanism. Later, in Section 6, we will describe how AdSplit does this transformation automatically.

We added a AppFrame element, which can appear in the XML manifest, allowing the system to attach a subsidiary activity to its host. This results in a distinct activity for the advertisement as well as a local stub to support the same API as if the advertisement code was still local to the host application. The stub packages up requests and passes them onto the advertisement service.

One complication of this process is that advertising libraries like AdMob were engineered to have one copy running in each process. If we create a single, global instance of any given advertising library, it won't have been engineered to maintain the state of the many original applications which hosted it.

Consequently, the advertisement service must manage distinct advertisement applications for each host application. If ten different applications include AdMob, then there need to be ten different AdMob user-ids in the system, mapping one-to-one with the host applications. The advertisement service is then responsible for ensuring that the proper host application speaks to the proper ad-

vertising application.

This is sufficient to ensure that the existing advertising libraries can run without requiring modifications. One complication concerns Android's mechanism for sharing processes across related activities. When a new activity is launched and there is already a process associated with the user-id of the application, Android will launch the new activity in the same process as the old one [2]. If there is already an instance of an activity running, for example, then Android will just resume the activity and bring its activity window to the front of the stack. This is normally a feature, ensuring that there is only a single process at a time for any given application. However, for AdSplit, we need to ensure that advertising apps map one-to-one with hosting apps and we must ensure that their activity windows stay "glued" to their hosts' activities. Consequently, we changed the default Android behavior such that advertisement activities are differentiated based not only by user-id, but also by the host activity. AdSplit thus required modest changes in how activities are launched and resumed as well as how windows are managed.

4.3 Permission separation

With Android's install-time permission system, an application requests every permission it needs at the time of its installation. As we described in Section 2, advertising libraries cause significant bloat in the permission requests made by their hosting applications. Our AdSplit architecture allows the advertisements to run as separate Android users with their own isolated permissions. Host applications no longer need to request permissions on behalf of their advertisement libraries.

We note that AdSplit makes no attempt to block a host application from explicitly delegating permissions to its advertisements. For example, the host application might obtain fine-grained location permissions (i.e., GPS coordinates with meter-level accuracy) and pass these coordinates to an advertising library which lacks any location permissions. Plenty of other Android extensions, including TaintDroid [15] and Paranoid Android [46], offer information-flow mechanisms that might be able to forbid this sort of thing if it was considered undesirable. We believe these techniques are complementary to our own, but we note that if we cannot create a hospitable environment for advertisers, they will have no incentive to run in an environment like AdSplit. We discuss this and other policy issues further in Section 8.

4.4 Click fraud

AdSplit leverages mechanisms from QUIRE [13] to detect counterfeit events, thus defeating the opportunity for an

Android host application to perform a click fraud attack against its advertisers. While a variety of strategies are used to defeat click fraud on the web (see, e.g., Juels et al. [31]), we need distinct mechanisms for AdSplit, since a smartphone is a very different environment from a web browser.

QUIRE uses a system built around HMAC-SHA1 where every process has a shared key with a system service. This allows any process to cheaply compute a “signed statement” and send it anywhere else in the system. The ultimate recipient can then ask the system service to verify the statement. QUIRE uses this on user-generated click events, before they are passed to the host activity. The host activity can then delegate a click or any other UI event, passing it to the advertising activity which will then validate it without being required to trust the host activity. The performance overhead is minimal.

QUIRE has support for making these signed statements meaningful to remote network services. Unlike the web, where we might trust a browser to speak truthfully about the context of an event (see Section 3.1), any app might potentially send any message to any network service. Instead, QUIRE provides a system service that can validate one of these messages, re-sign it using traditional public-key cryptography, and send it to a remote service over the network.

QUIRE’s event delivery mechanism is summarized in Fig. 4. The touch event is first signed by the input system and delivered to the host activity. The stub in the host activity then forwards the touch event to advertisement service which verifies the touch event and forwards it to the advertisement activity instance. This could then be passed to another system service (not shown) which would resign and transmit the message as described above.

Despite QUIRE’s security mechanisms, there are still several ways the host might attempt to defraud the advertiser. First, a host application might save old click events with valid signatures, potentially replaying them onto an advertisement. We thus include timestamps for advertisements to validate message freshness. Second, a host may send genuine click events but move the AdView, we prevent this kind of tampering by allowing the advertisement service to query layout information about the host activity. Third, a host application might attempt to hide the advertising. Android already includes mechanism for an activity to sort out its visibility to the screen [21] (touch events may include a flag that indicates the window is obscured); our advertising service uses these to ensure that the ad was being displayed at the time the click occurred.

It’s also conceivable that the host application could simply drop input events rather than passing them to the advertising application. This is not a concern be-

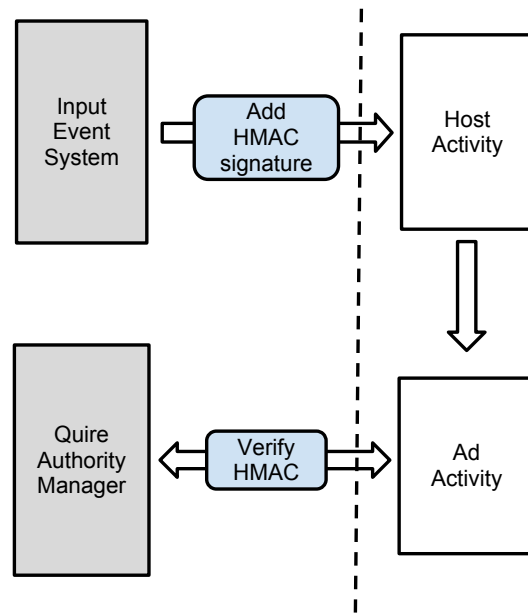


Figure 4: Motion event delivery to the advertisement activity.

cause the host application has no incentive to do this. The host only makes money from clicks that go through, not from clicks that are denied. (Advertising generally works on two different business models: payment per impression and payment per click. In our AdSplit efforts, we’re focused on per-click payments, but the same QUIRE authenticated RPC mechanisms could be used in per-impression systems, with the advertisement service making remotely verifiable statements about the state of the screen.) The host activity can also use a clickjacking attack by anticipating the location of user touch and moving AdView to the intended location. Our implementation currently does not prevent this attack; ads could certainly check that they were visible at the proper location for at least some minimum duration before considering a click to be valid.

4.5 Summary

AdSplit, as we’ve described it so far, would not leverage the QUIRE RPC mechanisms by default, since no off-the-shelf advertising library has been engineered to use it. There are other pragmatic issues, such as how the advertisement applications might be installed and managed. We address these issues in Sections 7 and 8. Nonetheless, we now have a workable skeleton design for AdSplit that we have implemented and benchmarked.

5 Performance

In order to evaluate the performance overhead of our system we performed our experiments on a standard Android developer phone, the Nexus One, which has a 1GHz ARM core (a Qualcomm QSD 8250), 512MB of RAM, and 512MB of internal Flash storage. We conducted our experiments with the phone displaying the home screen and running the normal set of applications that spawn at start up. We replaced the default “live wallpaper” with a static image to eliminate its background CPU load. All of our benchmarks are measured using the Android Open Source Project’s (AOSP) Android 2.3 (“Gingerbread”) plus the relevant portions of QUIRE, as discussed earlier.

Our performance analysis focuses on the effect of AdSplit on user interface responsiveness as well as the extra CPU and memory overhead.

5.1 Effect on UI responsiveness

We performed benchmarking to determine the overhead of AdSplit on touch event throughput. By default Android has a 60 event per second hard coded limit; for our experiments we removed this limit. Table 2 shows the event throughput in terms of number of touch events per second. (The overhead added by our system is due to passing touch events from the host activity to the advertisement activity. There is also additional overhead due to the additional traversal of the view hierarchy in the advertisement activity.) We can see the our system can still support about 183 events per second which is well above the default limit of 60. Furthermore, the Nexus One is much slower than current-generation Android hardware. CPU overhead, even in this extreme case, appears to be a non-issue.

Stock Android	AdSplit	Ratio
229.96	183.12	0.796

Table 2: Comparison of click throughput (Events/sec), averaged over 1 million events.

5.2 Memory and CPU overhead

Measuring memory overhead on Android is complicated since Android optimizes memory usage by sharing read-only data for common libraries. Consequently, if an activity has several copies of a UI widget, the effective overhead of adding a new instance of the same widget is low. Every advertisement library that we examined displays advertisements by embedding a WebView. A WebView is an instance of web browser. When the host ac-

tivity already has a WebView instance, a fairly common practice, and it includes an advertisement, then most of the code for the advertisement WebView will be shared, yielding a relatively low additional overhead for the advertisement. (In our experiments we found out that multiple WebViews in the same activity will share their cookies, which means that an advertisement can steal cookies from any other WebViews in the activity.)

Consequently, in order to determine the actual memory overhead of separating advertisements from their host applications, we need to differentiate between the cases when host activities contain an instance of WebView and when they don’t. We did our measurements by running the AdMob library, both inside the application and in a separate advertisement activity. To measure memory overhead we used procrank [14], which tells us the proportional set size (Pss) and unique set size (Uss). Pss is the amount of memory shared with other processes, divided equally among the processes who share it. Uss is the amount of memory used uniquely by the one process. Table 3 lists our results for the memory measurements.

Activity setup	Memory Overhead (MB)			
	Host Activity		Ad Activity	
	Pss	Uss	Pss	Uss
Without Ad or WebView	2.46	1.44	-	-
Only WebView	5.52	3.30	-	-
Only AdMob	9.67	6.58	-	-
WebView and AdMob	9.82	6.73	-	-
AdMob with AdSplit	2.46	1.56	9.55	6.56
WebView and AdMob with AdSplit	5.15	3.35	9.29	6.58

Table 3: Memory overhead for host and advertisement activities with different system configurations.

In interpreting our results we are primarily concerned with the sum of Pss and Uss. From the table, we see that starting with a simply activity without any WebView (due to AdMob or its own), consumes about 3.9 MB. This increases to about 9 MB if the activity has a WebView. Having AdMob loaded and displaying an advertisement takes about 16.3 MB of memory. When an activity has both WebView and AdMob, the total memory used is only about 16.5 MB, demonstrating the efficiency of Android’s memory sharing.

With AdMob in a separate process, we expect to pay additional costs for Android to manage two separate ac-

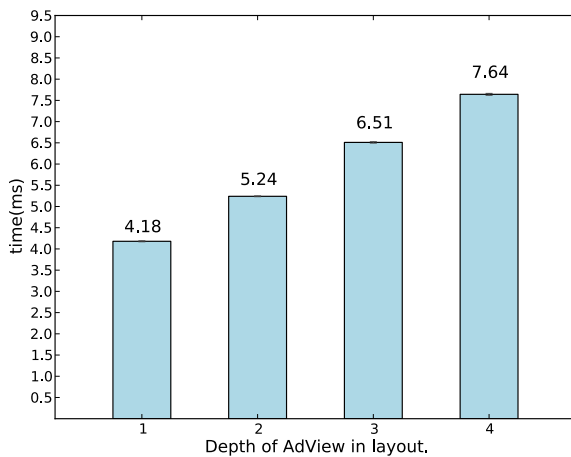


Figure 5: Layout query time vs view depth of host activity (average of 10K runs).

tivities, two separate processes, and so forth. The total memory cost in this configuration, with AdMob running in AdSplit and no other WebView, is about 20.2 MB, roughly a 4 MB increase relative to AdMob running locally. Furthermore, when a separate WebView is running in the host activity, there is no longer an opportunity to share the cost of that WebView. The total memory use in this scenario is 24.4 MB, or roughly an 8 MB increase relative to hosting AdMob locally. We expect we would see similar overheads with other advertising libraries.

The CPU overhead is same as the overhead of additional Dalvik virtual machine on Android. In fact, since the advertisement activities run in the background, they run with lower priority and can be safely killed without any issues.

As discussed in Section 4.4, we allow advertisement service to query layout information (type, position and transparency of views) about the host activity to prevent UI rearrangement attacks. In order to evaluate the overhead of layout information queries we experimented with different view configurations for host activities and varied the depth of AdView in the view hierarchy. Fig. 5 shows how the query overhead varies with view depth. The additional depth adds a small (1 ms) overhead. These queries will run infrequently—only once per click.

In summary, while AdSplit does introduce a marginal amount of additional memory and CPU cost, these will have negligible impact in practice.

6 Separation for legacy apps

The amount of permissions requested by mobile apps and lack of information about how they are used has been a

cause of concern (see, e.g., the U.S. government’s Federal Trade Commission study of privacy disclosures for children’s smartphone apps [17]). To some extent, the potential for information leakage is driven by advertisement permission bloat, so separating out the ad systems and treating them distinctly is a valuable goal.

As we showed in Section 2, a significant number of current apps with embedded advertising libraries would immediately benefit from AdSplit, reducing the permission bloat necessary to host embedded ads. This section describes a proof-of-concept implementation that can automatically rewrite an Android application to use AdSplit. Something like this could be deployed in an app store or even directly on the smartphone itself.

Figure 6 sketches the rewriting process. First the application is decompiled using android-apktool, converting dex bytecode into smali files. (Smali is to dex bytecode what assembly language is to binary machine code; smali is the human-readable version.) Because smali files are organized into directories based on their package names, it’s trivial to distinguish the advertisement libraries from their hosting applications. All we have to do is delete the advertisement code and drop in a stub library, supporting the same API, which calls out to the AdSplit advertisement service. We also analyze the permissions required without the advertisement present (see Section 2), remove permissions which are no longer necessary, and edit the manifest appropriately.

For our proof of concept, we decided to focus our attention on AdMob. Our techniques would easily generalize to support other advertising libraries, if desired. (Alternatively we believe we have a better solution, described next in Section 7.)

Our stub library was straightforward to implement. We manually implemented a handful of public methods from the AdMob library, whereafter we constructed a standard Android IPC message to send to the AdSplit advertising service. It worked.

While it would be tempting to use automated tools to translate an entire API in one go, any commercial implementation would require significant testing and, inevitably, there would be corner cases where the automated tool didn’t quite do the right thing. Instead, since there are a fairly small number of advertising vendors, we imagine that each one would best be supported by hand-written code, perhaps even supplied directly by the vendor in collaboration with an app store that did the rewriting.

Unfortunately, there are a number of significant problems that would stand in the way of an automated rewriting architecture becoming the preferred method of deploying AdSplit.

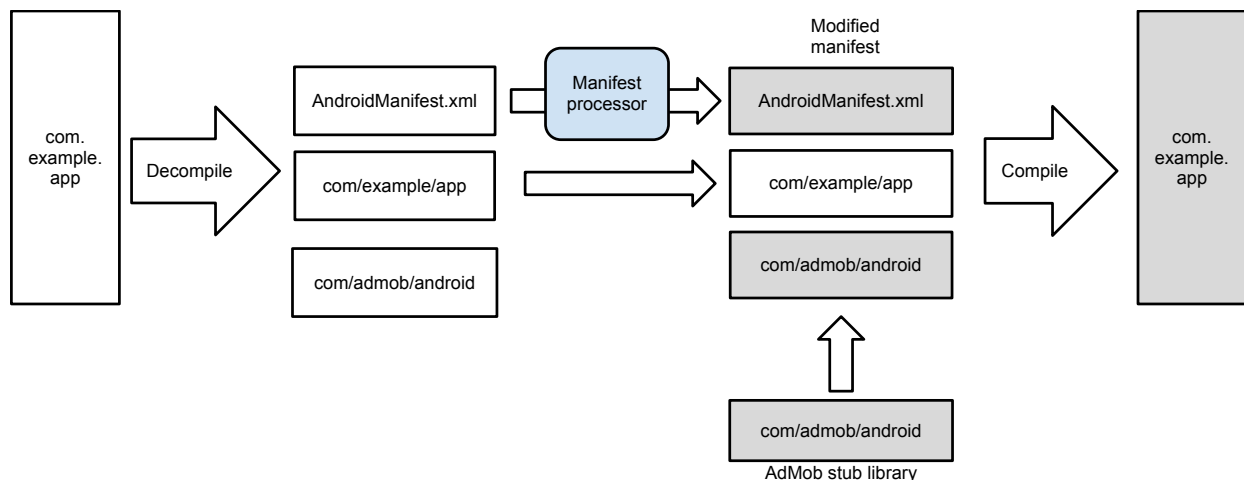


Figure 6: Automated separation of advertisement libraries from their host applications.

Ad installation. When advertisements exist as distinct applications in the Android ecosystem, they will need to be installed somehow. We're hesitant to give the host application the necessary privileges to install a third-party advertising application. Perhaps an application could declare that it had a dependency on a third-party app, and the main installer could hide this complexity from the user, in much the same way that common Linux package installers will follow dependencies as part of the installation process for any given target.

Ad permissions. Even if we can get the ad libraries installed, we have the challenge of understanding what permissions to grant them. Particularly when many advertising libraries know how to make optional use of a permission, such as measuring the smartphone's location if it's allowed, how should we decide if the advertisement application has those permissions? Must we install multiple instances of the advertising application based on the different subsets of permissions that it might be granted by the host application? Alternatively, should we go with a one-size-fits-all policy akin to the web's same-origin-policy? What's the proper "origin" for an application that was installed from an app store? Unfortunately, there is no good solution here, particularly not without generating complex user interfaces to manage these security policies.

Similarly, what should we do about permissions that many users will find to be sensitive, such as learning their fine-grained location, their phone number, or their address book? Again, the obvious solutions involve creating dialog boxes and/or system settings that users must interact with, which few user will understand, and which advertisers and application authors will all hate.

Ad unloading. Like any Android application, an advertisement application must be prepared to be killed at any time—a consequence of Android's resource management system. This could have some destabilizing consequences if the hosting application is trying to communicate with its advertisement and the ad is killed. Also, what happens if a user wants to uninstall an advertising application? Should that be forbidden unless every host application which uses it is also uninstalled?

7 Alternative design: HTML ads

While struggling with the shortcomings outlined above with the installation and permissions of advertising applications, we hit upon an alternative approach that uses the same AdSplit architecture. The solution is to expand on something that advertising libraries are already doing: embedded web views (see Section 5.2).

If an ad creator want to purchase advertising on smartphones, they want to specify their advertisements the same way they do for the web: as plain text, images, or perhaps as a "rich" ad using JavaScript. Needless to say, a wide variety of tools are available to create and manage such ads, and mobile advertising providers want to make it easy for ads to appear on any platform (iPhone, Android, etc.) without requiring heroic effort from the ad creator.

Consequently, all of the advertising libraries we examined simply include a WebView within themselves. All of the native Android code is really nothing more than a wrapper around a WebView. Based on this insight, we suggest that AdSplit will be easiest to deploy by providing a single advertising application, build into the Android core distribution, that satisfies the typical needs of

Android advertising vendors.

Installation becomes a non-issue, since the only advertiser-provided content in the system is HTML, JavaScript, and/or images. We still use the rest of the AdSplit architecture, running the WebView with a separate user-id, in a separate process and activity, ensuring that a malicious application cannot tamper with the advertisements it hosts. We still have the AdSplit advertisement service, leveraging QUIRE, to validate user events before passing them onto the WebView. We only need to extend the WebView's outbound HTTP transactions to include QUIRE RPC signatures, allowing the remote advertising server to have confidence in the provenance of its advertising clicks.

Security permissions are more straightforward. The same-origin-policy, standard across all the web, applies perfectly to HTML AdSplit. Since the Android WebView is built on the same Webkit browser as the standalone "Browser" application, it has the same security machinery to enforce the same-origin-policy.

Keeping all this in mind we introduced a new form of WebView specifically targeted for HTML ads: the AdWebView. The AdWebView is a way to host HTML ads in a constrained manner. We introduced two advertisement specific permissions which can be controlled by the user. These permissions control whether ads can make internet connections or use geolocation features of HTML5.

When an ad inside an AdWebView requests to load a url or performs call to HTML5 geolocation api, the AdWebView performs a permission check to verify if the associated advertisement origin has the needed advertisement permission. These advertisement permissions can be managed by the user.

About the only open policy question is whether we should allow AdSplit HTML advertisements to maintain long-term tracking cookies or whether we should disable any persistent state. Certainly, persistent cookies are a standard practice for web advertising, so they seem like a reasonable feature to support here as well. AdWebView, by default, doesn't support persistent cookies, but it would be trivial to add.

Implementation. We built an advertising application that embeds an AdWebView widget, as discussed above. The host application in this case specifies the URL of the advertisement server to be loaded in the AdWebView at initialization. We were successfully in downloading and running advertisements from our sample advertisement server.

Performance. Memory and performance overheads are indistinguishable from our AdMob experiments. Both versions host a WebView in a separate process, and

it's the same HTML/JavaScript content running inside the WebView.

8 Policy

While AdSplit allows for and incentivizes applications to run distinct from their advertisements, there are a variety of policy and user experience issues that we must still address.

8.1 Advertisement blocking

Once advertisements run as distinct processes, some fraction of the Android users will see this as an opportunity to block advertisements for good. Certainly, with web browsers, extension like Adblock and Adblock Plus are incredibly popular. The Chrome web store lists these two extensions in its top six¹ with "over a million" installs of each. (Google doesn't disclose exact numbers.)

The Firefox add-ons page offers more details, claiming that Adblock Plus is far and away the most popular Firefox extension, having been installed just over 14 million times, versus 7 million for the next most popular extension². The Mozilla Foundation estimates that 85% of their users have installed an extension [39]. Many will install an ad blocker.

To pick one example, Ars Technica, a web site popular with tech-savvy users, estimated that about 40% of its users ran ad blockers [35]. At one point, it added code to display blank pages to these users in an attempt to cajole them into either paying for ad-free "premium" service, or at least configuring their ad blocker to "white list" the Ars Technica website.

Strategies such as this are perilous. Some users, faced with a broken web site, will simply stop visiting it rather than trying to sort out why it's broken. Of course, many web sites instead employ a variety of technical tricks to get around ad blockers, ensuring their ads will still be displayed.

Given what's happening on the web, it's reasonable to expect a similar fraction of smartphone users might want an ad blocker if it was available, with the concomitant arms race in ad block versus ad display technologies.

So long as users have not "rooted" their phones, a variety of core Android services can be relied upon by host applications to ensure that the ads they're trying to host are being properly displayed with the appropriate advertisement content. Similarly, advertising applications (or HTML ads) can make SSL connections to their remote servers, and even embed the proper remote server's public key certificate, to ensure they are downloading data

¹<https://chrome.google.com/webstore/category/popular>

²<https://addons.mozilla.org/en-US/firefox/extensions/?sort=users>

from the proper source, rather than empty images from a transparent proxy.

Once a user has rooted their phone, of course, all bets are off. While it's hard to measure the total number of rooted Android phones, the CyanogenMod Android distribution, which requires a rooted phone for installation, is installed on roughly 722 thousand phones³—a tiny fraction of the hundreds of millions of Android phones reported to be in circulation [43]. Given the relatively small market share where such hacks might be possible, advertisers might be willing to cede this fraction of the market rather than do battle against it.

Consequently, for the bulk of the smartphone marketplace, *advertising apps on Android phones offer greater potential for blocking-detection and blocking-resistance than advertising on the web*, regardless of whether they are served by in-process libraries or by AdSplit. Given all the other benefits of AdSplit, we believe advertisers and application vendors would prefer AdSplit over the status quo.

8.2 Permissions and privacy

Some advertisers would appear to love their ability to learn additional data about the user, including their location, their contacts, and the other apps running on their phone, and so forth. This information can help profile a user, which can help target ads. Targeted ads, in turn, are worth more money to the advertiser and thus worth more money to the hosting application. When we offer HTML style advertisements, with HTML-like security restrictions, the elegance of the solution seems to go against the higher value profiling that advertisers desire.

Leaving aside whether it's *legal* for advertisers to collect this information, we have suggested that a host application could make its own requests that violate the users' privacy and pass these into the AdSplit advertising app. Can we disincentivize such behavior? We hope that, if we can successfully reduce apps' default requests for privileges that they don't really need, then users will be less accustomed to seeing such permission requests. When they do occur, users will push back, refusing to install the app. (Reading through the user-authored comments in the Android Market, many apps with seemingly excessive permission requirements will often have scathing comments from users, along with technical justifications posted by the app authors to explain why each permission is necessary.)

Furthermore, if advertisers ultimately prefer the AdSplit architecture, perhaps due to its improved resistance to click fraud and so forth, then they will be forced to make the trade-off between whether they prefer improved integrity of their advertising platform,

³<http://stats.cyanogenmod.com/>

or whether they instead want less integrity but more privacy-violating user details.

9 Related Work

Android has become quite popular with the security community, with researchers considering many aspects of the system.

9.1 Android advertisements

A number of researchers have considered the Android advertisement problem concurrent with our own work.

AdDroid [45] proposed a separation of advertisements similar to our HTML ads design (outlined in Section 7) by introducing a system service for advertisements. AdDroid does not use our process separation or otherwise defeat a malicious host application.

Leontiadis et al. [33] proposed market mechanisms which through peer pressure and user reviews incentivizes developers to reduce permission bloat due to advertisements. They introduced a separate advertisement service which exposes an intent which apps can subscribe to. Apps display advertisements in a specific UI gadget similar to our AdView. To limit privacy leaks, they monitor the flow of data between advertisement service and apps and use the information to reduce revenue of misbehaving apps and advertisements.

Roesner et al. [49] described user driven access control gadgets (ACGs). The kernel manages input isolation and provides a trusted path to ACGs, solving a problem similar to what we address in AdSplit with Quire signed statements.

While not directly considering security issues, Pathak et al. [44] analyzed the energy use in popular mobile apps and found that 65%-75% of apps energy budget is spent in third-party advertisement libraries. We note that AdSplit's process separation architecture allows the operating system to easily distinguish between advertisements and their hosting applications, allowing for a variety of energy management policies.

9.2 Web security

AdSplit considers an architecture to allow for controlled mashups of advertisements and applications on a smartphone. The web has been doing this for a while (as discussed in Section 3.1). Additionally, researchers have considered a variety of web extensions to further contain browser components in separate processes [26, 48], including constructing browser-based multi-principal operating systems [28, 54].

9.3 JavaScript sandboxes

Caja [37] and ADsafe [1] work as JavaScript sandboxes which use static and dynamic checks to safely host JavaScript code. They use a safe subset of JavaScript, eliminating dangerous primitives like `eval` or `document.write` that could allow an advertisement to take over an entire web page. Instead, advertisements are given a limited API to accomplish what they need. AdSplit can trivially host advertisements built against these systems, and as their APIs evolve, they could be directly supported by our `AdWebView` class. Additionally, because we run the `AdWebView` in a distinct process with its own user-id and permissions, we provide a strong barrier against advertisement misbehavior impacting the rest of the platform.

9.4 Advertisement privacy

Privad [27] and Juels et al. [30] address security issues related to privacy and targeted advertising for web ads. They use client side software that prevents behavior profiling of users and allows targeted advertisements without compromising user privacy.

AdSplit does not address privacy problems related to targeted advertisements but it provides framework for implementing various policies on advertisements.

9.5 Smart phone platform security

As mobile phone hardware and software increase in complexity the security of the code running on a mobile devices has become a major concern.

The Kirin system [16] and Security-by-Contract [12] focus on enforcing install time application permissions within the Android OS and .NET framework respectively. These approaches to mobile phone security allow a user to protect themselves by enforcing blanket restrictions on what applications may be installed or what installed applications may do, but do little to protect the user from applications that collaborate to leak data or protect applications from one another.

Saint [42] extends the functionality of the Kirin system to allow for runtime inspection of the full system permission state before launching a given application. Apex [41] presents another solution for the same problem where the user is responsible for defining run-time constraints on top of the existing Android permission system. Both of these approaches allow users to specify static policies to shield themselves from malicious applications, but don't allow apps to make dynamic policy decisions.

CREPE [11] presents a solution that attempts to artificially restrict an application's permissions based on envi-

ronmental constraints such as location, noise, and time-of-day. While CREPE considers contextual information to apply dynamic policy decisions, it does not attempt to address privilege escalation attacks.

9.5.1 Privilege escalation

XManDroid [8] presents a solution for privilege escalation and collusion by restricting communication at runtime between applications where the communication could open a path leading to dangerous information flows based on Chinese Wall-style policies [7] (e.g., forbidding communication between an application with GPS privileges and an application with Internet access). While this does protect against some privilege escalation attacks, and allows for enforcing a more flexible range of policies, applications may launch denial of service attacks on other applications (e.g., connecting to an application and thus preventing it from using its full set of permissions) and it does not allow the flexibility for an application to regain privileges which they lost due to communicating with other applications.

One feature of QUIRE that is not used in AdSplit is its ability to defeat confused deputy attacks, by annotating IPCs with the entire call chain. In concurrent work to QUIRE, Felt et al. present a solution to what they term "permission re-delegation" attacks against deputies on the Android system [20]. With their "IPC inspection" system, apps that receive IPC requests are poly-instantiated based on the privileges of their callers, ensuring that the callee has no greater privileges than the caller. IPC inspection addresses the same confused deputy attack as QUIRE's "security passing" IPC annotations, however the approaches differ in how intentional deputies are handled. With IPC inspection, the OS strictly ensures that callees have reduced privileges. They have no mechanism for a callee to deliberately offer a safe interface to an otherwise dangerous primitive. Unlike QUIRE, however, IPC inspection doesn't require apps to be recompiled or any other modifications to be made to how apps make IPC requests.

(AdSplit does not require QUIRE's IPC inspection system, and thus also does not require apps to be recompiled to have the semantics described in this paper.)

More recent work has focused on kernel extensions that can observe IPC traffic, label files, and enforce a variety of policies [9, 52]. These systems can enhance the assurance of many of the above techniques by centralizing the policy specification and enforcement mechanisms.

9.5.2 Dynamic taint analysis on Android

The TaintDroid [15] and ParanoidAndroid [46] projects present dynamic taint analysis techniques to preventing runtime attacks and data leakage. These projects attempt to tag objects with metadata in order to track information flow and enable policies based on the path that data has taken through the system. TaintDroid's approach to information flow control is to restrict the transmission of tainted data to a remote server by monitoring the outbound network connections made from the device and disallowing tainted data to flow along the outbound channels.

AdSplit allows ads to run in separate processes but applications can still pass sensitive information to separated advertisements. TaintDroid and ParanoidAndroid can be used to detect and prevent any such flow of information. Thus they are complementary to AdSplit.

10 Future Work

The work in this paper touches on a trend that will become increasingly prevalent over the next several years: the merger of the HTML security model and the smartphone application security model. Today, HTML is rapidly evolving from its one-size-fits-all security origins to allow additional permissions, such access to location information, for specific pages that are granted those permissions by the user. HTML extensions are similarly granted varying permissions rather than having all-or-nothing access [4, 34].

On the flip side, iOS apps originally ran with full, unrestricted access to the platform, subject only to vague policies enforced by human auditors. Only access to location information was restricted. In contrast, the Android security model restricts the permissions of apps, with many popular apps running without any optional permissions at all. Despite this, Android malware is a growing problem, particularly from third-party app stores (see, e.g., [19, 56]). Clearly, there's a need for more restrictive Android security, more like the one-size-fits-all web security model.

While the details of how exactly web apps and smartphone apps will eventually combine, our paper shows where this merger is already underway: when web content is embedded in a smartphone app. Well beyond advertising, a variety of smartphone apps take the strategy of using native code to set up one or more web views, then do the rest in HTML and JavaScript. This has several advantages: it makes it easier to support an app across many different smartphone platforms. It also allows authors to quickly update their apps, without needing to go through a third-party review process.

These trends, plus the increasing functionality in

HTML5, suggest that "native" apps may well be entirely supplanted by some sort of "mobile HTML" variant, not unlike HP/Palm's WebOS, where every app is built this way⁴.

Maybe this will result in a industry battle royale, but it will also offer the ability to ask a variety of interesting security questions. For example, consider the proposed "web intents" standard⁵. How can an "external" web intent interact safely with the "internal" Android intent system? Both serve essential the same purpose and use similar mechanisms. We, and others, will pursue these new technologies toward their (hopefully) interesting conclusions.

11 Conclusion

We have presented AdSplit, an Android-based advertising system that provides advertisers integrity guarantees against potentially hostile applications that might host them. AdSplit leverages several mechanisms from QUIRE to ensure that UI events are correct and to communicate to the outside world in a fashion that hosting applications cannot forge. AdSplit runs with marginal performance overhead and, with our HTML-based design, offers a clear path toward widespread adoption. AdSplit not only protects advertisers against click fraud and ad blocking, it also reduces the need for permission bloat among advertising-supported free applications, and has the potential to reduce the incentive for applications to leak privacy-sensitive user information in return for better advertising revenues.

Acknowledgments

We would like to thank Adrienne Porter Felt, David Wagner, Adam Pridgen, and Daniel Sandler for their valuable feedback. This work builds on our prior QUIRE project. We would like to thank Yuliy Pisetsky and Anhei Shu for their assistance and efforts. This work was supported in part by NSF grants CNS-1117943 and CNS-0524211.

References

- [1] ADsafe. *ADsafe*, Feb. 2012. <http://www.adsafe.org>.
- [2] Android. *Processes and Threads | Android Developers*, Nov. 2011. <http://developer.android.com/guide/topics/fundamentals/processes-and-threads.html>.
- [3] Android Open Source Project. *dex - Dalvik Executable Format*, Nov. 2007. <http://source.android.com/tech/dalvik/dex-format.html>.

⁴<http://developer.palm.com/blog>

⁵<http://webintents.org/>

- [4] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *17th Network and Distributed System Security Symposium (NDSS '10)*, San Diego, CA, Feb. 2010.
- [5] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *15th ACM Conference on Computer and Communications Security (CCS '08)*, Alexandria, VA, Oct. 2008.
- [6] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. In *17th USENIX Security Symposium*, San Jose, CA, July 2008.
- [7] D. F. C. Brewer and M. J. Nash. The Chinese wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, Oakland, CA, May 1989.
- [8] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. XManDroid: A new Android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Universität Darmstadt, Apr. 2011. http://www.trust.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_TRUST/PubsPDF/xmandroid.pdf.
- [9] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on Android. In *19th Network and Distributed System Security Symposium (NDSS '12)*, San Diego, CA, Feb. 2012.
- [10] T. Cheshire. *In depth: How Rovio made Angry Birds a winner (and what's next)*. Wired, Mar. 2011. <http://www.wired.co.uk/magazine/archive/2011/04/features/how-rovio-made-angry-birds-a-winner>.
- [11] M. Conti, V. T. N. Nguyen, and B. Crispo. CRePE: Context-related policy enforcement for Android. In *13th Information Security Conference (ISC '10)*, Boca Raton, FL, Oct. 2010.
- [12] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahhaan, and D. Vanoverberghe. Security-by-contract on the .NET platform. *Information Security Technical Report*, 13(1):25–32, 2008.
- [13] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *20th USENIX Security Symposium*, San Francisco, CA, Aug. 2011.
- [14] eLinux.org. *Android Memory Usage*, Feb. 2012. http://elinux.org/Android_Memory_Usage.
- [15] W. Enck, P. Gilbert, C. Byung-gon, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, pages 393–408, Vancouver, B.C., Oct. 2010.
- [16] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *16th ACM Conference on Computer and Communications Security (CCS '09)*, Chicago, IL, Nov. 2009.
- [17] Federal Trade Commission. *Mobile Privacy for Kids: Current Privacy Disclosures are Disappointing*, Feb. 2012. http://ftc.gov/os/2012/02/120216mobile_apps_kids.pdf.
- [18] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *18th ACM Conference on Computer and Communications Security (CCS '11)*, Chicago, IL, 2011.
- [19] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '11)*, Chicago, IL, Oct. 2011.
- [20] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *20th USENIX Security Symposium*, San Fansisco, CA, Aug. 2011.
- [21] Google. *View: Android developer reference*, Feb. 2011. <http://developer.android.com/reference/android/view/View.html#Security>.
- [22] Google Inc. *Google AdMob Ads Android Fundamentals*, Nov. 2011. <http://code.google.com/mobile/ads/docs/android/fundamentals.html>.
- [23] Google Project Hosting. *android-apktool - A tool for reengineering Android apk files*, Feb. 2012. <http://code.google.com/p/android-apktool>.
- [24] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *5th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '12)*, Tucson, AZ, Apr. 2012.
- [25] GreyStripe Inc. *Android - SDK Integration Overview*, Nov. 2011. <http://wiki.greystripe.com/index.php/Android#AndroidManifest.xml>.
- [26] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *2008 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.
- [27] S. Guha, B. Cheng, and P. Francis. Privad: Practical privacy in online advertising. In *8th Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, Mar. 2011.
- [28] J. Howell, C. Jackson, H. J. Wang, and X. Fan. MashupOS: Operating system abstractions for client mashups. In *11th USENIX Workshop on Hot Topics in Operating Systems (HotOS '07)*, pages 1–7, 2007.
- [29] InMobi. *InMobi Android SDK - Version a300*, Nov. 2011. <http://developer.inmobi.com/wiki/index.php?title=Android>.
- [30] A. Juels. Targeted advertising ... and privacy too. In *2001 Conference on Topics in Cryptology: The Cryptographer's Track at RSA (CT-RSA 2001)*, San Francisco, CA, Apr. 2001.
- [31] A. Juels, S. Stamm, and M. Jakobsson. Combating click fraud via premium clicks. In *16th USENIX Security Symposium*, Boston, MA, 2007.
- [32] Jumtap. *Jumtap Android SDK Integration*, Nov. 2011. https://support.jumtap.com/index.php/Jumtap_Android_SDK_Integration.

- [33] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo. Don't kill my ads! Balancing privacy in an ad-supported mobile application market. In *12th Workshop on Mobile Computing Systems & Applications (HotMobile '12)*, San Diego, CA, Feb. 2012.
- [34] L. Liu, X. Zhang, G. Yan, and S. Chen. Chrome extensions: Threat analysis and countermeasures. In *19th Network and Distributed System Security Symposium (NDSS '12)*, San Diego, CA, Feb. 2012.
- [35] L. McGann. *How Ars Technica's "experiment" with ad-blocking readers built on its community's affection for the site.* Nieman Journalism Lab, Mar. 2010. <http://www.niemanlab.org/2010/03/how-ars-technica-made-the-ask-of-ad-blocking-readers/>.
- [36] Millennial Media. *Millennial Media Android SDK - Version 4.5.0*, Nov. 2011. <http://wiki.millennialmedia.com/index.php/Android>.
- [37] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. *Caja: Safe active content in sanitized JavaScript.* Google, Dec. 2007. <http://google-caja.googlecode.com/files/caja-2007.pdf>.
- [38] Mobclix. *Mobclix SDK Integration Guide Version 3.1.0*, Nov. 2011. https://developer.mobclix.com/help/advertising/sdk_api/android.
- [39] Mozilla Foundation. *How Many Firefox Users Have Add-Ons Installed? 85%!*, June 2011. <http://blog.mozilla.com/addons/2011/06/21/firefox-4-add-on-users/>.
- [40] MSDN. *About Cross-Frame Scripting and Security.*, Oct. 2011. [http://msdn.microsoft.com/en-us/library/ms533028\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ms533028(v=vs.85).aspx).
- [41] M. Nauman, S. Khan, and X. Zhang. Apex: extending Android permission model and enforcement with user-defined runtime constraints. In *5th ACM Symposium on Information, Computer and Communications Security (ASIACCS '10)*, pages 328–332, Beijing, China, Apr. 2010.
- [42] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *25th Annual Computer Security Applications Conference (ACSAC '09)*, Honolulu, HI, Dec. 2009.
- [43] M. Panzarino. *Google: About 190 Million Android Devices Activated Worldwide. That's About 576900 A Day Since May.* The Next Web, Oct. 2011. <http://thenextweb.com/google/2011/10/13/google-190-million-android-devices-activated-worldwide-thats-about-576900-a-day-since-may/>.
- [44] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app? Fine grained energy accounting on smartphones with eprof. In *7th ACM European Conference on Computer Systems (EuroSys '12)*, Bern, Switzerland, Apr. 2012.
- [45] P. Pearce, A. P. Felt, and D. Wagner. AdDroid: Privilege separation for applications and advertisers in Android. In *7th ACM Symposium on Information, Computer and Communications Security (AsiaCCS '12)*, Seoul, Korea, May 2012.
- [46] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: Zero-day protection for smartphones using the cloud. In *Annual Computer Security Applications Conference (ACSAC '10)*, Austin, TX, Dec. 2010.
- [47] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iFRAMEs point to us. In *17th USENIX Security Symposium*, San Jose, CA, July 2008.
- [48] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *4th ACM European Conference on Computer systems (EuroSys '09)*, Nuremberg, Germany, Apr. 2009.
- [49] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-Driven Access Control: Rethinking permission granting in modern operating systems. In *2012 IEEE Symposium on Security and Privacy*, Berkeley, CA, May 2012.
- [50] G. Rydstedt, E. Bursztein, and D. Boneh. Framing attacks on smart phones and dumb routers: Tap-jacking and geolocalization. In *USENIX Workshop on Offensive Technologies (wOOT '10)*, Washington, DC, Aug. 2010.
- [51] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *IEEE Oakland Web 2.0 Security and Privacy (W2SP '10)*, Oakland, CA, May 2010.
- [52] S. Smalley. The case for SE Android. In *Linux Security Summit 2011*, Santa Rosa, CA, Sept. 2011. http://selinuxproject.org/~jmorris/lss2011_slides/caseforseandroid.pdf.
- [53] Tapjoy. *Getting Started with Publisher SDK*, Nov. 2011. <http://knowledge.tapjoy.com/integration-8-x/android/publisher/getting-started-with-offers-sdk>.
- [54] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *18th USENIX Security Symposium*, Montreal, Canada, Aug. 2009.
- [55] World Wide Web Consortium (W3C). *Frames in HTML Documents*, Nov. 2011. <http://www.w3.org/TR/REC-html40/present/frames.html#h-16.5>.
- [56] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *19th Network and Distributed System Security Symposium (NDSS '12)*, San Diego, CA, Feb. 2012.

DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis

Lok Kwong Yan^{†‡}

Heng Yin[†]

[†]*Syracuse University
Syracuse, New York, USA*

[‡]*Air Force Research Laboratory
Rome, New York, USA*

{loyan, heyin}@syr.edu

Abstract

The prevalence of mobile platforms, the large market share of Android, plus the openness of the Android Market makes it a hot target for malware attacks. Once a malware sample has been identified, it is critical to quickly reveal its malicious intent and inner workings. In this paper we present DroidScope, an Android analysis platform that continues the tradition of virtualization-based malware analysis. Unlike current desktop malware analysis platforms, DroidScope reconstructs both the OS-level and Java-level semantics simultaneously and seamlessly. To facilitate custom analysis, DroidScope exports three tiered APIs that mirror the three levels of an Android device: hardware, OS and Dalvik Virtual Machine. On top of DroidScope, we further developed several analysis tools to collect detailed native and Dalvik instruction traces, profile API-level activity, and track information leakage through both the Java and native components using taint analysis. These tools have proven to be effective in analyzing real world malware samples and incur reasonably low performance overheads.

1 Introduction

Android is a popular mobile operating system that is installed in millions of devices and accounted for more than 50% of all smartphone sales in the third quarter of 2011 [22]. The popularity of Android and the open nature of its application marketplace makes it a prime target for attackers. Malware authors can freely upload malicious applications to the Android Market¹ waiting for unsuspecting users to download and install them. Additionally, numerous third-party alternative marketplaces make delivering malicious applications even easier. Indeed recent research has shown that malicious applications exist in both the official and unofficial marketplaces with a rate of 0.02% and 0.2% respectively [41].

¹The Android Market has been superseded by the Android Apps Store in Google Play.

Malware analysis and exploit diagnosis on desktop systems is well researched. It is widely accepted that dynamic analysis is indispensable, because malware is often heavily obfuscated to thwart static analysis. Furthermore, runtime information is often needed for exploit diagnosis. In particular, much work has leveraged virtualization techniques, either whole-system software emulation or hardware virtualization, to introspect and analyze illicit activities within the virtual machine [11, 15, 18, 31, 33, 39, 37].

The advantages of virtualization-based analysis approaches are two-fold: 1) as the analysis runs underneath the entire virtual machine, it is able to analyze even the most privileged attacks in the kernel; and 2) as the analysis is performed externally, it becomes very difficult for an attack within the virtual machine to disrupt the analysis. The downside, however, is the loss of semantic contextual information when the analysis component is moved out of the box. To reconstruct the semantic knowledge, virtual machine introspection (VMI) is needed to intercept certain kernel events and parse kernel data structures [16, 21, 24]. Based on this idea, several analysis platforms (such as Anubis [1], Ether [15], and TEMU [35]) have been implemented.

Despite the fact that Android is based on Linux, it is not straightforward to take the same desktop analysis approach for Android malware. There are two levels of semantic information that must be rebuilt. In the lower level, Android is a Linux operating system where each Android application (or App in short) is encapsulated into a process. Within each App, a virtual machine (known as the Dalvik Virtual Machine) provides a runtime environment for the App's Java components.

In essence, to enable the virtualization-based analysis approach for Android malware analysis, we need to reconstruct semantic knowledge at two levels: 1) OS-level semantics that understand the activities of the malware process and its native components; and 2) Java-level semantics that comprehend the behaviors in the Java com-

ponents. Ideally, to capture the interactions between Java and native components, we need a unified analysis platform that can simultaneously rebuild these two semantic views and seamlessly bind these two views with the execution context.

With this goal in mind, we designed and implemented a new analysis platform, *DroidScope*, for Android malware analysis. *DroidScope* is built on top of QEMU (a CPU emulator [3]) and is able to reconstruct the OS-level and Java-level semantic views completely from the outside. Enriched with the semantic knowledge, *DroidScope* further provides a set of APIs to help analysts implement custom analysis plugins. To demonstrate the capability of *DroidScope*, we have implemented several tools, including native instruction tracer and Dalvik instruction tracer to obtain detailed instruction traces, API tracer to log an App's interactions with the Android system, and taint tracker to analyze information leakage.

We evaluated the performance impacts of these tools on 12 different benchmarks and found that the instrumentation overhead is reasonably low and taint analysis performance (from 11 to 34 times slowdown) is comparable with other taint analysis systems. We further evaluated the capability of these tools using two real world Android malware samples: *DroidKungFu* and *DroidDream*. They both have Java and native components as well as payloads that try to exploit known vulnerabilities. We were able to analyze their behavior without any changes to the virtual Android device, and obtain valuable insights.

In summary, this paper makes the following contributions:

- We describe two-level virtual machine introspection to rebuild the Linux and Dalvik contexts of virtual Android devices. Dalvik introspection also includes a technique to dynamically disable Dalvik Just-In-Time compilation.
- We present *DroidScope*, a new emulation based Android malware analysis engine that can be used to analyze the Java and native components of Android Applications. *DroidScope* exposes an event-based analysis interface with three sets of APIs that correspond to the three different abstraction levels of an Android Device, hardware, Linux and Dalvik.
- We developed four analysis tools on *DroidScope*. The *native instruction tracer* and *Dalvik instruction tracer* provide detailed accounts of the analysis sample's execution, while the *API tracer* provides a high level view of how the sample interacts with the rest of the system. The *taint tracker* implements dynamic taint analysis on native instructions but is capable of tracking taint through Java Objects with the help of the Dalvik view reconstruction. These tools were used to instrument

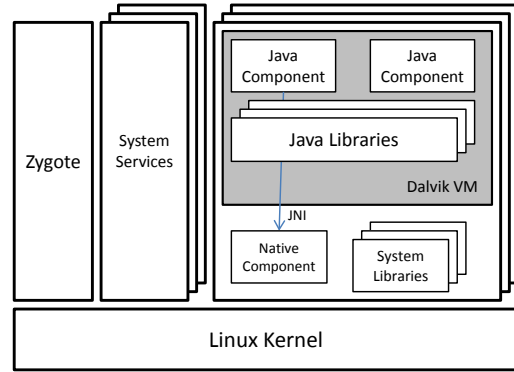


Figure 1: Overview of Android System

and analyze two real-world malware samples: *DroidKungFu* and *DroidDream*.

2 Background and Motivation

In this section, we give an overview of the Android system and existing Android malware analysis techniques to motivate our new analysis platform.

2.1 Android System Overview

Figure 1 illustrates the architecture of the Android system from the perspective of a system programmer. At the lowest level, the Android system uses a customized Linux kernel to manage various system resources and hardware devices. System services, native applications and Apps run as Linux processes. In particular, *Zygote* is the parent process for all Android Apps. Each App is assigned its own unique user ID (*uid*) at installation time and group IDs (*gids*) corresponding to requested permissions. These *uids* and *gids* are used to control access to system resources (i.e. network and file system) like on a normal Linux system.

All Apps can contain both Java and native components. Native components are simply shared libraries that are dynamically loaded at runtime. The Dalvik virtual machine (DVM), a shared library named *libdvm.so*, is then used to provide a Java-level abstraction for the App's Java components. At the same time, the Java Native Interface (JNI) is used to facilitate communications between the native and Java sides.

To create a Java component, an App developer first implements it in Java, compiles it into Java bytecode, and then converts it into Dalvik bytecode. The result is a Dalvik executable called a *dex* file. The developer can also compile native code into shared libraries, *.so* files, with JNI support. The *dex* file, the shared libraries and any other resources, including the *AndroidManifest.xml* file that describes the App, are packaged together into an *apk* file for distribution.

For instance, *DroidKungFu* is a malicious puzzle

game found in alternative marketplaces [25]. Its Java component exfiltrates sensitive information and awaits commands from the bot master. Its native component is used as a shell to execute those commands and it also includes three resource files that are encrypted exploits targeting known vulnerabilities, adb setuid exhaustion and udev [12], in certain versions of Android.

For security analysts, once a new Android malware instance has been identified, it is critical to quickly reveal its malicious functionality and understand its inner-workings. This often involves both static and dynamic analysis.

2.2 Android Malware Analysis

Like malware analysis on the desktop environment, Android malware analysis techniques can fall into two categories: static and dynamic. For static analysis, the sample's dex file can be analyzed by itself or it can be disassembled and further decompiled into Java using tools like *dex2jar* and *ded* [13]. Standard static program analysis techniques (such as control-flow analysis and data-flow analysis) can then be performed. As static analysis can give a complete picture, researchers have demonstrated this approach to be very effective in many cases [20].

However, static analysis is known to be vulnerable to code obfuscation techniques, which are commonplace for desktop malware and are expected for Android malware. In fact, the Android SDK includes a tool named Proguard [34] for obfuscating Apps. Android malware may also generate or decrypt native components or Dalvik bytecode at runtime. Indeed, Droid-KungFu dynamically decrypts the exploit payloads and executes them to root the device. Moreover, researchers have demonstrated that bytecode randomization techniques can be used to completely hide the internal logic of a Dalvik bytecode program [14]. Static analysis also falls short for exploit diagnosis, because a vulnerable runtime execution environment is needed to observe and analyze an exploit attack and pinpoint the vulnerability.

Complementary to static analysis, dynamic analysis is immune to code obfuscation and is able to see the malicious behavior on an actual execution path. Its downside is lack of code coverage, although it can be ameliorated by exploiting multiple execution paths [6, 9, 31]. The Android SDK includes a set of tools, such as *adb* and *logcat*, to help developers debug their Apps. With JDWP (Java Debug Wire Protocol) support, the debugger can even exist outside of the device. However, just like how desktop malware detects and disables debuggers, malicious Android Apps can also detect the presence of these tools, and then either evade or disable the analysis. The fundamental reason is that the debugging components and malware reside in the same execution environment with the same privileges.

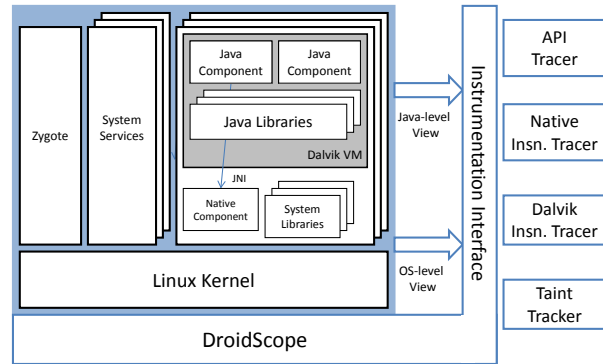


Figure 2: DroidScope Overview

Virtualization based analysis has proven effective against evasion, because all of the analysis components are out of the box and are more privileged than the runtime environment being analyzed, including the malware. Based on dynamic binary translation and hardware virtualization techniques, several analysis platforms [1, 15, 38] have been built for analyzing desktop malware. These platforms are able to bridge the semantic gap between the hardware-level view from the virtual machine monitor and the OS-level view within the virtual machine using virtual machine introspection techniques [16, 21, 24].

However, these tools cannot be immediately used for Android malware analysis. Android has two levels of semantic views, OS and Java, that need to be reconstructed versus the one for desktop malware. To enable virtualization-based analysis for Android malware, we need a unified analysis platform that reconstructs these two levels of views simultaneously and seamlessly binds these two views such that interactions between Java components and native components can be monitored and analyzed.

3 Architecture

DroidScope's architecture is depicted in Figure 2. The entire Android system (including the malware) runs on top of an emulator, and the analysis is completely performed from the outside. By integrating the changes into the emulator, the Android system remains unchanged and different virtual devices can be loaded. To ensure the best compatibility with virtual Android devices, we extended the QEMU [3] based Android emulator that ships with the Android SDK. This is done in three aspects: 1) we introspect the guest Android system and reconstruct OS-level and Java-level views simultaneously; 2) as a key binary analysis technique, we implement dynamic taint analysis; and 3) we provide an analysis interface to help analysts build custom analysis tools. Furthermore, we made similar changes to a different version of QEMU

to enable x86 support.

To demonstrate the capabilities of DroidScope, we have developed several analysis tools on it. The *API tracer* monitors the malware's activities at the API level to reason about how the malware interacts with the Android runtime environment. This tool monitors how the malware's Java components communicate with the Android Java framework, how the native components interact with the Linux system, and how Java components and native components communicate through the JNI interface.

The *native instruction tracer* and *Dalvik instruction tracer* look into how a malicious App behaves internally by recording detailed instruction traces. The Dalvik instruction tracer records Dalvik bytecode instructions for the malware's Java components and the native instruction tracer records machine-level instructions for the native components (if they exist).

The *taint tracker* observes how the malware obtains and leaks sensitive information (e.g., GPS location, IMEI and IMSI) by leveraging the taint analysis component in DroidScope. Dynamic taint analysis has been proposed as a key technique for analyzing desktop malware particularly with respect to information leakage behavior [18, 39]. It is worth noting that DroidScope performs dynamic taint analysis at the machine code level. With semantic knowledge at both OS and Java levels, DroidScope is able to detect information leakage in Java components, native components, or even collusive Java and native components.

We have implemented DroidScope to support both ARM and x86 Android systems. Due to the fact that the ARM architecture is most widely used for today's mobile platforms, we focus our discussion on ARM support, which is also more extensively tested.

4 Semantic View Reconstruction

We discuss our methodology for rebuilding the two levels of semantic views in this section. We first discuss how information about processes, threads, memory mappings and system calls are rebuilt at runtime. This constitutes the OS-level view. Then from the memory mapping, we locate the Dalvik Virtual Machine and further rebuild the Java or Dalvik-level view.

4.1 Reconstructing the OS-level View

The OS-level view is essential for analyzing native components. It also serves a basis for obtaining the Java-level view for analyzing Java components. The basic techniques for reconstructing the OS-level view have been well studied for the x86 architecture and are generally known as virtual machine introspection [16, 21, 24]. We employ similar techniques in DroidScope. We begin by first describing our changes to the Android emulator to

enable basic instrumentation support.

Basic Instrumentation QEMU is an efficient CPU emulator that uses dynamic binary translation. The normal execution flow in QEMU is as follows: 1) a basic block of guest instructions is disassembled and translated into an intermediate representation called TCG (Tiny Code Generator); 2) the TCG code block is then compiled down to a block of host instructions and stored in a code cache; and 3) control jumps into the translated code block and guest execution begins. Subsequent execution of the same guest basic blocks will skip the translation phase and directly jump into the translated code block in the cache.

To perform analysis, we need to instrument the translated code blocks. More specifically, we insert extra TCG instructions during the code translation phase, such that this extra analysis code is executed in the execution phase. For example, in order to monitor context switches, we insert several TCG instructions to call a helper function whenever the translation table registers (system control co-processor `c2_base0` and `c2_base1` in QEMU) are written to.

With basic instrumentation support, we extract the following OS-level semantic knowledge: system calls, running processes, including threads, and the memory map.

System Calls A user-level process has to make system calls to access various system resources and thus obtaining its system call behavior is essential for understanding malicious Apps. On the ARM architecture, the service zero instruction `svc #0` (also known as `swi #0`) is used to make system calls with the system call number in register `R7`. This is similar to x86 where the `int 0x80` instruction is used to transition into privileged mode and the system call number is passed through the `eax` register.

To obtain the system call information, we instrument these special instructions, i.e. insert the additional TCG instructions, to call a callback function that retrieves additional information from memory. For important system calls (e.g. `open`, `close`, `read`, `write`, `connect`, etc.), the system call parameters and return values are retrieved as well. As a result, we are able to understand how a user-level process accesses the file system and the network, communicates with another process, and so on.

Processes and Threads From the operating system perspective, Android Apps are user-level processes. Therefore, it is important to know what processes are active and which one is currently running. In Linux kernel 2.6, the version used in Gingerbread (Android 2.3), the basic executable unit is the task which is represented by the `task_struct` structure. A list of active tasks is maintained in a `task_struct` list which is pointed to by `init_task`. To make this information readily available to analysis tools, DroidScope maintains a shadow task

list with select information about each task.

To distinguish between a thread and a process, we gather a task's process identifier `pid` as well as its thread group identifier `tgid`. The `pgd` (the page global directory that specifies the memory space of a process), `uid` (the unique user ID associated with each App), and the process' name are also maintained as part of the shadow task list. Additionally, our experience has shown that malware often escalates its privileges or spawns child process(es) to perform additional duties. Thus, our shadow task list also contains the task's credentials, i.e. `uid`, `gid`, `euid`, `egid` as well as the process' parent `pid`.

Special attention is paid to a task's name since the `comm` field in `task_struct` can only store up to 15 characters. This is often insufficient to store the App's full name, making it difficult to pinpoint a specific App. To address this issue, we also obtain the complete application name from the command line `cmdline`, which is pointed to by the `mm_struct` structure pointed to by `task_struct`. Note that the command line is located in user-space memory, which is not shared like kernel-space memory where all the other structures and fields reside. To retrieve it, we must walk the task's page table to translate the virtual address into a physical one and then read it based on the physical address.

According to the design of the Linux kernel, the `task_struct` for the current process can be easily located. The current `thread_info` structure is always located at the (`stack pointer & 0x1FFF`), and `thread_info` has a pointer pointing to the current `task_struct`. We iterate through all active tasks by following the doubly linked `task_struct` list. We also update our shadow list whenever the base information changes. We do this by monitoring four system calls `sys_fork`, `sys_execve`, `sys_clone` and `sys_prctl`, and updating the shadow task list when they return.

Memory Map The Dalvik Virtual Machine, libraries and dex files are all memory mapped and we rely on the knowledge of their memory addresses for introspection. Therefore, it is important to understand the memory map of an App. This is especially true for the latest version of Android, Ice Cream Sandwich, since address space layout randomization is enabled by default.

To obtain the memory map of a process, we iterate through the process' list of virtual memory areas by following the `mmap` pointer in the `mm_struct` pointed to by the `task_struct`. To ensure the freshness of the memory map information, we intercept the `sys_mmap2` system call and update the shadow memory map when it returns.

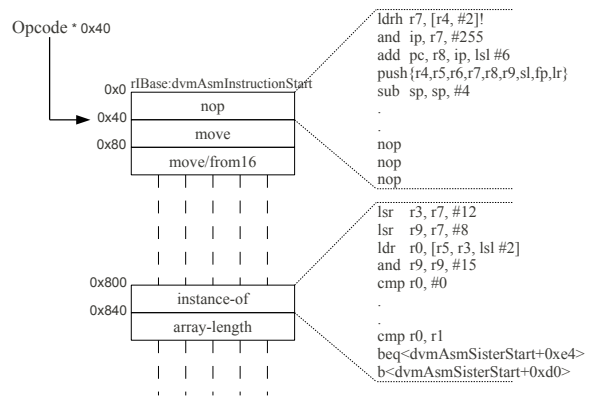


Figure 3: Dalvik Opcode Emulation Layout in *mterp*

4.2 Reconstructing the Dalvik View

With the OS-level view and knowledge of how the DVM operates internally, we are able to reconstruct the Java or Dalvik view, including Dalvik instructions, the current machine state, and Java objects. Some of the details are presented in this section.

Dalvik Instructions The DVM's main task is to execute Dalvik bytecode instructions by translating them into corresponding executable machine code. In Gingerbread and thereafter, it does so in two ways: interpretation and Just-In-Time compilation (JIT) [8].

The interpreter, named *mterp*, uses an offset-addressing method to map Dalvik opcodes to machine code blocks as shown in Figure 3. Each opcode has 64 bytes of memory to store the corresponding emulation code, and any emulation code that does not fit within the 64 bytes use an overflow area, `dvmAsmSisterStart`, (see `instance-of` in Figure 3). This design simplifies the emulation of Dalvik instructions. *mterp* simply calculates the offset, `opcode * 64`, and jumps to the corresponding emulation block.

This design also simplifies the reverse conversion from native to Dalvik instructions as well: when the program counter (R15) points to any of these code regions, we are sure that the DVM is interpreting a bytecode instruction. Furthermore, it is trivial to determine the opcode of the currently executing Dalvik instruction. In DroidScope we first identify the virtual address of `rIBase`, the beginning of the emulation code region, and then calculate the opcode using the formula $(R15 - rIBase)/64$. `rIBase` is dynamically calculated as the virtual address of `libdvm.so` (obtained from the shadow memory map in the OS-level view) plus the offset of `dvmAsmInstructionStart` (a debug symbol). If the debug symbol is not available, we can identify it using the signature for Dalvik opcode number 0 (`nop`).

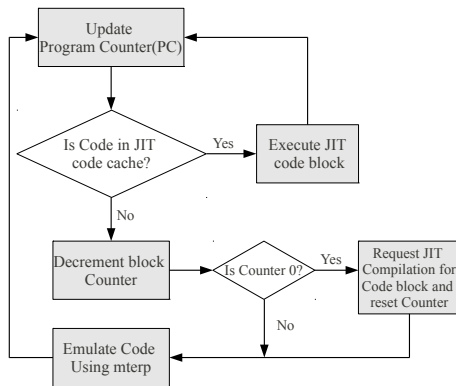


Figure 4: High Level Flowchart of *mterp* and JIT

The Just-In-Time compiler was introduced to improve performance by compiling heavily used, or hot, Dalvik instruction traces (consisting of multiple code blocks) directly into native machine code. While each translation trace has a single entry point, there can be multiple exits known as *chaining cells*. These chaining cells either chain to other translation traces or to default entry points of the *mterp* interpreter. Overall, JIT provides an excellent performance boost for programs that contain many hot code regions, although it makes fine-grained instrumentation more difficult. This is because JIT performs optimization on one or more Dalvik code blocks and thus blurs the Dalvik instruction boundaries.

An easy solution would be to completely disable JIT at build time, but it could incur a heavy performance penalty and more importantly it require changes to the virtual device, which we want to avoid. Considering that we are often only interested in a particular section of Dalvik bytecode (such as the main program but not the rest of system libraries), we choose to *selectively* disable JIT at runtime. Analysis plugins can specify the code regions for which to disable JIT and as a result only the Dalvik blocks being analyzed incur the performance penalty. All other regions and Apps still benefit from JIT.

Figure 4 shows the general flow of the DVM. When a basic block of Dalvik bytecode needs to be emulated, the Dalvik program counter is updated to reflect the new block's address. That address is then checked against the translation cache to determine if a translated trace for the block already exists. If it does, the trace is executed. If it does not then the profiler will decrement a counter for that block. When this counter reaches 0, the block is considered hot and a JIT compilation requested. To prevent thrashing, the counter is reset to a higher value and emulation using *mterp* commences. As can be seen in the flow chart, as long as the requested code is not in the code cache, then *mterp* will be used to emulate the

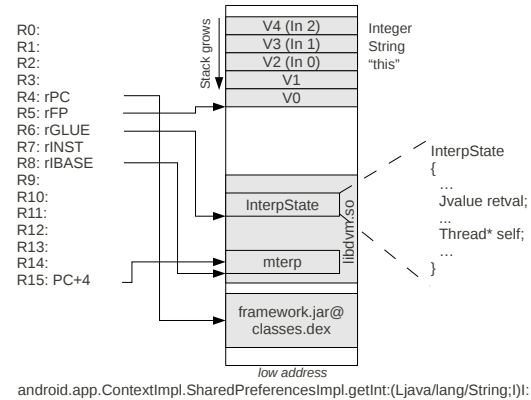


Figure 5: Dalvik Virtual Machine State

code.

The `dvmGetCodeAddr` function is used to determine whether a translated trace exists. It returns `NULL` if a trace does not exist and the address of the corresponding trace if it does. Thus, to selectively disable JIT, we instrument the DVM and set the return value of `dvmGetCodeAddr` to `NULL` for any translated trace we wish to disable. To show that our change to the virtual machine state does not have any ill side-effects, we make the following arguments. First, if the original return value was `NULL` then our change will not have any side effects. Second, if the return value was a valid address, then by setting it to `NULL`, the profile counter is decremented and if 0, i.e. the code region deemed hot again, another compilation request is issued for the block. In this case, the code will be recompiled taking up space in the code-cache. This can be prevented by not instrumenting the `dvmGetCodeAddr` call from the compiler.

In addition to preventing the translated trace from being executed, setting the value to `NULL` also prevents it from being chained to other traces. This is the desired behavior. For the special case where a translation trace has already been chained and thus `dvmGetCodeAddr` is not called, we flush the JIT cache whenever the disabled JIT'ed code regions change. This is done by marking the JIT cache as full during the next garbage collection event, which leads to a cache flush. While this is not a perfect solution, we have found it to be sufficient.

In all cases, the only side effect is wasted CPU cycles due to compilation; the execution logic is unaffected. Therefore, the side effects are deemed inconsequential.

DVM State Figure 5 illustrates how the DVM maintains the virtual machine state. When *mterp* is emulating Dalvik instructions, the ARM registers R4 through R8 store the current DVM execution context. More specifically, R4 is the Dalvik program counter, pointing to the current Dalvik instruction. R5 is the Dalvik stack frame pointer, pointing to the beginning of the current stack

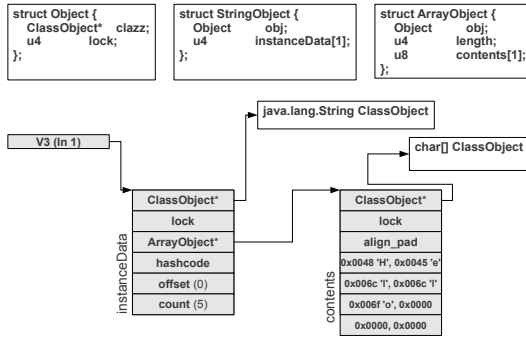


Figure 6: String Object Example

frame. R6 points to the `InterpState` data structure, called `glue`. R7 contains the first two bytes of the current Dalvik instruction, including the opcode. Finally R8 stores the base address of the mterp emulation code for the current DVM instruction. In x86, `edx`, `esi`, `edi` and `ebx` are used to store the program counter, frame pointer, mterp base address and the first two bytes of the instruction respectively. The `glue` object can be found on the stack at a predefined offset.

Dalvik virtual registers are 32 bits and are stored in reverse order on the stack. They are referenced relative to the frame pointer R5. Hence, the virtual register V0 is located at the top of the stack (pointed to by the ARM register R5,) and the virtual register V1 sits on top of V0 in memory, and so forth. All other Dalvik state information (such as return value and thread information) is obtained through `glue` pointed to by R6.

After understanding how DVM state is maintained, we are able to reconstruct the state from the native machine code execution. That is, by examining the ARM registers and relative data structures, we can get the current DVM program counter, frame pointer, all virtual registers, and so on.

Java Objects Java Objects are described using two data structures. Firstly, *ClassObject* describes a class type and contains important information about that class: the class name, where it is defined in a dex file, the size of the object, the methods, and the location of the member fields within the object instances. To standardize class representations, Dalvik creates a *ClassObject* for each defined class type and implicit class type, e.g. arrays. For example there is a *ClassObject* that describes a `char[]` which is used by `java.lang.String`. Moreover, if the App has a two dimensional array, e.g. `String[][]`, then Dalvik creates a *ClassObject* to describe the `String[]` and another to describe the array of the previously described `String[]` class.

Secondly, as an abstract type, *Object* describes a runtime object instance, i.e. member fields. Each *Object*

has a pointer to the *ClassObject* that it is an instance of plus a *tail accumulator array* for storing all member fields. Dalvik defines three types of *Objects*, *DataObject*, *StringObject* and *ArrayObject* that are all pointed to by generic *Object*s*. The correct interpretation of any *Object** fully depends on the *ClassObject* that it points to.

We use a simple String ("Hello") to illustrate the interpretation process. Figure 6 depicts the different data structures involved as well as the struct definitions on top. To access the String, we first follow the reference in the virtual register V3. Since Java references are simply *Object*s*, V3 points to an *Object*. To determine the type of the object, we follow the first 4 bytes to the *ClassObject* structure. This *ClassObject* instance describes the `java.lang.String` class. Internally, Dalvik does not store the String data inside the *StringObject* and instead use a `char[]`. Consequently, `instanceData[0]` is used to store the reference to the corresponding `char[]` object and `instanceData[3]` is used to store the number of characters in the String, 5 in this case.

We then obtain the String's data by following `instanceData[0]` to the character array. Once again we must follow the *Object** within the new object to correctly interpret it as an *ArrayObject*. Note that since ARM EABI requires all arrays to be aligned to its element size and `u8` is 8 bytes in length, we inserted an implicit 4 byte `align_pad` into the *ArrayObject* to ensure that the `contents` array is properly aligned. Given the length of the String from the *StringObject* and the corroborating length in the *ArrayObject*, the "Hello" String is found in the `contents` array encoded in UTF-16.

4.3 Symbol Information

Symbols (such as function name, class name, field name, etc.) provide valuable information for human analysts to understand program execution. Thus, DroidScope seeks to make the symbols readily available by maintaining a symbol database. For portability and ASLR support, we use one database of offsets to symbols per module. At runtime, finding a symbol by a virtual address requires first identifying the containing module using the shadow memory map, and then calculating the offset to search the database.

Native library symbols are retrieved statically through *objdump* and are usually limited to Android libraries since malware libraries are often stripped of all symbol information. On the other hand, Dalvik or Java symbols are retrieved dynamically and static symbol information through *dexdump* is used as a fallback. This has the advantage of ensuring the best symbol coverage for optimized dex files and even dynamically generated Dalvik bytecode.

	NativeAPI	LinuxAPI	DalvikAPI
Events	instruction begin/end	context switch	Dalvik instruction begin
	register read/write	system call	method begin
	memory read/write	task begin/end	
	block begin/end	task updated	
Query & Set		memory map updated	
	memory read/write	query symbol database	query symbol database
	memory r/w with pgd	get current context	interpret Java object
	register read/write	get task list	get/set DVM state
	taint set/check		taint set/check objects
			disable JIT

Table 1: Summary of DroidScope APIs

We rely on the data structures of DVM to retrieve symbols at runtime. For example, the *Method* structure contains two pointers of interest. `insns` points to the start of the method’s bytecode, the symbol address, and `name` points to the name. Conveniently, the `glue` structure pointed to by `R6` has a field `method` that points to the *Method* structure for the currently executing method.

There are times when this procedure fails though, e.g. if the corresponding page of the dex file has not been loaded into memory yet. In these cases, we first try to look up the information in a local copy of the corresponding dex file, and if that fails as well, use the static symbol information from *dexdump*. DroidScope uses this same basic method of relying on the DVM’s data structures to retrieve class and field names as well.

5 Interface & Plugins

DroidScope exports an event based interface for instrumentation. We describe the general layout of the APIs, present an example of how tools are implemented, and finally describe available tools in this section.

5.1 APIs

DroidScope defines a set of APIs to facilitate custom analysis tool development. The APIs provide instrumentation on different levels: native, OS and Dalvik, to mirror the context levels of a real Android device. At each level, the analysis tool can register callbacks for different events, and also query or set various kinds of information and controls. Table 1 summarizes these APIs.

At the native level, one can register callbacks for instruction start and end, basic block start and end, memory read and write, and register read and write. One can also read and write memory and register content. As taint analysis is implemented at the machine code level, one can also set and check taint in memory and registers. Currently, the taint propagation engine only supports copy and arithmetic operations, control flow dependencies are not tracked.

At the OS level, one can register callbacks for context switch, system call, task start, update (such as process

name), and end, and memory map update. One can also query symbols, obtain the task list, and get the current execution context (e.g., current process and thread). At the Dalvik level, one can instrument at the granularity of Dalvik instructions and methods. One can query the Dalvik symbols, parse and interpret Java objects, read and modify DVM state, and selectively disable JIT for certain memory regions. Through the Dalvik-view, one can also set and check taint in Java Objects as well.

5.2 Instrumentation Optimization

A general guideline for performance optimization in dynamic binary translation is to shift computation from the execution phase to the translation phase. For instance, if we need to instrument a function call at address x using basic blocks, then we should insert the instrumentation code for the block at x when it is being translated instead of instrumenting every basic block and look for x at execution time.

We follow this guideline in DroidScope. Consequently, our instrumentation logic becomes more complex. When registering for an event callback, one can specify a specific location (such as a function entry) or a memory range (to trace instructions or functions within a particular module). Therefore, our instrumentation logic supports single value comparisons and range checks for controlling when and where event callbacks are inserted during the translation phase.

The instrumentation logic is also dynamic, because we often want to register and unregister a callback at execution time. For example, when the virtual device starts, only the OS-view instrumentation is enabled so the Android system can start quickly as usual. When we start analyzing an App, instrumentation code is inserted to reconstruct the Dalvik view and to perform analysis as requested by the plugin. When instrumenting a function return, the return address will be captured from the link register `R14` at the function entry during execution, and a callback is registered at the return address. After the function has returned, this callback is removed. Then when the analysis has finished, other instrumentation code is removed as well. To maintain consistency, DroidScope invalidates the corresponding basic blocks in the translated code cache whenever necessary so that the new instrumentation logic can be enforced. Hence, the instrumentation logic in DroidScope is complex and dynamic. These details are hidden from the analysis plugins.

5.3 Sample Plugin

Figure 7 presents sample code for implementing a simple Dalvik instruction tracer. The `_init` function at L19 will be invoked once this plugin is loaded in DroidScope. In `_init`, it specifies which program to analyze by calling the

```

1. void opcode_callback(uint32_t opcode) {
2.     printf("[%x] %s\n", GET_RPC, opcodeToStr(opcode));
3. }
4.
5. void module_callback(int pid) {
6.     if (bInitialized || (getIBase(pid) == 0))
7.         return;
8.
9.     gva_t startAddr = 0, endAddr = 0xFFFFFFFF;
10.
11.     addDisableJITRange(pid, startAddr, endAddr);
12.     disableJITInit(getGetCodeAddrAddress(pid));
13.     addMterpOpcodesRange(pid, startAddr, endAddr);
14.     dalvikMterpInit(getIBase(pid));
15.     registerDalvikInsnBeginCb(&opcode_callback);
16.     bInitialized = 1;
17. }
18.
19. void _init() {
20.     setTargetByName("com.andhuhu.fengyinchuanshuo");
21.     registerTargetModulesUpdatedCb(&module_callback);
22. }

```

Figure 7: Sample code for Dalvik Instruction Tracer

`setTargetByName` function. It also registers a callback `module_callback` to be invoked when module information is updated. `module_callback` will check if the DVM is loaded and if so, disable JIT for the entire memory space (L9 and L11.) It also registers a callback, `opcode_callback`, for Dalvik instructions. When invoked, `opcode_callback` prints the opcode information.

This sample code will print all Dalvik instructions for the specified App, including the main program and all the libraries. If we are only interested in the execution of the main program, we can add a function call like `getModAddr("example@classes.dex", &startAddr, &endAddr)` at L10. This function locates the dex file in the shadow memory map and stores its start and end addresses in the appropriate variables. The rest of the code can be left untouched.

5.4 Analysis Plugins

To demonstrate the capability of DroidScope for analyzing Android malware, we have implemented four analysis plugins: API tracer, native instruction tracer, Dalvik instruction tracer, and taint tracker.

API tracer monitors how an App (including Java and native components) interacts with the rest of the system through system and library calls. We first log all of the App's system calls by registering for system call events. We then build a whitelist of the virtual device's built-in native and Java libraries. As modules are loaded into memory, any library not in the whitelist is marked for analysis. We instrument the `invoke*` and `execute*` Dalvik bytecodes to identify and log method invocations, including those of the sample. The log contains the currently executing Java thread, the calling address, the method being invoked as well as a dump of its input parameters. Since Java Strings are heavily used, we try to convert all Strings into native strings before logging them. We then instrument the `move-result*` bytecode instructions to detect when system methods return and gather the return values. We do not instrument any

of the other bytecodes to improve performance. To log library calls from the App's native components, we register for the block end event for blocks that are located in the App's native components. When the callback for the block end event is invoked, we check if the next block is within the Apps native components or not. If not, we log this event.

Native instruction tracer registers ARM or x86 instruction callbacks to gather information about each instruction including the raw instruction, its operands (register and memory) and their values.

Dalvik instruction tracer follows the basic logic of the above example and logs the decoded instruction to a file in the `dexdump` format. The operands, their values and all available symbol information, e.g. class, field and method names, are logged as well.

Taint tracker utilizes the dynamic taint analysis APIs to analyze information leakage in an Android App. It specifies sensitive information sources (such as IMEI, IMSI, and contact information) as tainted and keeps track of taint propagation at the machine code level until they reach sinks, e.g. `sys_write` and `sys_send`. With the OS and Dalvik views, it further creates a graphical representation to visualize how sensitive information has leaked out. To construct the graph, we first identify function and method boundaries. Whenever taint is propagated, we add a node to represent the currently executing function or method and nodes for the tainted memory locations. Since methods operate on Java Objects, we further try to identify the containing Object and create a node for it instead of the simple memory location. Currently, we only do this check against the method's input parameters and the current Object, e.g. "this". Further improvements are left as future work.

To identify method boundaries, we look for matching `invoke*` or `execute*` and `move-result*` Dalvik instructions. We do not rely on the `return*` instructions since they are executed in the invokee context, which might not be instrumented, e.g. inside an API. Since there are multiple ways for native code to call and return from functions plus malicious code is known to jump into the middle of functions, we do not rely on native instructions to determine function boundaries. Instead, we treat the nearest symbol that is less than or equal to the jump target in the symbol database as the function.

6 Evaluation

We evaluated DroidScope with respect to efficiency and capability. To evaluate efficiency, we used 7 benchmark Apps from the official Android Market: AnTuTu Benchmark (ABenchMark) by AnTuTu, CaffeineMark by Ravi Reddy, CF-Bench by Chainfire, Mobile processor benchmark (Multicore) by Andrei Karpushonak, Benchmark by Softweg, and Linpack by GreeneComputing. We then

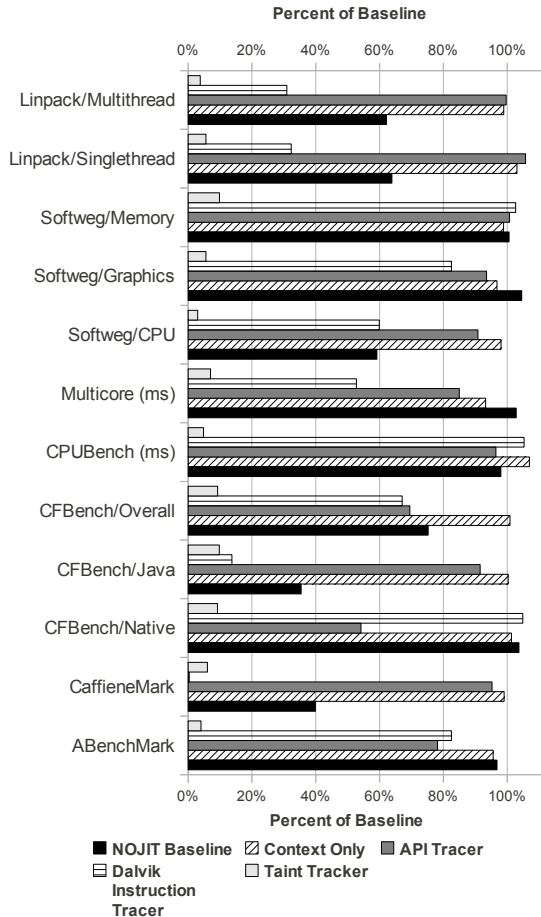


Figure 8: Benchmark Results

ran the benchmarks while using the different automatic analysis tools described above on the benchmarks themselves. The results are presented in Section 6.1. To evaluate capability, we analyzed two real world Android malware samples: DroidKungFu and DroidDream in detail, which will be presented in Sections 6.2 and 6.3. These samples were obtained from the Android Malware Genome project [40].

Experimental Setup All experiments were conducted on an Acer 4830TG with a Core i5 @ 2.40GHz and 3GB of RAM running Xubuntu 11.10. The Android guest is a Gingerbread build configured as "user-eng" for ARM with the Linux 2.6.29 kernel and uses the QEMU default memory size of 96 MB. No changes were made to the Android source.

6.1 Performance

To measure the performance impact of instrumentation, we took the analysis tools and targeted the benchmark Apps while the Apps performed their tests. This was repeated 5 times. As the baseline, we ran these benchmarks

on the default Android emulator without any instrumentation. Since DroidScope selectively disables JIT on the Apps, we also obtained a NOJIT baseline with JIT completely disabled at build time. The performance results are summarized in the bar chart in Figure 8. Each tool is associated with a set of bars that shows its benchmark results (y-axis) relative to the baseline as a percentage. The ARM Instruction Tracer results are excluded as they are similar to the taint tracker results.

Please note that the benchmarks are not perfect representations of performance as evidenced by the $> 100\%$ results. For example, in CPUBenchmark the standard deviation, σ , for Baseline, Dalvik tracer and Context Only is only 1%. This means that the results are consistent for each plugin, but might not be across plugins. Furthermore, we removed the Softweg filesystem benchmarking results due to high variability, $\sigma > 27\%$.

We can see from Figure 8 that the overhead (Context Only) of reconstructing the OS-level view is very small, up to 7% degradation. The taint tracker has the worst performance as expected, because it registers for instruction level events. The taint tracker incurs 11x to 34x slowdown, which is comparable to other taint analysis tools [10, 39] on the x86 architecture. A special case is seen in the Dalvik instruction tracer result for CaffeineMark. This result is attributed to the fact that the tracer dynamically retrieves symbol information from guest memory for logging.

The benefits of dynamically disabling JIT is evident in some Java based benchmarks such as Linpack, CFBench/Java and CaffeineMark. For those benchmarks, the API tracer's performance is greater than that of the NOJIT Baseline, despite the fact that instrumentation is taking place. This difference is due to Java libraries, such as String methods, still benefiting from JIT in the API tracer.

6.2 Analysis of DroidKongFu

The DroidKungFu malware contains three components. First, the core logic is implemented in Java and is contained within the `com.google.ssearch` package. This is the main target of our investigation. Second are the exploit binaries which are encrypted in the apk, decrypted by the Java component and then subsequently executed. Third is a native library that is used as a shell. It contains JNI exported functions that can run shell commands and is the main interface for command and control. Unfortunately the command and control server was unavailable at the time of our test and thus we did not analyze this feature.

Discovering the Internal Logic We began our investigation by running the API tracer on the sample and analyzing the log. We first looked for system calls of interest and found a `sys.open` for a file named "gjsvro".

```

getPermission {
  if checkPermission() then doSearchReport(); return
  if !isVersion221() then
    if getPermission1() then return
  if exists("bin/su" or "xbin/su") then
    getPermission2(); return
  if !isVersion221() then getPermission3(); return
}

```

Figure 9: getPermission Pseudocode

There was also a subsequent *sys_write* to the file from a byte array. We later found that this array is actually part of a Java *ArrayObject* which was populated by the *Utils.decrypt* method, which is part of DroidKungFu. Since *decrypt* takes a byte array as the parameter, we were able to search backwards and identify that this particular array was read from an asset inside the App’s package file called “gjsvro”. It means that during execution, DroidKungFu decrypts an asset from its package and generates the “gjsvro” file. We then found that DroidKungFu called *Runtime.exec* with parameters “chmod 4755” and the name of the file, making the file executable and setting the *setuid* bit. After that, it called *Runtime.exec* again for “su” which led to a *sys_fork*. Furthermore, the file path for “gjsvro” was then written to a *ProcessImpl* *OutputStream*, followed immediately by “exit”. Since this stream is piped to the child’s *stdin*, we know that the intention of “su” was to open a shell which is then used to execute “gjsvro” followed by “exit” to close the shell. This did not work though since “su” did not execute successfully.

Next we used the Dalvik instruction tracer to obtain a Dalvik instruction trace. The trace showed that the *decrypt* and *Runtime.exec* methods were invoked from a method called *getPermission2*, which was called from *getPermission* following a comparison using the result of *isVersion221* and some file existence checks. To get a more complete picture of the *getPermission* method, we ran *dexdump* and built the overview pseudocode shown in Figure 9. It is evident that to explore the *getPermission1* and *getPermission3*, we must instrument the sample and change the return values of the different method invocations.

With the Dalvik view support, we manipulated the return values of *isVersion221* and *exists* methods and were able to explore all three methods *getPermission1*, *getPermission2*, and *getPermission3*. They are essentially different ways to obtain the root privilege on different Android configurations. *getPermission1* and *getPermission2* only uses the “gjsvro” exploit. The main difference is that *getPermission1* uses *Runtime.exec* to execute the exploit while the other uses the “su” shell. On the other hand, *getPermission3* decrypts “ratc”, “killall” (a wrapper for “ratc”) and “gjsvro” and executes them using its own native library. As the API tracer monitors both

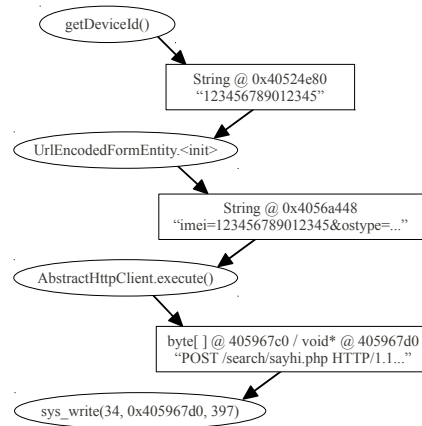


Figure 10: Taint Graph for DroidKungFu

Java and native components, our logs show that the library then calls *sys_vfork* and *sys_execve* to execute the commands. This indicates that *getPermission3* was trying to run both “udev” exploit and “rage against the cage” (ratc) exploits.

Analyzing Root Exploits Since Gingerbread has already been patched against these exploits, they never executed correctly. To further analyze these root exploits, we first needed to remove the corresponding patches from the virtual device build. Here we focus on “ratc,” since “udev” is analyzed in the same manner. Due to space constraints we present the exploit diagnosis of “ratc” in Appendix A.

We first ran the API tracer on the ratc exploit, but did not observe any malicious behavior in the API log. We did see suspicious behavior in the process log provided as part of the OS-view reconstruction. Particularly, we observed that numerous ratc processes (descendants of the original ratc process) were spawned, the *abdb* process with uid 2000 ended, followed by more ratc processes and then by an *abdb* process with uid 0 or root. This signifies that the attack was successful. It is worth noting that the traditional *adb* based dynamic analysis would fail to observe the entire exploiting process because *abdb* is killed at the beginning.

Further analysis of the logs and descendent processes showed that there are in fact three types of ratc processes. The first is the original ratc process that simply iterates through the */proc* directory looking for the pid of the *abdb* process. Its child then forked itself until *sys_fork* returned -11 or EAGAIN. At this point it wrote some data to a pipe and resumed forking. In the grandchild process we see a call to *sys_kill* to kill the *abdb* process followed by attempts to locate the *abdb* process after it re-spawns.

Triggering Data leakage Reverting back to the default Gingerbread build, we sought to observe the information leakage behavior in *doSearchReport*. As depicted

in Figure 9, this involves instrumenting *checkPermission* during execution of *getPermission*. The Dalvik instruction trace shows that *doSearchReport* invokes *updateInfo*, which obtains sensitive information about the device including the device model, build version and IMEI amongst other things. We also observed outgoing HTTP requests, which failed because the server was down. We then redirected these HTTP requests to our own HTTP server by adding a new entry into */etc/hosts*. To further analyze this information leakage, we used the taint tracker and built a simplified taint propagation graph, which is shown in Figure 10. Objects, both Java and native, are represented by rectangular nodes while methods are represented by oval nodes. We see that *UrlEncodedFormEntity* (the constructor) propagated the original tainted IMEI number in the String @ 0x40524e80 to a second String that looks like an HTTP request. The taint then propagated to a byte array at 0x405967c0 by *AbstractHttpClient.execute*. We finally see the taint arriving at the sink at *sys.write*. Note that *sys.write* used a void* at 0x405967d0, which is the contents array of the byte array Object (see the StringObject example in Section 4.2). This is expected since JNI provides direct access to arrays to save on the cost of *memcpy*.

6.3 Analysis of DroidDream

Like analyzing DroidKungFu, we first used the API tracer to get a basic understanding of DroidDream, and then obtained instruction traces and analyzed information leakage.

From the log generated by the API tracer and the shadow task list, we found that there are two DroidDream processes. “com.droiddream.lovePositions,” the main process, does not exhibit any malicious behavior except using *Runtime.exec* to execute “logcat -c” which clears Android’s internal log. Again, this behavior indicates that traditional Android debugging tools fall short for malware analysis.

“com.droiddream.lovePositions:remote,” the other process, is the malicious one. The logs show that DroidDream retrieves the IMSI number along with other sensitive information like IMEI, and encodes them into an XML String. Then we observed a failed attempt to open a network connection to 184.105.245.17:8080. In order to observe this networking behavior, we instrument the return values of *sys.connect* and *sys.write* to make DroidDream believe these network operations are successful.

Using the taint tracker, we marked these information sources as tainted and obtained taint propagation graphs, which confirm that DroidDream did leak sensitive information from these sources to a remote HTTP server. The graph for leaking IMSI information is illustrated in Figure 11. We simplified the graph and annotated it to in-

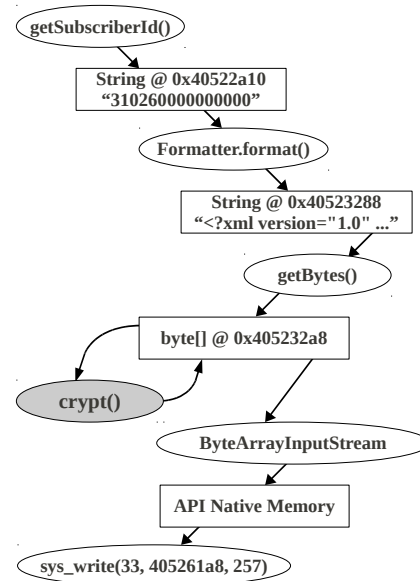


Figure 11: Taint Graph for DroidDream

clude *crypt* which is the DroidDream method used to xor-encrypt the byte array. The graph shows that *getSubscriberId* is used to obtain the IMSI from the system as a String @ 0x40522a10. The IMSI String, along with other information, is then encoded into an XML format using *format*. The resulting String is then converted into a byte[] @ 0x405232a8 for encryption by *crypt*. The encrypted version is used to create a *ByteArrayInputStream*. For brevity, we use a generic “API Native Memory” node to illustrate that the taint further propagates through memory until the eventual sink at *sys.write*.

We further investigated the *crypt* method by augmenting the Dalvik instruction tracer to track taint propagation and generate a taint-annotated Dalvik instruction trace. Not only do we see the byte array being xor-ed with a static field name “KEYVALUE,” we also see that the encryption is being conducted on the byte[] in-place. A snippet of the trace log is depicted in Figure 12.

DroidDream also includes the udev and ratc exploits (unencrypted), plus the native library terminal like DroidKungFu. Since we have already analyzed them in DroidKungFu, we skipped the analysis on them in DroidDream.

7 Discussion

Limited Code Coverage Dynamic analysis is known to have limited code coverage, as it only explores a single execution path at a time. To increase code coverage, we may explore multiple execution paths as demonstrated in previous work [6, 9, 31]. In the experiments, we demonstrated that we can discover different execution paths by manipulating the return values of system calls, native

```

[43328f40] aget-byte v2(0x01), v4(0x405232a8), v0(186)
  Getting Tainted Memory: 40523372(2401372)
  Adding M@410acce4(42c5cec) len = 4
[43328f44] sget-object v3(0x0000005e), KEYVALUE// field@0003
[43328f48] aget-byte v3(0x88), v3(0x4051e288), v1(58)
[43328f4c] xor-int/2addr v2(62), v3(41)
  Getting Tainted Memory: 410acce4(42c5cec)
  Adding M@410acce4(42c5cec) len = 4
[43328f4e] int-to-byte v2(0x17), v2(23)
  Getting Tainted Memory: 410acce4(42c5cec)
  Adding M@410acce4(42c5cec) len = 4
[43328f50] aput-byte v2(0x17), v4(0x405232a8), v0(186)
  Getting Tainted Memory: 410acce4(42c5cec)
  Adding M@40523372(2401372) len = 1

```

Figure 12: Excerpt of Dalvik Instruction Trace for DroidDream. A Dalvik instruction entry shows the location of the current instruction in square brackets, the decoded instruction plus the values of the virtual registers in parenthesis. A taint log entry is indented and shows tainted memory being read or written to. The memory’s physical address is shown in parenthesis and the total bytes tainted is represented by ”len.”

APIs and even internal Dalvik methods of the App. This simple approach works fairly well in practice although a more systematic approach is desirable. One method is to perform symbolic execution to compute path constraints and then automatically explore other feasible paths. We have not yet implemented symbolic execution and leave it as future work. In particular, we seek to use tainting in conjunction with the Dalvik view to implement a symbolic execution engine at the Dalvik instruction level.

Detecting and Evading DroidScope In the desktop environment, malware becomes increasingly keen to the execution environment. Emulation-resistant malware detect if they are running within an emulated environment and evade analysis by staying dormant or simply crashing themselves. Researchers have studied this problem for desktop malware [2, 26, 36]. The same problem has not arisen for Android malware analysis. However, as DroidScope or similar analysis platforms become widely adopted to analyze Android malware, we anticipate similar evasion techniques will eventually appear. As malware may detect the emulated environment using emulation bugs in the emulator, some efforts have been made to detect bugs in the CPU emulators and thus can improve emulation accuracy [28, 29].

More troubling are the intrinsic differences between the emulated environment and mobile systems. Mobile devices contain numerous sensors, e.g. GPS, motion and audio, with performance profiles which might be difficult to emulate. While exploring multiple execution paths may be used to bypass these types of tests, they might still not be sufficient. For example we have observed that Android, as an interactive platform, can be sensitive to the performance overhead due to analysis. If the anal-

ysis takes too long, certain timeout events are triggered leading to different execution paths. The analyst must be aware of these new challenges. In summary, further investigation in this area is needed.

8 Related Work

Virtual Machine Introspection Virtual Machine Introspection is a family of techniques that rebuild a guest’s context from the virtual machine monitor [21, 24]. This is achieved by understanding the important kernel data structures (such as the task list) and extracting important information from these data structures. For closed-source operating systems, it is difficult to have complete understanding of the kernel data structures. To solve this problem, Dolan-Gavitt et al. developed a technique that automatically generates introspection tools by first monitoring the execution of a similar tool within the guest system and then mimicking the same execution outside of the guest system [16]. With deep understanding of the Android kernel, DroidScope is able to intercept certain kernel functions and traverse proper kernel data structures to reconstruct the OS level view. In comparison, DroidScope takes it one step further to reconstruct the Dalvik/Java view, such that both Java and native components from an App can be analyzed simultaneously and seamlessly.

Dynamic Binary Instrumentation PIN [27], DynamoRIO [5], and Valgrind [32] are powerful dynamic instrumentation tools that analyze user-level programs. They are less ideal for malware analysis, because they share the same memory space with user-level malware and thus can be subverted. Bernat et al. used a formal model to identify observable differences due to instrumentation of *sensitive* instructions and created a *sensitivity-resistant* instrumentation tool called SR-Dyninst [4]. Like the other tools though, it cannot be used to analyze kernel-level malware.

Anubis [1], PinOS [7], TEMU [35], and Ether [15] are based on CPU emulators and hypervisors. They have the full system view of the guest system and thus are better suited for malware analysis. These systems only support the x86 architecture and Ether, in principle, cannot support ARM, because it relies on the hardware virtualization technology on x86. A new port must be developed for ARM virtualization [30]. While Atom based mobile platforms are available, ARM still dominates the Android market and thus ARM based analysis is important. To the best of our knowledge, DroidScope is the first fine-grained dynamic binary instrumentation framework that supports the ARM architecture and provides a comprehensive interface for Android malware analysis. We do not however support control flow tainting or different tainting profiles like Dytan [10]. Since Dytan is

based on PIN, it is theoretically feasible to port the tool to PIN for ARM [23], although it will still be limited to analyzing user-level malware.

Dalvik Analysis Tools Enck et al. used *ded* to convert Dalvik bytecode into Java bytecode and *soot* to further convert it into Java source code to identify data flow violations [20]. While powerful, the authors note that some violations could not be identified due to code recovery failures. DroidRanger is a static analysis tool that operates on Dalvik bytecode directly and was successful in identifying previously unknown malicious Apps in Android marketplaces [41]. TaintDroid and DroidBox are two examples of dynamic analysis tools for Android applications [17, 19]. TaintDroid is a specially crafted DVM that supports taint analysis of Dalvik instructions and across API calls. DroidBox is a project that uses TaintDroid to build an android application sandbox for analysis purposes. The biggest advantage of using TaintDroid is that it runs on actual devices. All of the hardware, sensors, vendor software and unpredictable intricacies that come with a real device are there. This can't be achieved in an emulated environment. The major negative of all these tools is that they are limited to analyzing the Java portion of Apps. Thus, if there is a native component, like DroidKungFu has, they will not be able to fully analyze it.

9 Conclusion

We presented DroidScope, a fine grained dynamic binary instrumentation tool for Android that rebuilds two levels of semantic information: operating system and Java. This information is provided to the user in a unified interface to enable dynamic instrumentation of both the Dalvik bytecode as well as native instructions. In this manner, the analyst is able to reveal the behavior of a malware sample's Java and native components as well as interactions between them and the rest of the system as evidenced by the successful analysis of DroidKungFu and DroidDream using DroidScope. These capabilities are provided to the analyst without changing the guest Android system and particularly with JIT intact. Our performance evaluation showed the benefits of dynamically disabling JIT for targeted analysis such as API tracing. The overall performance seems reasonable as well.

Acknowledgements

We thank the anonymous reviewers for their insightful comments towards improving this paper. This work is supported in part by the US National Science Foundation NSF under Grants #1018217 and #1054605. Any opinions, findings, and conclusions made in this material are those of the authors and do not necessarily reflect the views of the NSF or the Air Force Research Laboratory.

References

- [1] Anubis: Analyzing Unknown Binaries. <http://anubis.isecclab.org/>.
- [2] BALZAROTTI, D., COVA, M., KARLBERGER, C., KRUEGEL, C., KIRDA, E., AND VIGNA, G. Efficient Detection of Split Personalities in Malware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, February 2010).
- [3] BELLARD, F. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track* (April 2005).
- [4] BERNAT, A. R., ROUNDY, K., AND MILLER, B. P. Efficient, sensitivity resistant binary instrumentation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2011), ISSTA '11, ACM, pp. 89–99.
- [5] BRUENING, D., GARNETT, T., AND AMARASINGHE, S. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization (CGO'03)* (March 2003).
- [6] BRUMLEY, D., HARTWIG, C., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SONG, D. BitScope: Automatically dissecting malicious binaries. Tech. Rep. CS-07-133, School of Computer Science, Carnegie Mellon University, Mar. 2007.
- [7] BUNGALE, P. P., AND LUK, C.-K. PinOS: a programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd international conference on Virtual execution environments* (2007), VEE '07, pp. 137–147.
- [8] CHENG, B., AND BUZBEE, B. A JIT compiler for android's dalvik VM. <http://www.google.com/events/io/2010/sessions/jit-compiler-androids-dalvik-vm.html>, 2010. Google I/O.
- [9] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Mar. 2011).
- [10] CLAUSE, J., LI, W., AND ORSO, A. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA'07)* (2007), pp. 196–206.
- [11] CRANDALL, J. R., AND CHONG, F. T. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO'04)* (December 2004).
- [12] Cve-2009-1185. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1185>.
- [13] ded: Decompiling Android Applications. <http://siis.cse.psu.edu/ded/index.html>.
- [14] Dynamic, metamorphic (and opensource) virtual machines. http://archive.hack.lu/2010/Desnos_Dynamic_Metamorphic_Virtual_Machines-slides.pdf.
- [15] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (2008), pp. 51–62.
- [16] DOLAN-GAVITT, B., LEEK, T., ZHIVICH, M., GIFFIN, J., AND LEE, W. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2011), SP '11, IEEE Computer Society, pp. 297–312.

- [17] Droidbox: Android application sandbox. <http://code.google.com/p/droidbox/>.
- [18] EGELE, M., KRUEGEL, C., KIRDA, E., YIN, H., AND SONG, D. Dynamic Spyware Analysis. In *Proceedings of the 2007 Usenix Annual Conference (Usenix'07)* (June 2007).
- [19] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–6.
- [20] ENCK, W., OCTEAU, D., MCDANIEL, P., AND CHAUDHURI, S. A study of android application security. In *Proceedings of the 20th USENIX Security Symposium* (2011).
- [21] GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of Network and Distributed Systems Security Symposium (NDSS'03)* (February 2003).
- [22] Gartner says sales of mobile devices grew 5.6 percent in third quarter of 2011; smartphone sales increased 42 percent. <http://gartner.com/it/page.jsp?id=1848514>, 2011.
- [23] HAZELWOOD, K., AND KLAUSER, A. A dynamic binary instrumentation engine for the arm architecture. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems* (New York, NY, USA, 2006), CASES '06, ACM, pp. 261–270.
- [24] JIANG, X., WANG, X., AND XU, D. Stealthy malware detection through vmm-based “out-of-the-box” semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS'07)* (October 2007).
- [25] Security alert: New sophisticated android malware droidkungfu found in alternative chinese app markets. <http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html>.
- [26] KANG, M. G., YIN, H., HANNA, S., MCCAMANT, S., AND SONG, D. Emulating emulation-resistant malware. In *Proceedings of the 2nd Workshop on Virtual Machine Security (VM-Sec'09)* (November 2009).
- [27] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of 2005 Programming Language Design and Implementation (PLDI) conference* (june 2005).
- [28] MARTIGNONI, L., MCCAMANT, S., POOSANKAM, P., SONG, D., AND MANIATIS, P. Path-exploration lifting: Hi-fi tests for lo-fi emulators. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (London, UK, Mar. 2012).
- [29] MARTIGNONI, L., PALEARI, R., ROGLIA, G. F., AND BRUSCHI, D. Testing cpu emulators. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09)* (2009), pp. 261–272.
- [30] MIJAR, R., AND NIGHTINGALE, A. Virtualization is coming to a platform near you. Tech. rep., ARM Limited, 2011.
- [31] MOSER, A., KRUEGEL, C., AND KIRDA, E. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (Oakland'07)* (May 2007).
- [32] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI* (2007), pp. 89–100.
- [33] PORTOKALIDIS, G., SLOWINSKA, A., AND BOS, H. Argos: an emulator for fingerprinting zero-day attacks. In *EuroSys 2006* (April 2006).
- [34] Proguard. <http://proguard.sourceforge.net>.
- [35] TEMU: The BitBlaze dynamic analysis component. <http://bitblaze.cs.berkeley.edu/temu.html>.
- [36] YAN, L.-K., JAYACHANDRA, M., ZHANG, M., AND YIN, H. V2E: Combining hardware virtualization and software emulation for transparent and extensible malware analysis. In *Proceedings of the Eighth Annual International Conference on Virtual Execution Environments (VEE'12)* (March 2012).
- [37] YIN, H., LIANG, Z., AND SONG, D. HookFinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)* (February 2008).
- [38] YIN, H., AND SONG, D. Temu: Binary code analysis via whole-system layered annotative execution. Tech. Rep. UCB/EECS-2010-3, EECS Department, University of California, Berkeley, Jan 2010.
- [39] YIN, H., SONG, D., MANUEL, E., KRUEGEL, C., AND KIRDA, E. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)* (October 2007).
- [40] ZHOU, Y., AND JIANG, X. Dissecting android malware: Characterization and evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland 2012)* (San Francisco, CA, USA, May 2012), IEEE.
- [41] ZHOU, Y., WANG, Z., ZHOU, W., AND JIANG, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the 19th Network and Distributed System Security Symposium* (San Diego, CA, February 2012).

A Trace-Based Exploit Diagnosis of “ratc”

In this section, we provide an example of exploit diagnosis using DroidScope and the ARM instruction tracer on “ratc”. These results corroborate with publicly available information on “ratc” and the setuid exhaustion vulnerability.

We know that *adb* is supposed to downgrade its privileges by setting its uid to AID_SHELL (2000), and yet *adb* retained its root privileges after the attack. Thus, in an effort to identify the root cause of the vulnerability, we used DroidScope to gather an ARM instruction trace that includes both user and kernel code.

A simplified and annotated log is shown in Figure 13. In the log, the instruction’s address comes first followed by a colon, the decoded instruction and then the operands. We have also indented the instructions to illustrate the relative stack depth.

The log begins when *setgid* returns from the kernel space and returns back to *adb.main* at address 0x0000c3a4. Almost immediately, the log shows *setuid* being called. After transitioning into kernel mode, we see *sys.setuid* being called followed by a call to *set.user*. Later we see *set.user* returning an error code 0xfffff5 which is (-11 in 2’s complement or -EAGAIN).

Tracing backwards in the log reveals that this error code was the result of the *RLIMIT_NPROC* check in *set.user*. This reveals why *setuid* failed to downgrade

```

;;;setgid returns from kernel back to adbd
0000813c: pop {r4, r7}
00008140: movs r0, r0
00008144: bxpl lr : Read Oper[0]. R14, Val = 0xc3a5
;; Return back to 0xc3a4 (caller) in Thumb mode

;;;adbd_main sets up for setuid
0000c3a4: movs r0, #250
0000c3a6: lsls r0, r0, #3 : Write Oper[0]. R0, Val = 0x7d0
;; 250 * 8 = 0x7d0 = 2000 = AID_SHELL

...

;;;Start of setuid section
;;; 213 is syscall number for sys_setuid
00008be0: push {r4, r7} : Write Oper[0]. M@be910bb8, Val = 0x7d0
;; push AID_SHELL onto the stack
00008be4: mov r7, #213
00008be8: svc 0x00000000
;; Make sys call

;;; === TRANSITION TO KERNEL SPACE ===

;;;sys_setuid then calls set_user in kernel mode

;;;inside sys_setuid
;; Has rlimit been reached?
c0048944: cmp r2, r3 : Read Oper[0]. R3, Val = 300 Read Oper[1]. R2, Val = 300

;;; RLIMIT(300) is reached and !init_user so return -11
c0048960: mvn r0, #10 : Write Oper[0]. R0, Val = 0xffffffff5
;; the return value is now -11 or -EAGAIN
c0048964: ldmib sp, {r4, r5, r6, fp, sp, pc}

;;;Return back to sys_setuid which returns back to userspace

;;; === RETURN TO USERSPACE ===

;;;setuid continues
00008bec: pop {r4, r7}
00008bf0: movs r0, r0 : Read Oper[0]. R0, Val = 0xffffffff5
;; -11 is still here

;;;Return back to adb_main at 0xc3ac (the return address) above
;;; Immediately starts other work, does not check return code
0000c3ac: ldr r7, [pc, #356] : Read Oper[0]. M@0000c514, Val = 0x19980330
Write Oper[0]. R7, Val = 0x19980330
;; 0x19980330 is _LINUX_CAPABILITY_VERSION

```

Figure 13: Annotated adbd trace

adbd's privileges. Further analysis of the log shows that the return value from *setuid* was not used by adbd nor was a call to *getuid* seen. The same applies to *setgid*. This indicates that adbd failed to ensure that it is no longer running as root. Thus, our analysis shows that the vulnerability is due to two factors, RLIMIT_NPROC and failure to check the return code by adbd.

STING: Finding Name Resolution Vulnerabilities in Programs

Hayawardh Vijayakumar, Joshua Schiffman and Trent Jaeger
*Systems and Internet Infrastructure Security Laboratory,
Department of Computer Science and Engineering,
The Pennsylvania State University*
{hvijay, jschiffm, tjaeger}@cse.psu.edu

Abstract

The process of *name resolution*, where names are resolved into resource references, is fundamental to computer science, but its use has resulted in several classes of vulnerabilities. These vulnerabilities are difficult for programmers to eliminate because their cause is external to the program: the adversary changes namespace bindings in the system to redirect victim programs to a resource of the adversary's choosing. Researchers have also found that these attacks are very difficult to prevent systematically. Any successful defense must have both knowledge about the system namespace and the program intent to eradicate such attacks. As a result, finding and fixing program vulnerabilities to such as attacks is our best defense. In this paper, we propose the STING test engine, which finds name resolution vulnerabilities in programs by performing a dynamic analysis of name resolution processing to produce directed test cases whenever an attack may be possible. The key insight is that such name resolution attacks are possible whenever an adversary has write access to a directory shared with the victim, so STING automatically identifies when such directories will be accessed in name resolution to produce test cases that are likely to indicate a true vulnerability if undefended. Using STING, we found 21 previously-unknown vulnerabilities in a variety of Linux programs on Ubuntu and Fedora systems, demonstrating that comprehensive testing for name resolution vulnerabilities is practical.

1 Introduction

The association between names and resources is fundamental to computer science. Using names frees computer programmers from working with physical references to resources, allowing the system to store resources in the way that it sees fit, and enables easy sharing of resources, where different programs may use the different names for

the same object. When a program needs access to a resource, it presents a name to a name server, which uses a mechanism called *name resolution* to obtain the corresponding resource.

While name resolution simplifies programming in many ways, its use has also resulted in several types of vulnerabilities that have proven difficult to eliminate. Adversaries may control inputs to the name resolution process, such as namespace bindings, which they can use to redirect victims to resources of the adversaries' choosing. Programmers often fail to prevent such attacks because they fail to validate names correctly, follow adversary-supplied namespace bindings, or lack insight into which resources are accessible to adversaries. Table 1 lists some of the key classes of these vulnerabilities.

While a variety of system defenses for these attacks have been proposed, particularly for name resolution attacks based on race conditions [14, 20, 22, 39, 40, 48, 50–52, 57], researchers have found that such defenses are fundamentally limited by a lack of knowledge about the program [12]. Thus, the programmers' challenge is to find such vulnerabilities before adversaries do. However, finding such vulnerabilities is difficult because the vectors for name resolution attacks are outside the program. Table 1 shows that adversaries may control namespace bindings to redirect victims to privileged resources of their choice, using what we call *improper binding attacks* or redirect victims to resources under the adversaries' control, using what we call *improper resource attacks*. Further, both kinds of attacks may leverage the non-atomicity of various system calls to create races, such as the time-of-check-to-time-of-use (TOCTTOU) attacks [6, 38], which makes them even more difficult for victims to detect.

Researchers have explored the application of dynamic and static analysis to detect namespace resolution attacks. Dynamic analyses [1, 32, 35, 36, 56] log observed system calls to detect possible problems, such as check-

Attack	CWE ID
Improper Binding Attacks	
UNIX Symlink Following	CWE-61
UNIX Hard Link Following	CWE-62
Improper Resource Attacks	
Resource Squatting	CWE-283
Untrusted Search Path	CWE-426
Attacks Caused by Either Bindings or Resources	
TOCTTOU Race Condition	CWE-362

Table 1: Classes of name resolution attacks.

use pairs that may be used in TOCTTOU attacks. However, the existence of problems does not necessarily mean that the program is vulnerable. Many of the check-use pairs found were not exploitable. Static analyses use syntactic analyses [6, 53] and/or semantic models of programs to check for security errors [15, 44], sometimes focusing on race conditions [27]. These static analyses do not model the system environment, however, so they often produce many false positives. In addition, several of these analyses result in false negatives as they rely on simpler models of program behavior (e.g., finite state machines), limited alias analysis, and/or manual annotations.

The key insight is that such name resolution attacks are possible only when an adversary has write access to a directory shared with the victim. Using this write access, adversaries can plant files with names used by victims or create bindings to redirect the victim to files of the adversaries' choice. Chari *et al.* [14] demonstrated that when victims use such bindings and files planted by adversaries attacks are possible, so they built a system mechanism to authorize the bindings used in name resolution. However, we find that only a small percentage of name resolutions are really accessible to adversaries and most of those are defended by programs. Further, the solution proposed by Chari *et al.* is prone to false positives, as any pure system solution is, because it lacks information about the programs' expected behaviors [12]. Instead, we propose to test programs for name resolution vulnerabilities by having *the system assume the role of an adversary*, performing modifications that an adversary is capable of, at runtime. Using the access control policy and a list of adversarial subjects, the system can determine whether an adversary has write access to a directory to be used in a name resolution. If so, the system prepares an attack as that adversary would and detect whether the program was exploited or immune to the attack (e.g., did the program follow the symbolic link created?). This is akin to directed black-box testing [23], where a program is injected with a dictionary of commonly known attacker inputs.

In this paper, we design and implement the STING test

engine, which finds name resolution vulnerabilities in programs by performing a dynamic analysis of name resolution processing to produce directed test cases whenever an attack may be possible. STING is an extension to a Linux Security Module [58] that implements the additional methods described above to provide comprehensive, system-wide testing for name resolution vulnerabilities. Using STING, we found 21 previously-unknown name resolution vulnerabilities in 19 different programs, ranging from startup scripts to mature programs, such as cups, to relatively new programs, such as x2go. We detail several bugs to demonstrate the subtle cases that can be found using STING. Tests were done on Ubuntu and Fedora systems, where interestingly some bugs only appeared on one of the two systems because of differences in the access control policies that implied different adversary access.

This research makes the following novel contributions:

- We find that name resolution attacks are always possible whenever a victim resolves a name using a directory where its adversaries have permission to create files and/or links, as defined in Section 3. If a victim uses such a directory in resolving a name, an adversary may redirect them to a resource of the adversary's choosing, compromising victims that use such resources unwittingly.
- We develop a method for generating directed test cases automatically that uses a dynamic analysis to detect when an adversary could redirect a name resolution in Section 4.1.
- We develop a method for system-wide test case processing that detects where victims are vulnerable to name resolution attacks, restores program state to continue testing, and manages the testing coverage in Section 4.2.
- We implement a prototype system STING for Linux 3.2.0 kernel, and run STING on the current versions of Linux distributions, discovering 21 previously-unknown name resolution vulnerabilities in 13 different programs. Perhaps even more importantly, STING finds that 90% of adversary-accessible name resolutions are defended by programs correctly, eliminating many false positives.

We envision that STING could be integrated into system distribution testing to find programs that do not effectively defend themselves from name resolution attacks given that distribution's access control policy before releasing that distribution to the community of users.

2 Problem Definition

Processes frequently require system level resources like files, libraries, and sockets. Since the system's management of these objects is unknown to the process, names are used as convenient references to the desired resource. A *name resolution* server is responsible for converting the requested resource name to the desired object via a *namespace binding*. Typical namespaces in Unix-based systems include the filesystem and System V IPC namespaces (semaphores, shared memory, message queues, etc.). Some namespaces may even support many-to-one mappings (e.g., multiple pathnames may be linked to the same file inode).

Unfortunately, various *name resolution attacks* are possible when an attacker is able to affect this indirection between the desired resource and its name. In this section, we broadly outline two classes of name resolution attacks and give several instances of them. We then discuss how previous efforts attempt to defend against these attacks and their limitations. Finally, we present our solution, STING, that overcomes many of these shortcomings.

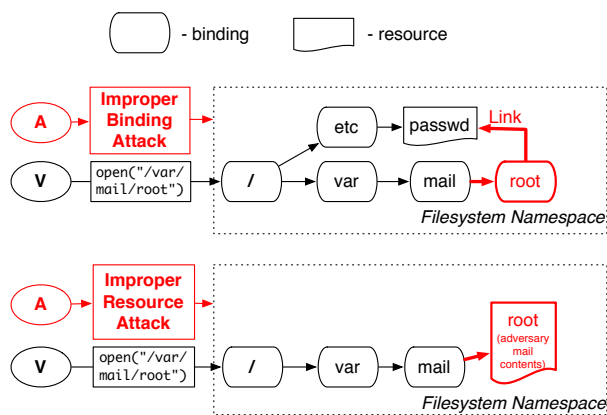


Figure 1: Improper binding and improper resource attacks. A and V are adversary and victim processes respectively.

2.1 Name Resolution Attacks

Malicious parties can control the name resolution process by modifying the namespace's binding to trick victim processes into accessing unintended resources. We find that these attacks can be categorized into two classes. The first, *improper binding attacks*, are when attackers introduce bindings to resources outside of the attackers control. This can give adversary indirect access to the resource through the victim. Such attacks are instances of the confused deputy [33]. The second class, *improper resource attacks*, is when an attacker creates an

unexpected binding to a resource the adversary controls.

Instances of these attacks depend on the namespace. For example, the filesystem namespace is often exploited through malicious path bindings like symbolic links and the creation of files with frequently used names. Consider a mail reader program running as root attempting to check mail from /var/mail/root. Users in the mail group are permitted to place files in this directory for the program to read and send. Figure 1 demonstrates how name resolution attacks from both categories could be performed on this program.

- **Symbolic link following:** The adversary wishes to exfiltrate a protected file (/etc/passwd) that it cannot normally access. Since users in group mail are permitted to create (and delete) bindings (files) in /var/mail, the adversary inserts a symbolic link /var/mail/root in the namespace that is bound to the desired file. If a victim mail program running as root does not check for this link, it might inadvertently leak the protected file. A similar attack can be launched through hard links. This is an instance of an improper binding attack, where adversaries use control of bindings to redirect victim programs with privileges to access or modify resources the adversaries cannot directly.
- **Squatting:** Even if the mail program defends itself against link following attacks, the adversary could simply squat a file on /var/mail. If the mail program accepts this file, the adversary could spoof the contents of mail read by root. This is an example of an improper resource attack, where the adversary uses control of bindings to create a resource under her control when the victim does not expect to interact with the adversary.
- **Untrusted search path:** Programs frequently rely on files like system libraries or configuration files, but the names they supply to access these files may be wrong. One frequent cause is the program supplying a name relative to its working directory, which causes a problem if the working directory is adversary controlled. Adversaries can then simply bind arbitrary resources at these filenames, possibly gaining control of the victim's program. This is another instance of an improper resource attack, where the adversary supplies an improper resource to the victim.

While the attacks an adversary can carry out are well known, the ways in which programs defend themselves are often ad hoc and complex [13]. Even the most diligent programs may fail to catch all the ways in which an adversary might manipulate these namespaces.

Moreover, defenses to these attacks can often be circumvented through *time-of-check-to-time-of-use* (TOCTTOU) attacks. To do this, the adversary waits until the mail program checks that `/var/mail/root` is a regular file prior to opening it and then switches the file to a link before the open call is made. Given the variety of possible name resolution attacks and the complex code needed to defend against them, it should come as little surprise that vulnerabilities of this type continue to be uncovered. Such attacks contribute 5-10% of CVE entries each year.

2.2 Detecting Name Resolution Attacks

Researchers have explored a variety of dynamic and static analyses to detect instances of name resolution attacks, particularly TOCTTOU attacks. However, all such analyses are limited in some ways when applied to the problem of detecting name resolution attacks.

Static Analysis Static analyses of TOCTTOU attacks vary from syntactic analyses specific to *check-use* pairs [6, 53], to building various models of programs to check for security errors [15, 44, 45], to race conditions in general [27]. However, static analyses are disconnected from essential environmental information, such as the system's access control policy to determine whether an adversary can even launch an attack. For example, a program may legitimately access files in `/proc` without checking for name resolution attacks; however, the same cannot be done in `/tmp`. Thus, these analyses yield a significant number of false positives. Further, static techniques are limited to TOCTTOU attacks, due to the absence of standardized program checks against name resolution attacks in general.

Dynamic Analysis Dynamic analyses [1, 32, 35, 36, 56] typically take a system-wide view, logging observed system calls from processes to detect possible problems, such as check-use pairs. Dynamic analyses can also detect specific vulnerabilities, either at runtime [36] or after the fact [35]. Compared to static analyses, dynamic analyses can take into account the system's environment, but suffer the disadvantage of being unaware of the internal code of the program. In addition, the quality of dynamic analysis is strongly dependent on the test cases produced. Because name resolution attacks require an active adversary, the problem is to produce adversary actions in a comprehensive manner. Using benign system traces may identify some vulnerabilities, such as those built-in to the normal system configuration [13], but will miss many other feasible attacks. Finally, any dynamic analysis must distinguish program actions that are safe from those that are vulnerable effectively. We have found that programs successfully defend themselves from a large per-

centage of the attempted name resolution attacks (only 12.5% were vulnerable), so test case processing must find cases where program defenses are actually missing or fail. Since previous dynamic analyses lack insight into the program, several false positives have resulted.

Symbolic Execution Researchers have recently had success finding the conditions under which a program is vulnerable using *symbolic execution* [7, 8, 10, 11, 18, 19, 25, 29–31, 46]. Symbolic execution has been used to produce test cases for programs to look for bugs [9, 37], to generate filters automatically [8, 18], and to generate test cases to leverage vulnerabilities in programs [3] automatically. In these symbolic execution analyses, the program is analyzed to find constraints on the input values that lead to a program instruction of interest (i.e., where an error occurs). Then, the symbolic execution engine solves for those constraints to produce a concrete test case that when executed would follow the same path. Finding name resolution attacks using symbolic execution may be difficult because the conditions for attack are determined mainly by the operating environment rather than the program. While symbolic execution often requires a model of the environment in which to examine the program, the environment needs to be the focus of analysis for finding name resolution attacks.

2.3 Our Solution

As a result, we use a dynamic analysis to find name resolution vulnerabilities, but propose four key enhancements to overcome the limitations of prior analyses of all types.

First, each name resolution system call is evaluated at runtime to find the bindings used in resolution and to determine whether an adversary is capable of applying one or more of the attack types listed in Table 1. If so, a test case resource is automatically produced at an adversary-redirected location in the namespace and provided to the victim. As a result, test cases are only applied where adversaries have the access necessary to perform such attacks.

Second, we track the victim's use of the test case resource to determine whether it accepts the resource as legitimate. If the victim uses the resource (e.g., reads from it), we log the program entrypoint¹ that obtained the resource as vulnerable. While it is not always possible to exploit such a flaw to compromise the program, this approach greatly narrows the number of false positives while still locating several previously-unknown true

¹A program entrypoint is a program instruction that invoked the name resolution system call, typically indirectly via a library (e.g., `libc`).

vulnerabilities. We also log the test cases run by program endpoint to avoid repeating the same attack.

Third, another problem with dynamic analysis is ensuring that the analysis runs comprehensively even though programs may fail or take countermeasures when attacks are detected. We take steps to keep programs running regardless of whether they fall victim to the attack or not. Our test case resources use the same data as the expected resource to enable vulnerable programs to continue, and we automatically revert namespaces after completion of a test and restart programs that terminate when an attempted attack on them is detected.

3 Testing Model

In this section, we define an adversarial model that we use to generate test cases that can be used to identify program vulnerabilities.

Our goal is to discover vulnerabilities that will compromise the *integrity* of a process. Classically, an integrity violation is said occur when a lower integrity process provides input to a higher integrity process or object [5, 16]. For the name resolution attacks described in the last section (see Table 1), integrity violations are created in two ways: (1) improper binding attacks, where adversaries may redirect name resolutions to resources that are normally not modifiable by adversaries, enabling adversaries to modify higher integrity objects, and (2) improper resource attacks, where adversaries may redirect name resolutions to resources that are under the adversaries' control, enabling adversaries to deliver lower integrity objects to processes. In this section, we define how such attacks are run and detected to identify the requirements for the dynamic analysis.

A nameserver performs namespace resolution by using a sequence of namespace bindings, $b_{ij} = (r_i, n_j, r_k)$, to retrieve resource r_k from resource r_i given a name n_j . In a file system, r_i is a directory, n_j is an element of the name supplied to the nameserver for resolution, and r_k is another directory or a file. Attacks are possible when adversaries of a victim program have access to modify binding b_{ij} to $(r_i, n_j, r_{k'})$ or create such a binding if it does not exist, enabling them to redirect the victim's process to a resource $r_{k'}$ instead of r_k . Since bindings cannot be modified like files, adversaries generally require the delete permission to remove the old binding and the create permission to create the desired binding to perform such modification. Two types of name resolution attacks are possible when adversaries have such permission (e.g., write permission to directories in UNIX systems).

Improper binding attacks use the permission to modify a binding to create a link (symbolic or hard) to an existing resource that is inaccessible to the adversary, as

in symbolic and hard link attacks described above. That is, the improper binding may lead to privilege escalation for the adversary by redirecting the victim process to use an existing resource on behalf of that adversary.

Improper resource attacks use the permission to modify a binding to create a new resource controlled by the adversary. That is, the adversary tries to trick the victim into using the improper resource to enable the adversary to provide malicious input to the victim, such as in resource squatting and untrusted search path attacks described above.

STING discovers name resolution vulnerabilities by identifying scenarios where an attack is possible and generating test cases to validate the vulnerability. Whenever a *name resolution system call* is requested by the victim (i.e., a system call that converts a name to a resource reference, such as `open`), STING finds the bindings that would be used in the namespace resolution process to determine whether an adversary of the process has access to modify one or more of these bindings. If so, STING generates an *attack test case* by producing a test case resource, which emulates either an existing, privileged resource or a new adversary-controlled resource, and adjusting the bindings as necessary to resolve to that resource. A reference to this test case resource is returned to the victim.

Vulnerability in the Victim. We define a victim to be vulnerable if the victim runs an *accept system call* using a reference to the test case resource.

A victim accepts a test case resource if it runs an *accept system call*, a system call that uses the returned reference to the test case resource to access resource data (e.g., `read` or `write`). If a victim is tricked into reading or writing a resource inaccessible to the adversary, the adversary can modify the resource illicitly². If a victim is tricked into reading or writing a resource that is controlled by the adversary, then the adversary can control the victim's processing.

4 Design

The design of STING is shown in Figure 2. STING is divided into two phases. The *attack phase* of STING is invoked at the start of a name resolution system call. STING resolves the name itself to obtain the bindings that would be used in normal resolution, and then determines whether an attack is possible using the program's adversary model. When an attack is possible, STING chooses an attack from the list in Table 1 that has not already been tried and produces a test case resource and asso-

²A read operation on a test case resource is indicative of integrity problems if the resource is opened with read-write access.

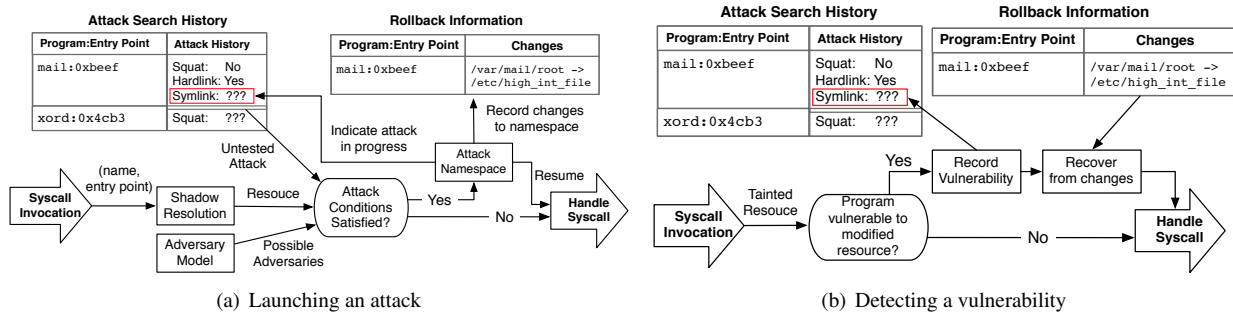


Figure 2: STING consists of two phases: (a) launching an attack, and (b) detecting a vulnerability due to the attack and recovering original namespace state.

ciated bindings to launch the attack. The *detect* phase of STING is invoked on accept system calls. This phase detects if a process “accepted” STING’s changes, indicating a vulnerability, and records the vulnerability information in the previously added entry in the attack history. STING reverts the namespace to avoid side-effects. These two phases are detailed below.

STING is designed to test systems, not just individual programs, so STING will generate test cases for any program in the system that has an adversary model should the opportunity arise. To control the environment under which a program is run, STING intercepts `execve` system calls. For example, programs that may be run by unprivileged users (e.g., `setuid` programs) are started in an adversary’s home directory by this interception code. Other than initialization, the attack and detect phases are the same for all processes.

4.1 Attack Phase

Shadow resolution. Whenever a name resolution system call is performed, STING needs identify whether an attack is possible against that system call. The first step is to collect the bindings that would normally be used in the resolution. We cannot use the existing name resolution mechanism, however, since that has side-effects that may impact the process and also does not gather the desired bindings for evaluation. Instead, we perform a *shadow resolution* that only collects the bindings.

There are two challenges with shadow resolution. First, we have to ensure that all name resolution operations performed by the system are captured in the shadow resolution. This task can be tricky because some name resolution is performed indirectly. For example, `exec` resolves the interpreter that executes the program in the “shebang” line in addition to the program whose name is an argument to the system call. To capture all the name resolution code, we use `Cscope`³ to find all the system

³<http://cscope.sourceforge.net/>

calls that invoke a fundamental function of name resolution, `do_path_lookup`. Using this we find 62 system calls that do name resolution for the Linux filesystem. The three System V IPC system calls that do name resolution were identified manually.

Second, we need to modify the name resolution code to collect the bindings used without causing side-effects in the system. Fortunately, the name resolution code in Linux does not cause side-effects itself. The system call code that uses name resolution creates the side-effects. Thus, we simply invoke the name resolution functions directly when the system call is received. Some effort must be taken to format the call to the name resolution code at the start of the system call, but fortunately the necessary information is available (name, flags, etc.).

Find vulnerable bindings. To carry out an attack, STING has to determine whether any adversary of the program has the necessary permissions to the bindings involved in the resolution. To answer this question, we need to identify the program’s adversaries and evaluate the permissions these adversaries have to bindings efficiently. We note that the specific permissions necessary to launch an attack are specified in Section 3.

We do not want the dynamic analysis to depend on a single adversary model for the system, but instead permit the use of *program-specific adversary models*. The adversaries of a process are determined by the process’s subject (i.e., in the access control policy) and optional program-specific sets of subjects and/or objects that are adversaries or adversary-controlled, respectively. From this information, a comprehensive set of adversary subjects are computed. Using a discretionary access control (DAC) policy, an adversary is any subject other than the victim and the trusted superuser `root`. Chari *et al.* used the DAC policy in their dynamic analysis [13], which worked adequately for `root` processes but incurred some false positives for processes run under other subjects. For systems that enforce a mandatory access control (MAC) policy, methods have been devised to compute the adver-

saries of individual subjects [34, 49]. We note that MAC adversaries may potentially be running processes under the same DAC user, so they are typically finer-grained.

Finding the permissions of a process's adversaries at runtime must be done efficiently. If a process has several adversaries, the naive technique of querying the access control policy for each adversary in turn is unacceptable. To solve this, we observe we can precompute the adversaries of particular process as in a capability-list, where each process has a list of tuples consisting of an object (or label in a MAC policy), a list of adversaries with create permission to that object (or label), and the list of adversaries with delete permission to that object (or label). We store these in a hash table for quick lookup at runtime.

Identify possible attacks. Once we identify a binding that is accessible to an adversary, we need to choose an attack from which to produce a test case. For improper binding attacks, an attack needs to modify a binding from an existing resource to the target resource using a symbolic or hard link. Such attacks are only possible in the Linux filesystem namespace, where a single file (inode) may have multiple names.

Improper resource attacks are applicable across all namespaces. We consider two instances of improper resource attacks (see Table 1). For resource squatting, attacks are only meaningful if the adversary can supply a resource with a lower integrity than the victim intended to access. To determine the victim's intent, we simply check if a non-adversarial subject has permissions to supply the resource the adversary is attacking⁴. This occurs in directories shared by parties at more than one integrity level. If so, we assume that the victim intended to access the higher integrity file (i.e., one that could be created by a non-adversarial subject), and attempt a squatting attack which succeeds if the victim later accepts the test case resource. MOPS [44] uses a similar but narrower heuristic to identify intent and detect ownership stealing attacks, which are another case of resource squatting attacks.

Launch an attack. Launching an attack involves making modifications to the namespace to generate realistic attack scenarios. Different attacks modify the namespace in different ways. For improper binding attacks, we create a new test case resource (e.g., file) that represents a privileged resource, and change the adversary-modifiable bindings to point to it (e.g., symbolic link). For improper resource attacks, we replace the existing resource (if present) with a new test case resource and binding.

Modification of the filesystem namespace in particular presents challenges of backing up existing files, rollback

⁴We discount `root` superuser permissions while checking non-adversarial subjects, as otherwise `root` will be a non-adversary in any directory.

and multiple views for different subjects. First, we have to change the file system to create the test cases, such as deleting existing files. Second, once the test case finishes, we need to rollback the namespace to its original state. While we can back up files (costing the overhead of copy), other resources such as UNIX domain sockets are hard or impossible to rollback once destroyed. Another requirement is that the attack should only be visible to the appropriate victim subjects having the attacker as an adversary. Thus, direct modification of the existing filesystem is undesirable.

To solve the above problems, we take inspiration from the related problem of filesystem unions. Union filesystems unite two or more filesystems (called branches) together [2, 59]. A common use-case is in Live CD images, where the base filesystem mounted from a CDROM is read-only, and a read-write branch (possibly temporary) is mounted on top to allow changes. When a file on the lower branch is modified, it is "copied up" to the upper branch and thereafter hides the corresponding lower branch file. "Whiteouts" are created on the upper branch for deleted files.

To support STING, our general strategy is thus to have a throw-away "upper branch" mounted on top of the underlying filesystem "lower branch". STING creates resources only on the upper branch. As STING does not deal with data, files are created empty. Next, STING directs operations to upper branches if the resource exists on the upper branch *and* was created by an adversary to the currently running process. This enables different processes to have different views of the filesystem namespace.

Once a test case resource is created, we taint it using extended attributes to identify when it is used in an accept system call⁵, signaling a vulnerability. We also record rollback information about the resources created in a rollback table.

These changes to the bindings have to be done as the adversary. The most straightforward option is to have a separate "adversary process" that is scheduled in to perform needed changes. This was the first option we explored; however, it introduced significant performance degradation due to scheduling. Instead, we perform the change using the currently running victim process itself. We change the credentials of the victim process to that of the adversary, carry out the change, and then revert to the victim's original credentials. We do this without leaving behind side effects on the victim process' state. For example, if we create a namespace binding using `open`, we close the opened file descriptor.

⁵Some filesystems do not support extended attributes. Since we use `tmpfs` as the upper branch, we extended it to add such support for our testing. For other namespaces such as IPCs, we store the taint status in a field in the `security` data structure defined by SELinux.

There are some cases where the system needs to revert a test case resource back to a “benign” version. “Check” system calls [56] (e.g., `stat`, `lstat`) resolve the name to verify properties of the resource, so attacks should present a benign resource to prevent the victim from detecting the attack. We simply redirect such accesses to the lower branch.

In addition to adversarial modification of the namespace, STING also changes process attributes relevant to name resolution. In particular, it changes the working directory a process is launched in to an adversary-controlled directory.

We want to prevent STING from launching the same attack multiple times. Trying the same attack on the same system call prevents forward exploration of the attack space and further program code from being exercised. A unique attack associates an attack type with a particular system call entrypoint in the program. Thus, when we launch an attack, we check an attack history that stores which attack types have been attempted already and their result (see Figure 2). We do not attempt multiple binding changes for an attack type. We have not found any programs that perform different checks for name resolution attacks based on the names used. Tracking such history requires unique identification of system call entrypoints in the program, which we discuss in Recording below.

4.2 Detect Phase

Detect vulnerability. Detecting whether a victim is vulnerable to an attack is relatively straightforward – we simply have to determine if the program “accepted” the test case resource. Definition of acceptance for different attacks are presented in Table 2. On the other hand, we conclude that the program defends itself properly from an attack if it: (1) exits without accepting the test case resource or (2) retries the operation at the same program entrypoint. When detection determines that a victim is vulnerable or invulnerable, it fills this information in the attack history entry created during the attack phase, and optionally logs the fact.

STING detects successful attacks by identifying use of a test case resource. Each test case resource is marked when returned to the victim. To detect when a victim uses a test case resource, we must have access to the inode, so such checking is integrated with the access control mechanism (e.g., Linux Security Module). Once a test case resource is found, we need to determine if it is being accepted by retrieving the system call invoked. As an access control check may apply to multiple system calls, we have to retrieve the identity of the system call from the state of the calling process. Vulnerabilities found have their attack history record logged into user space.

Attack	Accept
Symbolic link	write-like, read, readlink
Hard link	write-like, read
File squat	write-like, read
UNIX-domain socket squat	connect
System V IPC squat	msgsnd, semop, shmat

Table 2: Table showing calls that signify acceptance, and therefore detection, for various attacks. write-like system calls are any calls that modifies the resource’s data or metadata (e.g., `fchmod`).

We note that the process of detecting a vulnerability is the same for all attack types, including those based on race conditions. STING automatically switches resources between check and use as discussed above, so we only need to detect when an untrusted resource is accepted. `fstat` is not an accept system call, so the “use” of the test case resource in that system call does not indicate a vulnerability. Thus, if the program should somehow detect an attack using `fstat`, preventing further use of the test case resource, then STING will not record a vulnerability.

Update attack history. Once a particular attack has been tried on a system call, trying it again in future invocations of the program is redundant and may prevent further code from being exercised. Avoiding this problem requires storing attacks tested for system calls in the attack history. The challenge is unique identification of the system call entrypoint, which uniquely identifies the instruction from which the program made the system call. To find this instruction, we perform a backtrace of the user stack to find the first frame within the program that is not in the libraries. We also extend our system to support interpreters by porting interpreter debugging code into the kernel that locates and parses interpreter data structures to the current line number in the script, for the Bash, PHP and Python interpreters. Only between 11 and 59 lines of code were necessary for each interpreter. We use the current line number in the script as the entrypoint for interpreted programs.

Namespace recovery. Finally, we make changes so that STING can work online despite changing the namespace state. While it appears that such changes could cause processes to crash, we have not found this to be the case. Unlike data fuzzing, we find changes in namespace state do not cause programs to arbitrarily crash, as we preserve data and only change resource metadata. When an attack succeeds, the only change needed is to redirect the access to the corresponding resource in the lower branch of the unioned filesystem that contains the actual data (if one exists), and delete the resource in the upper branch. On the other hand, if the attack fails, STING again deletes the resource in the up-

per branch. Programs that protect themselves proceed in two ways. First, the program might retry the same system call again. Interestingly, we find this happens in a few programs (Section 6.2). In this case, STING will not launch an attack at that entrypoint again, and the program again continues. Second, the program might exit. If so, STING records that the attack failed at that entrypoint and restarts the program with its original arguments (recorded via `execve` interception). For many programs that exit, restarting them from the beginning does not affect system correctness. Thus, we find our tool can perform online without complex logic. We are currently exploring how to integrate process checkpointing and rollback [17] to carry out recovery more gracefully for the exit cases.

5 Implementation

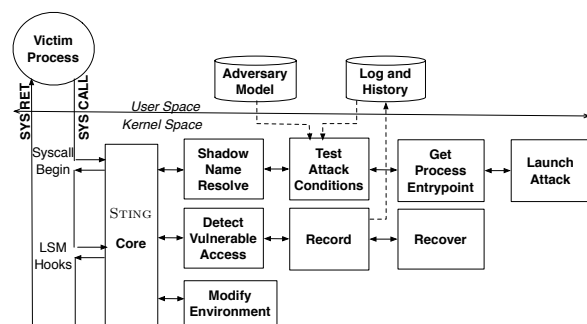


Figure 3: STING is implemented in the Linux kernel and hooks on to the system call beginning and LSM hooks. It has a modular implementation. We show here in particular the interaction between user and kernel space.

Figure 3 shows the overall implementation of STING. STING is implemented in the Linux 3.2.0 kernel. It hooks into LSM through the SELinux function `avc_has_perm` for its detect phase, and into the beginning of system calls for its attack phase. STING has an extensible architecture, with modules registering themselves, and rules specifying the order and conditions under which modules are called.

The modules implement functionality corresponding to the steps shown in the design (Figure 2). The entrypoint module uses the process' `eh_frame` ELF section to perform the user stack unwinding. `eh_frame` replaces the `DWARF debug_frame` section, and is enabled by default on Ubuntu 11.10 and Fedora 15 systems. Stack traces in interpreters yield an entrypoint inside the interpreter, and not the currently executing script. We extended our entrypoint module to identify line numbers inside scripts for the Bash, PHP and Python interpreters [54].

Server Programs Installed	
BIND DNS Server	Apache Web Server
MySQL Database	PHP
Postfix Mail Server	Postgresql Database
Samba File Server	Tomcat Java Server

Table 3: Server programs installed on Ubuntu and Fedora.

The data for the MAC adversary model is pushed into the kernel through a special file, and code to use each of these to decide adversary access is in the test attack conditions module. After launching an attack, the modified resources are tainted through extended attributes for later detection when a victim program uses the resource. We had to extend the `tmpfs` filesystem to handle extended attributes. The recording module can print vulnerabilities as they are detected, and also log the vulnerabilities and search history into userspace files through `relayfs`. A startup script loads the attack search history into the kernel during bootup, so the same attacks are not tried again.

We prototyped a modified version of the UnionFS filesystem [59] for STING. We mount a `tmpfs` as the upper branch, and the root filesystem as the lower branch. The main change involved redirecting to the upper or lower branches depending on a subject's adversaries, and disabling irrelevant UnionFS features such as copy-up.

6 Evaluation

We first evaluate STING's ability to finding bugs, as well as broader security issues in Section 6.1. We then analyze the suitability of STING as an online testing tool in Section 6.2

6.1 Security Evaluation

The aim of the security evaluation is to show that:

- STING can detect real vulnerabilities with a high percentage of them being exploitable in both newer programs, and older, more mature programs, and
- Normal runtime and static analysis would result in a large number of false positives, and normal runtime would also miss some attacks.

We tested STING on the latest available versions of two popular distributions - Ubuntu 11.10 and Fedora 16. In both cases, we installed the default base Desktop distribution, and augmented them with various common server programs (Table 3). Note that STING requires no additional special setup; it simply reacts to normal name resolution requests at runtime. We collected data on both systems over a few days of normal usage.

Adversary model	Total Resolutions	Adversary Access	Vulnerable
DAC - Ubuntu	2345	134 (5.7%)	21 (0.9%)
DAC - Fedora	1654	66 (4%)	5 (0.3%)

Table 4: Table showing the total number of distinct entrypoints invoking system calls performing namespace resolutions, number accessible to adversaries under an adversary model, and number of interfaces for which STING detected vulnerabilities.

6.1.1 Finding Vulnerabilities

Using a DAC attacker model, in total, STING found 26 distinct vulnerable resolutions across 20 distinct programs (including scripts). Of the 26 vulnerable resolutions, 5 correspond to problems already known but un-fixed. 17 of these vulnerabilities are *latent* [13], meaning a normal local user would have to gain privileges of some other user and can then attempt further attacks. For example, one bug we found required the privileges of the user `postgres` to carry out a further attack on `root`. This can be achieved, for example, by remote network attackers compromising the PostgreSQL daemon. For all vulnerabilities found, we manually verified the source code that a name resolution vulnerability existed. Several bugs were reported, of which 2 were deemed not exploitable (although a name resolution vulnerability existed) (Section 6.1.3).

Table 4 shows the total number of distinct name resolutions received by STING that were attackable. This data shows challenges facing static and normal runtime analysis. Only 4-5.7% of the total name resolutions are accessible to the adversary under the DAC adversary model. Therefore, static analysis that looks at the program alone will have a large number of false positives, because programs do not have to protect themselves from name resolutions inaccessible to the adversary. Second, normal runtime analysis cannot differentiate between when programs are vulnerable and when they protect themselves appropriately. We found only 7.5-15.6% of the name resolutions accessible to the adversary are actually vulnerable to different name resolution attacks. Further, 6 of these vulnerabilities would simply not have been uncovered during normal runtime; they are untrusted search paths that require programs to be launched in insecure directories.

Table 5 shows the total number of vulnerabilities by type. A single entrypoint may be vulnerable to more than one type of attack. We note that STING was able to find vulnerabilities of all types, including 7 that required race conditions.

Table 6 shows the various programs across which vulnerabilities were found. Interestingly, we note that 6 of the 24 vulnerable name resolutions in Ubuntu were found in Ubuntu-specific scripts. For example, CVE-

Type of vulnerability	Total
Symlink following	22
Hardlink following	14
File squatting	10
Untrusted search	6
Race conditions	7

Table 5: Number and types of vulnerabilities we found. Race is the number of TOCTTOU vulnerabilities, where a check is made but the use is improper. A single entrypoint in Table 6 may be vulnerable to more than one kind of attack.

Program	Vuln. Entry	Priv. Escalation DAC: uid->uid	Distribution	Previously known
dbus-daemon	2	messagebus->root	Ubuntu	Unknown
landscape	4	landscape->root	Ubuntu	Unknown
Startup scripts (3)	4	various->root	Ubuntu	Unknown
mysql	2	mysql->root	Ubuntu	1 Known
mysql_upgrade	1	mysql->root	Ubuntu	Unknown
tomcat script	2	tomcat6->root	Ubuntu	Known
lightdm	1	*->root	Ubuntu	Unknown
bluetooth-applet	1	*->user	Ubuntu	Unknown
java (openjdk)	1	*->user	Both	Known
zeitgeist-daemon	1	*->user	Both	Unknown
mountall	1	*->root	Ubuntu	Unknown
mailutils	1	mail->root	Ubuntu	Unknown
bsd-mailx	1	mail->root	Fedora	Unknown
cupsd	1	cups->root	Fedora	Known
abrt-server	1	abrt->root	Fedora	Unknown
yum	1	sync->root	Fedora	Unknown
x2gostartagent	1	*->user	Extra	Unknown
19 Programs	26			21 Unknown

Table 6: Number of vulnerable entrypoints we found in various programs, and the privilege escalation that the bugs would provide.

2011-4406 and CVE-2011-3151 were assigned to two bugs in Ubuntu-specific scripts that STING found. Further, the programs containing vulnerabilities range from mature (e.g., `cupsd`) to new (e.g., `x2go`). We thus believe that STING can help in detecting vulnerabilities before an adversary, if run on test environments before they are deployed.

MAC adversary model. We carried out similar experiments for a MAC adversary model on Fedora 16's default SELinux policy. We assume an adversary limited only by the MAC labels, and allow the adversary permissions to run as the same DAC user. This is one of the aims of SELinux – even if a network daemon running as `root` gets compromised, it should still not compromise the whole system arbitrarily. However, we found that the SELinux policy allowed subjects we consider untrusted (such as the network-facing daemon `sendmail_t`) create permissions to critical labels such as `etc_t`. Thus STING immediately started reporting vulnerable name resolutions whenever any program accessed `/etc`. Thus, either the SELinux policy has to be made stricter, the adversary model must be weakened for mutual trust among all these programs, or all programs have to defend themselves from name resolution attacks in `/etc` (which is probably impractical). This problem is consistent with the findings that `/etc` requires exceptional trust in the SELinux policy reported elsewhere [42].

```

01 /* filename = /var/mail/root */
02 /* First, check if file already exists */
03 fd = open (filename, flg);
04 if (fd == -1) {
05     /* Create the file */
06     fd = open(filename, O_CREAT|O_EXCL);
07     if (fd < 0) {
08         return errno;
09     }
10 }
11 /* We now have a file. Make sure
12 we did not open a symlink. */
13 struct stat fdbuf, filebuf;
14 if (fstat (fd, &fdbuf) == -1)
15     return errno;
16 if (lstat (filename, &filebuf) == -1)
17     return errno;
18 /* Now check if file and fd reference the same file,
19 file only has one link, file is plain file. */
20 if ((fdbuf.st_dev != filebuf.st_dev
21     || fdbuf.st_ino != filebuf.st_ino
22     || fdbuf.st_nlink != 1
23     || filebuf.st_nlink != 1
24     || (fdbuf.st_mode & S_IFMT) != S_IFREG)) {
25     error (_("%s must be a plain file
26 with one link"), filename);
27     close (fd);
28     return EINVAL;
29 }
30 /* If we get here, all checks passed.
31 Start using the file */
32 read(fd, ...)

```

Figure 4: Code from the GNU mail program in mailutils illustrating a squat vulnerability that STING found.

6.1.2 Examples

In this section, we present particular examples highlighting STING’s usefulness, and also broader lessons.

Mail Programs. GNU mail is the default mail client on Ubuntu 11.10, in which STING found a vulnerability. This example shows the difficulty of proper checking in programs, and why detection tools with low false positives are necessary – programmers can easily get such checks wrong, and there are no standardized ways to write code to defend against various name resolution attacks.

The code shows the program preparing to read the file `/var/mail/root`. In summary, this program creates an empty file when the file doesn’t already exist (lines 4-10), using flags (`O_EXCL`) to ensure that a fresh file is created. The program performs several checks to verify the safety of the file opened, guarding against race conditions and link traversal (both symbolic and hard links) (11-29). Unfortunately, the program fails to protect itself against a squatting attack if the file already exists, as it does not check `st_uid` or `st_gid`; any user in group mail can control the contents of `root`’s inbox. Interestingly, it protects itself against squatting attacks on line 6.

X11 script STING found a race condition exploitable by a symbolic link attack on the script that creates `/tmp/.X11-unix` in Ubuntu 11.10. The code snippet is shown in Figure 5. The aim of the script is to cre-

```

01 SOCKET_DIR=/tmp/.X11-unix
...
02 set_up_socket_dir () {
03 if [ "$VERBOSE" != no ]; then
04     log_begin_msg "Setting up X server socket directory"
05 fi
06 if [ -e $SOCKET_DIR ] && [ ! -d $SOCKET_DIR ]; then
07     mv $SOCKET_DIR $SOCKET_DIR.$$
08 fi
09 mkdir -p $SOCKET_DIR
10 chown root:root $SOCKET_DIR
11 chmod 1777 $SOCKET_DIR
12 do_restorecon $SOCKET_DIR
13 [ "$VERBOSE" != no ] && log_end_msg 0 || return 0
14 }

```

Figure 5: Code from an X11 startup script in Ubuntu 11.10 that illustrates a race condition that STING found.

ate a root-owned directory `/tmp/.X11-unix`. Lines 6-8 check if such a file already exists that is not a directory, and if so, moves it away so a directory can be created. In Line 9, the programmer creates the directory, and assumes it will succeed, because the previous code had just moved any file that might exist. However, because `/tmp` is a shared directory, an adversary scheduled in between the moving of the file and the `mkdir` might again create a file in `/tmp/.X11-unix`, thus breaking the programmer’s expectation. If the file is a link pointing to, for example, `/etc/shadow`, the `chmod` on Line 11 will make it world-readable. STING was able to detect this race condition by changing the resource into a symbolic link after the move and before the creation on line 9, as it acts just before the system call on line 9. This script has existed for many years, showing how it is easy to overlook such conditions. This also shows how STING can synchronously produce any race condition an adversary can, because it is in the system. This script was independently fixed by Ubuntu in its latest release. The discussion page for the bug [21] shows how such checks are challenging to get right even for experienced programmers. Consequently, manually scanning source code can also easily miss such vulnerabilities.

mountall This program has an untrusted search path that is not executed in normal runtime but was discovered by STING. This Ubuntu-specific utility simply mounts all filesystems in `/etc/fstab`. When launched in an untrusted directory, it issues mount commands that search for files such as `none` and `fusectl` in the current working directory. If these are symbolic links, then the contents of these files are read through `readlink`, and put in `/etc/mtab`. Thus, the attacker can influence `/etc/mtab` entries and potentially confuse utilities that depend on this file, such as unmounters. This is an example of how very specific conditions are required to detect the attack – the program needs to be launched in an adversarial directory, and the name searched for needs to be a symbolic link. Normal runtime would not give any hint of such attacks.

postgresql init script. This vulnerability highlights the challenge facing developers and OS distributors. This script runs as root, and is vulnerable to symbolic and hard link attacks on accessing files in `/etc/postgresql`. That directory is owned by the user `postgres`, which could be compromised by remote attacks on PostgreSQL, who can then use this vulnerability to gain root privileges. The problem is that the developers who wrote the script did not expect the directory `/etc/postgresql` to be owned by a non-root user. However, the access control policy did not reflect this assumption. STING is useful in finding such discrepancies in access control policies as it can run with attacker models based on different policies.

6.1.3 False Positives

Two issues in STING cause false positives.

Random Name. The programs `yum`, `abrt-server`, `zeitgeist-daemon` in Table 6 were vulnerable to name resolution attacks, but defended themselves by creating files with random names. Library calls such as `mktmp` are used to create such names. STING cannot currently differentiate between “random” and “non-random” names. Exploiting such vulnerabilities requires the adversary to guess the filename, which may be challenging given proper randomness. In any case, such bugs can be fixed by adding the `O_EXCL` flag to `open` when creating files.

Program Internals. STING does not know the internal workings of a program. Thus, it cannot know if use of a resource from a vulnerable name resolution can affect the program or not, and simply marks it for further perusal. A vulnerable name resolution involving a write-like `accept` operation can always be exploited. However, whether those involving `read` can be exploited depends on the internals of the program. Eight of the 26 vulnerable name resolutions in Table 6 are due to `read`. While this has led to some false positives (two additional vulnerable name resolutions involving `read` not in Table 6 were deemed to not affect program functioning), STING narrows the programmers’ effort significantly. Nonetheless, more knowledge regarding program internals would improve the accuracy even further.

6.2 Performance Evaluation

We measured the performance of STING to assess its suitability as an online testing tool. While the performance of STING is of not of primary importance because it is meant to be run on test environments in non-production systems before deployment, it must nevertheless be responsive to online testing. We measured performance using micro- and macro-benchmarks. While

Case	Time (μ s)	Overhead
<i>Attack Phase: open system call</i>		
Base	14.57	–
+ Find Vulnerable Bindings	31.44	2.15 \times
+ Obtain entrypoint and check attack history	211.20	12.33 \times
+ Launch attack	365.87	25.1 \times
<i>Detect Phase: read system call</i>		
Base	8.73	–
+ Detect vulnerability	9.18	1.05 \times
+ Namespace recovery	63.08	7.22 \times

Table 7: Micro-overheads for system calls showing median over 10000 system call runs.

Benchmark	Base	STING	Overhead
Apache 2.2.20 compile	151.65s	163.79s	8%
Apachebench: Throughput	231.77Kbps	221.89Kbps	4.33%
Latency	1.943ms	2.088ms	7.46%

Table 8: Macro-benchmarks showing compilation and Apachebench throughput and latency overhead. The standard deviation was less than 3% of the mean.

STING does cause noticeable overhead, it did not impede our testing. All tests were done on a Dell Optiplex 980 machine with 8GB of RAM.

Micro-performance (Table 7) was measured by the time taken for individual system calls under varying conditions. For an attack launch, system call overhead is caused by the time to (1) detect adversary accessibility, (2) get and compare process entrypoint against attack history, and (3) launch the attack. The main overhead is due to obtaining the entrypoint to check the attack history and carrying out the attack. However, obtaining the entrypoint is required only if the name resolution is adversary accessible (around 4-5.7% in Table 4), and an attack is launched only once for a particular entrypoint, thereby alleviating their impact on overall performance. For the detection phase, we have (1) detect vulnerable access, and (2) rollback namespace. Namespace recovery through rollback is expensive, but occurs only once per attack launched.

Macro-benchmarks (Table 8) showed upto 8% overhead. Apache compilation involved a lot of name resolutions and temporary file creation. During our system tests, the system remained responsive enough to carry out normal tasks, such as browsing the Internet using Firefox and checking e-mail. We are investigating further opportunities to improve performance.

Program retries and restarts. We came across thirteen programs that retried a name resolution system call on failure due to a STING attack test case. The most

common case was temporary file creation – programs retry until they successfully create a temporary file with a random name. Programs that retry integrate well with STING, which maintains its attack history and does not retry the same attacks on the same entrypoints. On the other hand, a few programs exited on encountering an attack by STING. We currently simply restart such programs (Section 4.2). For example, `dbus-daemon` exited during boot due to a STING test case and had to be restarted by STING to continue normal boot. However, programs may lose state across restarts. We are investigating integrating process checkpoint and rollback mechanisms [17].

7 Related Work

Related work that deals with detecting name resolution attacks was presented in Section 2.2. Here, we discuss dynamic techniques to detect other types of program bugs, and revisit some dynamic techniques that detect name resolution attacks.

Black-box testing. Fuzz testing, an instance of black-box testing, runs a program under various input data test cases to see if the program crashes or exhibits undesired behavior. Particular program entrypoints (usually network or file input) are fed totally random input with the hope of locating input filtering vulnerabilities such as buffer overflows [24, 41, 47]. Black-box fuzzing generally does not scale because it is not directed. To alleviate this, techniques use some knowledge of the semantics of data expected at program entrypoints. SNOOZE [4] is a tool to generate fuzzing scenarios for network protocols using which bugs in programs were discovered. TaintScope [55] is a directed fuzzing tool that generates fuzzed input to pass checksum code in programs. Web application vulnerability scanners [23] supply very specific strings to detect XSS and SQL injection attacks.

We find a parallel can be drawn between our approach and directed black-box testing, where semantics of input data is known. While such techniques change the *data* presented to a program to exercise program paths with possible vulnerabilities, we change the resource, or the *metadata* presented to the application for the same purpose. Thus, STING can be viewed as an instance of black-box testing that changes the namespace state to evaluate program responses.

Taint Tracking. Taint techniques track flow of tainted data inside programs. They can be used to find bugs given specifications [60], or can detect secrecy leaks in programs [26]. Flax [43] uses a taint-enhanced fuzzing approach to detect input filtering vulnerabilities in web applications. However, taint tracking by itself does not actively change any data presented to applications, and thus has different applications.

Dynamic Name Resolution Attacks Detection. As mentioned in Section 2.2, most dynamic analysis are specific to detecting TOCTTOU attacks. Chari *et al.* [13] present an approach to defend against improper binding attacks; however, they cannot detect them until they actively occur in the system. We use active adversaries to generate test cases and to exercise potentially vulnerable paths in programs to detect vulnerabilities that would not occur in normal runtime traces. Further, none of the approaches deal with improper resource attacks, of which we detect several.

8 Discussion

Other System State Attacks. More program vulnerabilities may be detected by modifying system state. For example, non-reentrant signal handlers can be detected by delivering signals to a process already handling a signal. Similarly, return values of system calls can be changed to cause conditions suitable for attack (e.g., call to drop privileges fails). While STING could be easily extended to perform these attacks, we believe that these cases are more easily handled through static analysis, as standard techniques are available (e.g., lists of non-reentrant system calls, unchecked return value) through tools such as Fortify [28]. For the reasons we have seen, no such standard techniques are available for name resolution attacks.

Solutions. One of the more effective ways we have seen programs defending against improper binding attacks is by dropping privileges. For example, the privileged part of `sshd` drops privileges to the user whenever accessing user files such as `.ssh/authorized_keys`. Thus, even if code is vulnerable to improper binding attacks, the user cannot escalate privileges.

User directory. Administrators running as `root` should take extreme care when in user-owned directories, as there are several opportunities for privilege escalation. For example, we found during our testing that if the Python interpreter is started in a user-owned directory, Python searches for modules in that directory. If a user has malicious libraries, then they will be loaded instead. More race conditions are also exploitable as a user can delete even `root`-owned files in her directory.

Integration with Black-box Testing. We believe that STING can also integrate with other data fuzzing platforms [41]. Such tools need special environments (e.g., attaching to running processes with debuggers) to carry out their tests on running programs. Instead, we can take input from these platforms and use STING to feed such input into running processes. Since STING also takes into account the access control policy, opportunities to supply adversarial data can readily be located.

Deployment. We envision that STING would be deployed during Alpha and Beta testing of distribution re-

leases. We plan to package STING for distributions, so users can easily install it through the distribution's package managers. STING will test various programs as users are running them, and program vulnerabilities found can be fixed before the final release. Being a runtime analysis tool, STING can possibly find more vulnerabilities as it improves its runtime coverage. Even if a small percentage of users install the tool, we expect a significant increase in the runtime coverage, because different users configure and run programs in different ways.

9 Conclusion

In this paper, we introduced STING, a tool that detects name resolution vulnerabilities in programs by dynamically modifying system state. We examine the deficiencies of current static and normal runtime analysis for detecting name resolution vulnerabilities. We classify name resolution attacks into improper binding and improper resource attacks. STING checks for opportunities to carry out these attacks on victim programs based on an adversary model, and if an adversary exists, launches attacks as an adversary would by modifying namespace state visible to the victim. STING later detects if the victim protected itself from the attack, or it was vulnerable. To allow online operation, we propose mechanisms for rolling back the namespace state. We tested STING on Fedora 15 and Ubuntu 11.10 distributions, finding 21 previously unknown vulnerabilities across various programs. We believe STING shall be useful in detecting name resolution vulnerabilities in programs before attackers. We plan to release and package STING for distributions for this purpose.

References

- [1] A. Aggarwal and P. Jalote. Monitoring the security health of software systems. In *ISSRE-06*, pages 146–158, 2006.
- [2] Aups. <http://aups.sourceforge.net/>.
- [3] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. Aeg: Automatic exploit generation. In *Network and Distributed System Security Symposium*, Feb. 2011.
- [4] G. Banks *et al.*. Snooze: Toward a stateful network protocol fuzzer. In *Lecture Notes in Computer Science*, 2006.
- [5] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE, April 1977.
- [6] M. Bishop and M. Digler. Checking for race conditions in file accesses. *Computer Systems*, 9(2), Spring 1996.
- [7] P. Boonstoppel, C. Cadar, and D. Engler. Rwsset: attacking path explosion in constraint-based test generation. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, 2008.
- [8] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.
- [9] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224, 2008.
- [10] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN*, 2005.
- [11] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2), 2008.
- [12] X. Cai *et al.*. Exploiting Unix File-System Races via Algorithmic Complexity Attacks. In *IEEE SSP '09*, 2009.
- [13] S. Chari and P. Cheng. Bluebox: A policy-driven, host-based intrusion detection system. *ACM Transaction on Information and System Security*, 6:173–200, May 2003.
- [14] S. Chari *et al.* Where do you want to go today? escalating privileges by pathname manipulation. In *NDSS '10*, 2010.
- [15] B. Chess. Improving computer security using extended static checking. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, 2002.
- [16] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. *Security and Privacy, IEEE Symposium on*, 0:184, 1987.
- [17] Container-based checkpoint/restart prototype. <http://lwn.net/Articles/430279/>.
- [18] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *SOSP*, pages 117–130, 2007.
- [19] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005.
- [20] C. Cowan, S. Beattie, C. Wright, and G. Kroah-Hartman. Raceguard: Kernel protection from temporary file race vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Berkeley, CA, USA, 2001. USENIX Association.
- [21] init script x11-common creates directories in insecure manners. <http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=661627>.
- [22] D. Dean and A. Hu. Fixing races for fun and profit. In *Proceedings of the 13th USENIX Security Symposium*, 2004.
- [23] Doup, Adam and Cova, Marco and Vigna, Giovanni. Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners. In *DIMVA*, 2010.
- [24] W. Drewry and T. Ormandy. Flayer: exposing application internals. In *Proceedings of the first USENIX workshop on Offensive Technologies*, 2007.
- [25] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 international symposium on Software testing and analysis, ISSTA '07*, 2007.
- [26] W. Enck *et al.* Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010.
- [27] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.
- [28] Hp fortify static code analyzer (sca). <https://www.fortify.com/products/hpfssc/source-code-analyzer.html>.
- [29] P. Godefroid. Compositional dynamic test generation. *SIGPLAN Not.*, 2007.
- [30] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.
- [31] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated white-box fuzz testing. In *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.

- [32] B. Goyal, S. Sitaraman, and S. Venkatesan. A unified approach to detect binding based race condition attacks. In *International Workshop on Cryptology And Network Security*, 2003.
- [33] N. Hardy. The confused deputy. *Operating Systems Review*, 22:36–38, 1988.
- [34] T. Jaeger, R. Sailer, and X. Zhang. Analyzing Integrity Protection in the SELinux Example Policy. In *Proceedings of the 12th USENIX Security Symp.*, 2003.
- [35] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 91–104, New York, NY, USA, 2005. ACM.
- [36] C. Ko and T. Redmond. Noninterference and intrusion detection. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 177–, Washington, DC, USA, 2002. IEEE Computer Society.
- [37] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, pages 11–11, Berkeley, CA, USA, 2005. USENIX Association.
- [38] W. S. McPhee. Operating system integrity in OS/VS2. *IBM Syst. J.*, 13:230–252, September 1974.
- [39] OpenWall Project - Information security software for open environments. <http://www.openwall.com/>, 2008.
- [40] J. Park, G. Lee, S. Lee, and D.-K. Kim. Rps: An extension of reference monitor to prevent race-attacks. In *PCM (1) 04*, 2004.
- [41] Peach fuzzing platform. <http://peachfuzzer.com/>.
- [42] S. Rueda, D. H. King, and T. Jaeger. Verifying compliance of trusted programs. In *Proceedings of the 17th USENIX Security Symposium*, pages 321–334, Aug. 2008.
- [43] P. Saxena *et al.* Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS*, 2010.
- [44] B. Schwarz, H. Chen, D. Wagner, J. Lin, W. Tu, G. Morrison, and J. West. Model checking an entire linux distribution for security violations. In *Proceedings of the 21st Annual Computer Security Applications Conference*, 2005.
- [45] B. Schwarz, H. Chen, D. Wagner, J. Lin, W. Tu, G. Morrison, and J. West. Model checking an entire linux distribution for security violations. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 13–22, Washington, DC, USA, 2005. IEEE Computer Society.
- [46] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, 2005.
- [47] Sharefuzz. <http://sourceforge.net/projects/sharefuzz/>.
- [48] K. suk Lhee and S. J. Chapin. Detection of file-based race conditions. *Int. J. Inf. Sec.*, 2005.
- [49] W. Sun, R. Sekar, G. Poothia, and T. Karandikar. Practical proactive integrity protection: A basis for malware defense. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, May 2008.
- [50] D. Tsafirir, T. Hertz, D. Wagner, and D. Da Silva. Portably solving file tocttou races with hardness amplification. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 13:1–13:18, Berkeley, CA, USA, 2008. USENIX Association.
- [51] E. Tsyrlkevich and B. Yee. Dynamic detection and prevention of race conditions in file accesses. In *Proceedings of the 12th USENIX Security Symposium*, pages 243–255, 2003.
- [52] P. Uppuluri, U. Joshi, and A. Ray. Preventing race condition attacks on file-systems. In *SAC-05*, 2005.
- [53] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. Its4: A static vulnerability scanner for c and c++ code. In *ACSAC*, 2000.
- [54] H. Vijayakumar, G. Jakka, S. Rueda, J. Schiffman, and T. Jaeger. Integrity Walls: Finding Attack Surfaces from Mandatory Access Control Policies. In *AsiaCCS*, 2012.
- [55] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *IEEE Symposium on Security and Privacy*, 2010.
- [56] J. Wei *et al.* Tocttou vulnerabilities in unix-style file systems: an anatomical study. In *USENIX FAST '05*, 2005.
- [57] J. Wei *et al.* A methodical defense against TOCTTOU attacks: the EDGI approach. In *IEEE International Symp. on Secure Software Engineering (ISSSE)*, 2006.
- [58] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General security support for the Linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, pages 17–31, August 2002.
- [59] C. P. Wright and E. Zadok. Unionfs: Bringing File Systems Together. *Linux Journal*, pages 24–29, December 2004.
- [60] W. Xu, E. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*, pages 121–136, 2006.

Tracking Rootkit Footprints with a Practical Memory Analysis System

Weidong Cui
Microsoft Research
wdcui@microsoft.com

Marcus Peinado
Microsoft Research
marcuspe@microsoft.com

Zhilei Xu
Massachusetts Institute of Technology
timxu@mit.edu

Ellick Chan
University of Illinois at Urbana-Champaign
emchan@illinois.edu

Abstract

In this paper, we present MAS, a practical memory analysis system for identifying a kernel rootkit's memory footprint in an infected system. We also present two large-scale studies of applying MAS to 848 real-world Windows kernel crash dumps and 154,768 potential malware samples.

Error propagation and invalid pointers are two key challenges that stop previous pointer-based memory traversal solutions from effectively and efficiently analyzing real-world systems. MAS uses a new memory traversal algorithm to support error correction and stop error propagation. Our enhanced static analysis allows the MAS memory traversal to avoid error-prone operations and provides it with a reliable partial type assignment.

Our experiments show that MAS was able to analyze all memory snapshots quickly with typical running times between 30 and 160 seconds per snapshot and with near perfect accuracy. Our kernel malware study observes that the malware samples we tested hooked 191 different function pointers in 31 different data structures. With MAS, we were able to determine quickly that 95 out of the 848 crash dumps contained kernel rootkits.

1 Introduction

Kernel rootkits represent a significant threat to computer security because, once a rootkit compromises the OS kernel, it owns the entire software stack which allows it to evade detections and launch many kinds of attacks. For instance, the Alureon rootkit [1] was infamous for stealing passwords and credit card data, running botnets, and causing a large number of Windows systems to crash. Kernel rootkits also present a serious challenge for malware analysis because, to hide its existence, a rootkit attempts to manipulate the kernel code and data of an infected system.

An important task in detecting and analyzing kernel rootkits is to identify all the changes a rootkit makes to an infected OS kernel for hijacking code execution or hiding its activities. We call these changes a rootkit's *memory footprint*. We perform this task in two common scenarios: We detect if real-world computer systems are infected by kernel rootkits. We also analyze suspicious software in a controlled environment. One can use either execution tracing or memory analysis in a controlled environment, but is usually limited to memory analysis for real-world systems. In this paper we focus on the memory analysis approach since it can be applied in both scenarios.

After many years of research on kernel rootkits, we still lack a *practical* memory analysis system that is *accurate*, *robust*, and *performant*. In other words, we expect such a practical system to correctly and quickly identify all memory changes made by a rootkit to arbitrary systems that may have a variety of kernel modules loaded. Furthermore, we lack a large-scale study of kernel rootkit behaviors, partly because there is no practical system that can analyze memory infected by kernel rootkits in an accurate, robust and performant manner.

In this paper, we present MAS, a practical memory analysis system for identifying a rootkit's memory footprint. We also present the results of two large-scale experiments in which we use MAS to analyze 837 kernel crash dumps of real-world systems running Windows 7, and 154,768 potential malware samples from the repository of a major commercial anti-malware vendor. These are the two major contributions of this paper.

Previous work [2, 3, 19] has established that, to identify a rootkit's memory footprint, we need to check not only the integrity of kernel code and static data but also the integrity of dynamic data, and the real challenge lies in the latter task.

In order to locate dynamic data, these systems first locate static data objects in each loaded module, then recursively follow the pointers in these objects and in all

newly identified data objects, until no new data object can be added. Unlike the earlier systems, KOP [3] includes *generic pointers* (e.g., *void**) in its memory traversal, and shows that failing to do so will prevent the memory traversal from reaching about two thirds of the dynamic objects.

Previous solutions do not sufficiently address an important practical problem of this memory traversal procedure: *its tendency to accumulate and propagate errors*. A typical large real-world kernel memory image is bound to contain *invalid pointers*. That is, there are likely to be dynamic objects with pointer fields not pointing to valid objects. Following such pointers results in objects being incorrectly included in the object mapping. Worse, such identification errors can be propagated due to the nature of the recursive, greedy memory traversal. A single incorrectly identified data object may cause many more mistakes in the subsequent traversal.

Invalid pointers may exist for a variety of reasons. For example, an object may have been allocated, but not yet initialized. KOP is exposed to a second source of potential errors. KOP tries to follow all generic pointers. If the pointer type cannot be uniquely determined, KOP tries to decide the correct type using a heuristic. A fraction of these guesses are bound to be incorrect.

In light of these problems, we design MAS to control the number of errors that arise from following invalid pointers and to contain their effects. Instead of performing a greedy memory traversal that is vulnerable to error propagation, MAS uses a new traversal scheme to support error correction. MAS also uses static analysis to derive information that can be used to uniquely identify many objects and their types without having to rely on the recursive traversal procedure. Furthermore, MAS is not subject to errors caused by *ambiguous pointers*, i.e., pointers whose type cannot be uniquely determined. It uses an enhanced static analysis to identify unique types for a large fraction of generic pointers and ignores all remaining ambiguous pointers. While this may reduce coverage, it will never cause an object to be recognized incorrectly. Our evaluation will show that the impact on coverage is minor. Finally, before accepting an object, MAS checks a number of constraints, including new constraints we derive from our static analysis.

We implemented a prototype of MAS and compared it with KOP on eleven crash dumps of real-world systems running Windows Vista SP1. MAS's performance is one order of magnitude better than KOP regarding both static analysis and memory traversal. MAS did not miss or misidentify any function pointers found by KOP, but KOP missed or misidentified up to 40% of suspicious function pointers (i.e., function pointers that point to untrusted code).

In our large-scale experiments, we ran MAS over

crash dumps taken from 837 real-world systems running Windows 7 and memory snapshots taken from Windows XP SP3 VMs subjected to one of 154,768 potential real-world malware samples. For the Windows 7 crash dumps, MAS took 105 seconds to analyze a single dump on average. It identified a total of about 400,000 suspicious function pointers. We were able to verify the correctness of all but 24 of them. Moreover, with the results of MAS, we were able to quickly identify 90 Windows 7 crash dumps (and five Windows Vista SP1 crash dumps) that were infected by kernel rootkits. In our study of malware samples, MAS required about 30 seconds to analyze each VM memory snapshot. Our study shows that the kernel rootkits we tested hooked 191 function pointer fields in 31 data structures. It also shows that many malware samples had identical footprints, which suggests that we can use MAS to detect new malware samples/families that have different memory footprints.

The rest of this paper is organized as follows. Section 2 provides an overview of the paper. Sections 3 and 4 describe the design of MAS and explain the algorithms used for static analysis and memory traversal. Section 5 explains how we evaluate the set of objects found by MAS for suspicious activity. Section 6 describes our implementation of MAS. Section 7 describes our evaluation of MAS. Section 8 and Section 9 describe two large-scale experiments in which we analyze malware samples and identify rootkits from crash dumps. Sections 10 and Section 11 discuss related work and limitations. Finally, Section 12 concludes the paper.

2 Overview

The goal of MAS is to identify all memory changes a rootkit makes for hijacking execution and hiding its activities. MAS does so in three steps: static analysis, memory traversal, and integrity checking.

Static Analysis: MAS takes the source code of the OS kernel and drivers as the input and uses a pointer analysis algorithm to identify candidate types for generic pointers such as *void** and linked list constructs. Furthermore, it also computes the associations between data types and pool tags [18].

Memory Traversal: MAS tries to identify dynamic data objects in a given memory snapshot. Besides the snapshot, the input includes the type related information derived from static analysis and the symbol information [15] for each loaded module (if it is available).

Integrity Checking: MAS identifies the memory changes a rootkit makes by inspecting the integrity of code, static data and dynamic data (recognized

from memory traversal). In addition to checking if some code section is modified, MAS detects two kinds of violations: (1) a function pointer points to a memory region outside of a list of known good modules; (2) a data object is hidden from a system program. The list of identified integrity violations is the final output of MAS. Such information can be used to detect if a system is infected by a rootkit or analyze a rootkit's behavior.

Next we describe these three steps in detail.

3 Static Analysis

In this section, we present our demand-driven pointer analysis algorithm. After that, we describe how we use this algorithm to identify candidate types for generic pointers and data types associated with pool tags.

3.1 Demand-Driven Pointer Analysis

We use *demand-driven* pointer analysis because we do not need the alias information for all the variables in a program, which traditional pointer analyses compute. Instead, we only compute the alias sets of generic pointers, a small portion of all the variables in a program.

Our demand-driven pointer analysis follows largely the approach of Zheng and Rugina [27]. Since our goal is to *precisely* identify candidate types for generic pointers, we extend Zheng and Rugina's pointer analysis to be field-sensitive, context-sensitive and partially flow-sensitive. We achieve partial flow-sensitivity by converting a program to the Static Single Assignment (SSA) form conservatively. We enforce context-sensitivity in a way similar to [23]. We handle indirect calls in our analysis as well.

Next we will summarize the approach of [27] and provide a detailed description of our extension to field sensitivity.

3.1.1 Program Expression Graph

The algorithm of [27] operates on a *Program Expression Graph* (PEG), a graph representation of all expressions and assignments in a C-like program. In this paper, we represent an expression as a C variable with `*` (for the dereference operation), `&` (for the take-address operation) and `→` (for the field operation). In a PEG, the nodes are program expressions, and the edges are of two kinds:

Assignment Edge (A): For each assignment $e_1 = e_2$, there is an *A*-edge from e_2 to e_1 .

Dereference Edge (D): For each dereference $*e$, there is a *D*-edge from e to $*e$; for each address $&e$, there is a *D*-edge from $&e$ to e .

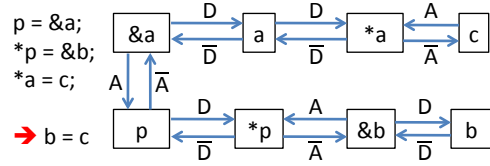


Figure 1: Sample program and its PEG

For each *A* and *D* edge, there is also a corresponding *inverse* edge in the opposite direction, denoted by \bar{A} and \bar{D} . The edges can also be treated as relations between the corresponding nodes; so relations \bar{A} and \bar{D} are the *inverse relations* of *A* and *D*. Figure 1 shows a sample program and its PEG.

3.1.2 CFL-Reachability

In addition to the *A* and *D* relations (edges), we further define two relations between expressions (nodes):

Value Alias (V): If a and b may evaluate to the same value, we say they are value aliases, represented as aVb .

Memory Alias (M): If the addresses of a and b may denote to the same location, we say they are memory aliases, represented as aMb .

Given an interesting expression p , our pointer analysis searches for the set of expressions q such that pVq . We call this set the *value alias set* of p . Similar to [27], we formulate the computation of the *V* relation as a Context-Free Language (CFL) reachability problem [21] over the program expression graph. Specifically, a relation R over the nodes of a PEG can be formulated as a CFL-reachability problem by constructing a grammar G such that a node pair (a, b) has the relation R if and only if there is a path from a to b such that the sequence of labels along the path belongs to the language $L(G)$. The context-free grammar G_V for value and memory alias relations is:

Value Aliases: $V ::= M \mid M\bar{A}V \mid VAM$
 Memory Aliases: $M ::= \varepsilon \mid \bar{D}VD$

The grammar G_V has *non-terminals* V and M , and *terminals* A, \bar{A}, D , and \bar{D} . Readers can verify that the sample PEG in Figure 1 contains a path from b to c with label sequence $\bar{D}A\bar{D}A\bar{D}D\bar{A}$ that can be produced by the *V* non-terminal in G_V . So the grammar successfully deduces that b and c are value aliases. The intuition behind each production rule is:

$M ::= \varepsilon$ a is a memory alias of itself.

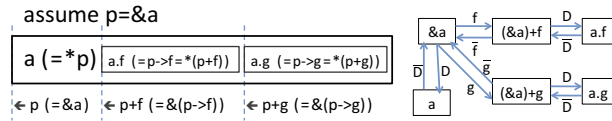


Figure 2: The relation of struct, field, and base pointer; and the corresponding PEG representation.

$M ::= \overline{D}VD$ Given $*p\overline{D}pVqD*q$ then, because p and q are value aliases, $*p$ and $*q$ are memory aliases.

$V ::= M$ Memory aliases are also value aliases.

$V ::= VAM$ Given $aVbAcMd$, the value of a propagates to c , which may reside in the same memory as d . Thus, a and d are value aliases. Similarly $V ::= MAV$.

Given this grammar, Zheng and Rugina go on to construct a hierarchical state machine and design an algorithm that decides whether two expressions are memory aliases. They also sketch an extension of the alias analysis algorithm for computing the value alias set of a single expression, which we adopt in MAS. Next, we describe how we extend the basic grammar to achieve field sensitivity.

3.1.3 Field Sensitivity

Field-sensitivity is necessary for our pointer analysis since we want to distinguish a generic pointer field from other fields in the same data structure. Field-insensitive analysis, on the other hand, treats all fields in a data structure as the structure itself.

Fields in C can be represented by means of *pointer arithmetic*: given a base pointer p and a field f , $\&(p \rightarrow f)$ is the *field pointer* which points to a field inside the structure $*p$. We use $p + f$ to denote $\&(p \rightarrow f)$, and $p + f$ is in fact the result of offsetting p by a fixed number of bytes determined by field f . The other constructs $p \rightarrow f (= *(p + f))$ and $a.f (= (\&a) \rightarrow f = *(\&a + f))$ are merely syntactic sugar, as shown in Figure 2.

To support field-sensitivity in our pointer analysis, we first add edges to the PEG to represent the field relations. For every field descriptor, we create a field label f_i . Then for each base pointer p , if its field pointer $p + f_i$ exists in the program, we add an edge labeled f_i from p to $p + f_i$ and an inverse edge \overline{f}_i in the opposite direction. As shown in Figure 2.

Zheng and Rugina [27] suggest adding $V ::= \overline{f}_iVf_i$ to the grammar G_V for field-sensitivity. With this addition, the grammar becomes:

$$M ::= (\overline{D}VD)?$$

$$V ::= M \mid (M\overline{A})^*V(AM)^* \mid \overline{f}_iVf_i$$

```
typedef struct {
    void *header; // call this field F
    int status;
} KOBJECT;
KOBJECT *x, *y;
*x = *y;
```

Figure 3: Example code of struct assignment.

However, we observe that this is insufficient to track all the value aliases because of a feature in C called *struct assignment*. One can assign a structure to another as if they were both simple variables, and the effect is the same as doing assignments between corresponding fields recursively (because each field can possibly be an embedded structure).

Figure 3 shows a simple example where handling struct assignment becomes crucial to the analysis. $x \rightarrow header$ and $y \rightarrow header$ are value aliases, as well as $x \rightarrow status$ and $y \rightarrow status$. However, the extended grammar suggested by Zheng and Rugina can not capture these alias relationships correctly. The relevant edges connecting from $x \rightarrow header$ to $y \rightarrow header$ produce the label sequence $\overline{D}fD\overline{A}DfD$, which cannot be generated from the “V” non-terminal in Zheng and Rugina’s extended grammar. Struct assignment is a common feature widely used in various programs. We must handle it properly when computing value aliases.

Struct assignment can only happen when the two variables involved are of the same type, and that type is precisely known to the compiler. Taking advantage of this property, we have an effective and efficient fix for Zheng and Rugina’s algorithm. In the program expression graph, we expand each struct assignment to the individual assignments of all corresponding fields. In the example code, $*x = *y$ is expanded to $x \rightarrow header = y \rightarrow header; x \rightarrow status = y \rightarrow status$. If some field is an embedded struct, then this expansion is done recursively, eventually down to the “leaf” fields. The program expression graph built this way is free of struct assignment, and Zheng and Rugina’s extended grammar works properly on this kind of PEG.

3.2 Type Candidate Inference

We have implemented Zheng and Rugina’s algorithm with our extension to do demand-driven pointer analysis. MAS uses this pointer analysis to derive the set of type-related information for identifying dynamic data object in memory traversal. The set of type-related information has two parts: candidate types for generic pointers and candidate types for pool tags [18] Note that we use candidate types and type candidates interchangeably. Next we will describe how we derive them in detail.

3.2.1 Candidate Types of Generic Pointers

A generic pointer is a pointer whose type definition does not reveal the actual type of the data it refers to. In MAS, we consider two kinds of generic pointers: *void** and pointers in linked list constructs. We consider linked list constructs because the declared type of its pointer fields does not reflect the larger data structure it links together in the list.

For an expression p of type *void**, its candidate types are the set of types of its value aliases. For instance, given $FOO * q; void * p; p = q$, we get p 's candidate type as $FOO*$. To derive the candidate types for a pointer field f_i of type *void**, we need to consider all its instances. Thus, f_i 's candidate types are the set of types of all the value aliases of the pointer field's instances in the form of $X \rightarrow f_i$.

We need to solve two problems to compute candidate types for pointer fields in linked list constructs. First, we are concerned with the larger data structures that are linked together in a list. When a linked list pointer field's value alias is in the form of $\&(a \rightarrow f_i)$, we say its *nested* candidate type is $\&(A \rightarrow f_i)$ where a 's type is $A*$. This nested candidate type allows us to identify the larger data structure A when the linked list pointer points to its field f_i . For simplicity, we still use candidate types when we discuss linked list constructs.

Second, the head node and the entry nodes in a linked list tend to have different data structures. If we do not differentiate them, the candidate types of a linked list pointer field will have both types, which causes unnecessary type ambiguity. To solve this problem, we leverage the semantics of APIs for linked list constructs. For instance, *InsertTailList* is a function in Windows [16] for inserting an entry at the tail of a doubly linked list. It takes two parameters, *ListHead* and *Entry*. To differentiate the list head and entry, we compute the value alias sets of *ListHead/InsertTailList* and *Entry/InsertTailList*, where $a/func$ represents the parameter a of a function $func$. Then we match value aliases from each set based on the call stack. For each valid pair of $\&(a \rightarrow f_i)$ and $\&(b \rightarrow f_j)$, we derive that a list head at $\&(A \rightarrow f_i)$ has a nested candidate type of $\&(B \rightarrow f_j)$ where a 's type is $A*$ and b 's type is $B*$. This approach requires prior knowledge of all linked list constructs and their APIs. Given the limited number of such constructs, it is not a hurdle for adapting MAS to large programs like the Windows kernel and drivers.

To control the number of candidate types, we apply three refinement techniques to the basic algorithm. First, for every linked list pointer p , MAS excludes all value aliases q of p if q 's type is different from p . This is because we did not observe any link list pointers being converted to other types, and such value aliases are almost

```
struct A {
    struct C ac;
    struct D ad;
};
struct B {
    struct C bc;
    struct E be;
};
```

Figure 4: Example of a common nested type.

always false positives introduced by imprecise analysis. Second, for each pointer path from p to its value alias q , we check if it involves a type cast to *void**. If so, we will ignore the path. We do this for two reasons: the type before the cast has already revealed the candidate type, and we avoid the noisy aliases following the type cast. Third, when there are multiple candidate types, we look for the largest common nested types among all candidate types. If such a common nested type exists, we use it as the single candidate type. In the example shown in Figure 4, the largest common nested type of struct A and struct B is struct C.

3.2.2 Candidate Types of Pool Tags

In recent Windows operating systems, pool tags [18] are used to track memory allocations of one or more particular data types by a kernel component. A pool tag is a four-character literal passed to the pool manager at a memory allocation or deallocation. One such API is *ExAllocatePoolWithTag*. For many pool tags, a memory block with a particular pool tag is always allocated for a unique data type. For instance, "Irp" is always for the data type *IRP*. In MAS, we use static analysis to automatically unearth the associations between a pool tag and data types and use them in our memory traversal. We call the types associated with a pool tag the candidate types for the pool tag. Note that such associations are not limited to Windows. In the Linux kernel, the slab allocator is used to provide specialized per-type allocations. In this paper, our design and implementation are focused on supporting Windows kernel pool management. But the techniques can be easily ported to support Linux kernel memory management.

Our approach for computing pool tag's type information is similar to the approach used for linked list constructs. Taking *ExAllocatePoolWithTag* as an example, we first compute the value alias sets for *return/ExAllocatePoolWithTag* and *Tag/ExAllocatePoolWithTag*, where the former represents the return value of *ExAllocatePoolWithTag* and the latter is the pool tag parameter. Since pool tags are usually specified directly at function calls for memory allocations, we do a simple traversal by following as-

signments on the program expression graph to compute the “value alias” set of *Tag/ExAllocatePoolWithTag*. Then we match the value aliases in each set based on the call stack. For instance, given the following code, our analysis will infer that the pool tag 'DooF' is associated with the type *FOO*.

```
FOO *f = (FOO*) ExAllocatePoolWithTag( NonPagedPool, sizeof( FOO ), 'DooF');
```

4 Memory Traversal

In this section, we describe how MAS locates dynamic data objects in a given memory snapshot and identifies their types. The inputs to this step include the memory snapshot, the type related information derived from static analysis, and the symbol information [15] for each loaded module in the memory snapshot (if it is available).

The basic memory traversal in MAS is similar to previous work [2, 3, 19]. It first locates the static objects in each loaded module based on the symbol information, then performs a breadth-first traversal by following pointers in the static objects and all newly identified data objects until no new object is added. MAS follows generic pointers for which our static analysis was able to derive a unique type. In the absence of a robust method for resolving multiple type candidates during memory traversal, MAS ignores all ambiguous pointers.

In order to increase coverage, MAS uses the associations between a pool tag and data types that may appear in memory blocks labeled with this tag. We directly identify data objects (i.e., without following a pointer) when a pool tag is only associated with a single data type.

Invalid pointers are common in kernel memory for many reasons. There may be a lag between the time a pool block is allocated and the time it is initialized. Also, a dangling pointer may point to a pool block that was freed and allocated again for different use. There exist even data objects that are partially initialized due to performance optimizations (or programming errors).

Our solution to invalid pointers have two main components: constraint checking and error correction. We only add a new data object during memory traversal or during type assignment based on pool tags if it satisfies the following constraints.

- **Size Constraint:** a data object must lie completely within a memory block. (We collect the information of all allocated memory blocks before the memory traversal.)
- **Pointer Constraint:** a data object's pointer fields must either be null or point to the kernel address range.

- **Enum Constraint:** a data object's enum fields must take a valid enum value which is stored in the PDB files.

- **Pool Tag Constraint:** the type of a data object must be in the set of data types associated with the pool block in which the data object is located.

KOP [3] only checks size and pointer constraints, which is not effective for smaller sized objects since they tend to have fewer pointer fields and fit into most memory blocks. The checking of pool tag constraints allows MAS to mitigate this problem.

A final constraint states that two incompatible objects cannot occupy overlapping addresses. We say two overlapped objects are type compatible if their overlapped parts have equivalent types (i.e., with the same memory layout after being expanded into primitive types and pointers). For example, one object may be a sub structure of the other object. We check this constraint before accepting an object. A violation of this constraint is a clear indication that an error has been made or is about to be made. Either the new object or the existing object that collides with it must be wrong.

We select one of the two objects based on several confidence criteria. Objects that we found without following pointers, such as global variables or objects identified through pool tags, are not subject to invalid pointer errors. We always select such objects over other objects. If both objects were found by following pointers, we select the larger object, since we typically check more constraints for larger objects. If the decision is to reject the existing object, we also remove all objects that were added by following its pointers recursively and cannot be reached from other objects.

5 Integrity Checking

The last step in identifying a kernel rootkit's memory footprint is to perform integrity checking. The inputs to integrity checking include the memory snapshot, the list of data objects identified from memory traversal, the pdb and image file of each loaded module when it is available. Note that the set of image files serves as the white list of trusted code.

A rootkit tampers with kernel memory for two main purposes: run its own code and hide its own activity. To do so, a rootkit either hijacks kernel execution by modifying code or function pointers or directly manipulates kernel data. MAS checks three kinds of integrity as follows.

- **Code Integrity:** trusted code in memory should match with the image file on disk.

- **Function Pointer Integrity:** function pointers should point to the trusted code.
- **Visibility Integrity:** data objects found by MAS should be visible to system tools (e.g., those available in a debugger for listing processes and modules).

The visibility integrity checking allows MAS to report *hidden objects* such as hidden processes and hidden modules. For instance, to find hidden processes, MAS uses a debugger command (e.g., `!process`) to get the list of processes in a memory snapshot, then compares it with the process objects found by memory traversal. If a process object is not in the list returned by the debugger command, it is marked as a hidden process. To check function pointer integrity, MAS inspects not only well known hooking points such as the system call table but also each function pointer in the data objects identified from the memory traversal. Function pointers that point to a memory region outside of the trusted code are reported as *suspicious function pointers*. Violations of code integrity are reported as *suspicious code hooks*.

MAS can be used in two scenarios: detect if a real-world system is infected by rootkits or analyze the behavior of a malware sample in a controlled environment. If the white list of trusted code is complete, any integrity violation can be automatically attributed to rootkit infection. It is trivial to construct such a complete list based on a copy of a clean system in a controlled environment. However, when checking real-world systems, such a complete list may be available in some cases (e.g., machines inside an enterprise or virtual machines in a cloud) but not always. When the list of trusted code is incomplete, we will need an expert to inspect integrity violations reported by MAS before deciding if a system is infected. We will report our experiences of detecting rootkits from real-world crash dumps in Section 9.

6 Implementation

We implemented MAS with 12,000 lines of C++ code for the static analysis and 24,000 lines of code for memory traversal and integrity checking.

For static analysis, we developed a PREfast [14] plugin to extract information from the AST trees generated by the Microsoft C/C++ compiler. We implemented the pointer analysis as a stand alone DLL that, upon request, computes the value alias set for a given program expression based on the information extracted by the PREfast plugin. Since our pointer analysis is demand-driven and can run in parallel, we implemented our type candidate lookup to take advantage of that. We run a separate parallel job for each generic pointer. After all parallel jobs

are done, we merge the inferred type relations together. We implemented the parallel type candidate lookup on a cluster running Windows HPC Server 2008 R2 [17].

For analyzing memory snapshots, the key logic was implemented as an extension of WinDbg [13]. In addition, we implemented a DLL based on the Debug Interface Access SDK [12] to programmatically access the symbol information stored in PDB files [15].

During memory traversal, we frequently access two kinds of data, allocated memory blocks and data objects identified, where a memory block may contain multiple data objects and no two data objects overlap in memory. We use a multi-level data structure in MAS in order to obtain fast *store* and *retrieve* operations for the two kinds of type data. At the bottom level, we use a page-table like data structure to achieve fast lookup for an arbitrary address. Here a hash table simply based on the starting addresses of allocated memory blocks cannot meet our need because a given memory address may fall into the middle of a memory block. Given a memory address, if there exists a memory block that covers it, the lookup in the bottom-level structure returns a pointer to a data structure that stores all the information for the memory block. In this data structure, we use a sorted list to store all the data objects identified in the memory block. We choose a sorted list because the number of data objects on a single memory block is small.

To speed up type check, we maintain a cache of matched subtypes and their offsets for each aggregate type and check the cache first before doing the type consistency check in a brute force way. We choose to use a cache because, for an aggregate type, type consistency checks usually occur repeatedly for a small number of its nested types.

7 Evaluation

This section evaluates the accuracy, robustness and performance of MAS. We perform the evaluation on three sets of memory snapshots: (a) 154,768 memory snapshots derived from our large scale kernel malware analysis; (b) a set of 837 real-world crash dumps from end user machines running Windows 7; (c) a set of 11 real-world crash dumps from end user machines running Windows Vista SP1. The last set of Windows Vista SP1 crash dumps allowed us to compare MAS directly to KOP [3]. For our analysis on real-world crash dumps, the white list of trusted code contains all the binaries available on Microsoft's symbol server. For our analysis of malware samples, the white list of trusted code contains all the binaries from a clean VM image. Our experiments were conducted on a machine running Intel Xeon Quad-Core 2.93 GHz with 12 GB RAM unless specified otherwise.

Id	Size (MB)	Modules	Fct. ptrs. MAS	Fct. ptrs. KOP	FP. KOP	FN. KOP
1	245	154	64	43	22	21
2	149	144	55	47	28	8
3	305	203	673	N/A	N/A	N/A
4	270	157	257	236	37	21
5	247	159	75	45	19	30
6	127	125	46	38	9	8
7	315	157	283	265	30	18
8	250	141	105	97	26	8
9	204	144	50	40	26	10
10	255	141	167	157	24	10
11	312	203	235	189	11	46

Table 1: Results on eleven Windows Vista SP1 crash dumps. “Fct. ptrs.” represents the number of function pointers correctly identified by MAS or KOP.

7.1 Accuracy and Robustness

The goal of this section is to evaluate the accuracy and robustness of MAS. We face the general difficulty that it is hard and time consuming to obtain an object mapping that is known to be correct (i.e., ground truth) even in a controlled environment. For the real-world crash dumps for which we had no data beyond the crash dumps themselves, it appears unclear if and how a ground truth could be established. Given these methodological difficulties, much of the evidence we present in this section has to be indirect.

Our first data set consists of the outputs of MAS on the 837 Windows 7 crash dumps. We tried to establish whether the function pointers reported by MAS as suspicious are indeed function pointers. We inspected whether the target of the function pointers appeared to be the beginning of a function. The vast majority of function pointer targets contained a small set of code patterns corresponding to function preambles. This allowed us to automate most of pointer checks by running a program that checks for these patterns. We inspected the remaining pointers manually. We applied a second criterion to the function pointers whose targets did not appear to be code. We accepted all function pointer candidates that were fields in objects whose existence could be derived directly and unambiguously from the symbol information. This included global variables and objects that could be reached from global variables by following only uniquely determined typed pointers. This left us with a total of 24 dubious pointers out of total of 398,987 function pointers that MAS had output.

The eleven Windows Vista SP1 crash dumps in our data set allowed us to perform a direct comparison with KOP. We examined manually all discrepancies between the outputs of MAS and KOP. KOP appeared to suffer from both false positives and false negatives (see Ta-

ble 1). We first examined all function pointers returned by MAS and found that they are valid. Then we examined manually the targets of all function pointers reported by KOP that had not been output by MAS. None of the targets appeared to be the start of a function. Thus, we classified these pointers as false positives for KOP (FP. KOP in Table 1). We also observed a number of function pointers that were found by MAS, but not by KOP. Since we had concluded that the targets of these pointers are function entry points, we classified them as false negatives for KOP (FN. KOP in Table 1). KOP missed as much as 40% of the function pointers found by MAS. Furthermore, KOP as much as 40% of the function pointers reported by KOP appear to be incorrect.

We also tried to interpret the function pointers returned by MAS. A large fraction of the reported function pointers appeared to point to third-party drivers that were not included in our static analysis. However, in addition to detecting the footprints of widely used anti-virus software, we also found clear signs of rootkit infections in five out of the eleven crash dumps. We will discuss how we detect rootkits in real-world crash dumps in Section 9.

Next, we attempted to estimate the internal consistency of the objects found by MAS. We examined the complete kernel object mappings produced by MAS for inconsistent pointers. These are pointers whose type is incompatible with the object type that the object mapping has assigned to the pointer’s target. For example, an object mapping might contain an object of type T_1 at address A . Another object in the mapping might contain a pointer P of some other type $T_2 \neq T_1$ that also points to A . P is an inconsistent pointer. Such inconsistencies may exist even if the object mapping is error free because of invalid pointers in objects and because of memory corruptions in the crash dump. But they may also indicate errors in the object mapping, for example as a result of following invalid pointers. We call an object inconsis-

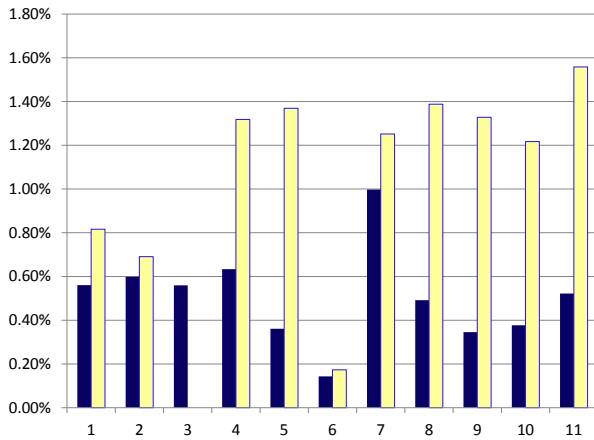


Figure 5: Percentage of inconsistent objects in the object mappings for MAS (left) and KOP (right). KOP did not produce a result for the third dump.

tent if it is the target of at least one inconsistent pointer. Figure 5 displays the percentage of inconsistent objects in the object mappings found by MAS and KOP for the Windows Vista SP1 crash dumps. We consider this number to be an indication of the correctness of the object mapping. On average, the object mappings produced by MAS contain 0.5% inconsistent objects. This number is 1% for the objects mappings produced by KOP.

7.2 Performance

This section evaluates the running time of MAS.

Static Analysis We performed the static analysis for Windows XP SP3, Windows Vista SP1 and Windows 7. Our evaluation is focused on Windows Vista SP1 since it allows us to compare MAS and KOP directly. The static analysis on Windows Vista SP1 includes the Windows kernel and a set of 63 standard drivers (such as win32k, ntfs and tcpip). This is the same set of drivers analyzed by KOP. The code base has 3.5 million lines of code. The program expression graph has 2.2 million nodes and 7.3 million edges. MAS performed almost 23,000 candidate type lookups.

We performed the static analysis on a 100 node cluster running Windows Server 2008 R2 HPC Edition, where each node has two Quad-Core 2.5 GHz Xeon processors with 16 GB RAM. Each node was used to perform 228 candidate type lookups. The whole static analysis took less than 5 hours. The corresponding time for KOP reported in [3] is 48 hours on a somewhat older, single processor machine.

The key advantage of MAS over KOP is that MAS's static analysis can run in parallel. This allows MAS to

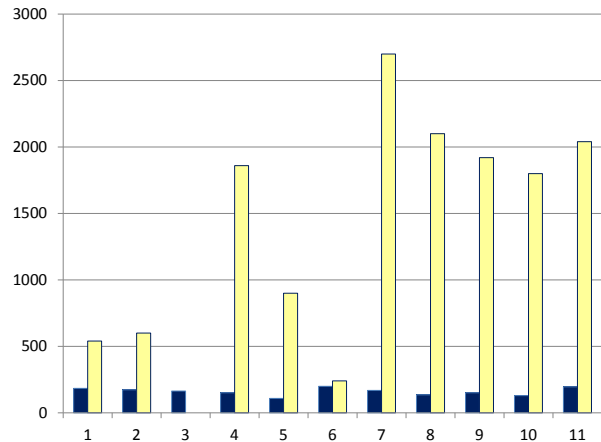


Figure 6: Running times in seconds of MAS (left) and KOP (right) on eleven real-world Windows Vista SP1 crash dumps. KOP did not produce a result for the third dump.

finish the static analysis in 5 hours on 100 nodes. On the other hand, the combined machine time of 500 hours is much larger than KOP's running time. This is partly because MAS does not achieve perfect parallelization. For instance, it takes 0.5 hour to load the program expression graph into memory on every node; alias analyses for indirect calls are computed on demand on each node and thus are not shared, which causes repeated computations as well. Furthermore, MAS converts a program to the Static Single Assignment (SSA) form conservatively, which increases the computation.

Dynamic Analysis Next, we report on the total running times of memory traversal and integrity checking of MAS on three sets of memory snapshots. Figure 6 displays the running times of MAS and KOP on the eleven Windows Vista SP1 crash dumps. On average, MAS (160 seconds per dump) is more than 9 times faster than KOP (24.5 *minutes* per dump). KOP failed to terminate on crash dump 3 within the two hour time limit we had set.

Figure 7 displays the distribution of MAS's running times on the 837 Windows 7 crash dumps. The running times are concentrated between 40 and 160 seconds. The average running time is 105 seconds, and 99.9% of all runs complete in less than 5 minutes.

Finally, the average running time of MAS on the 154,768 memory snapshots from our large-scale malware study is 31 seconds. The running time distribution is highly concentrated around this value.

In summary, our experiments demonstrate that MAS can quickly and accurately analyze real-world crash dumps as well as memory snapshots of virtual machines.

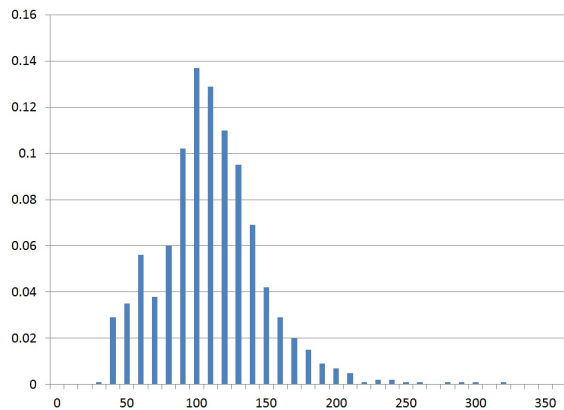


Figure 7: Running time (in seconds) distribution of MAS on 837 real-world Windows 7 crash dumps.

When compared directly, MAS was nearly an order of magnitude faster than KOP. MAS did not misidentify or miss any functions pointers found by KOP in the eleven Windows Vista SP1 dumps, but KOP missed or misidentified as much as 40% of the suspicious function pointers.

8 Kernel Malware Study

In this section we present the results of our study of a large collection of 154,768 potential malware samples that we obtained from a major vendor of anti-virus software. These samples originated from a variety of sources. Their behavior was unknown to us. This included the question whether a sample even contained malware. All samples were different types of Windows binaries: executables (.exe), dynamically linked libraries (.dll) and drivers (.sys).

We used MAS to analyze the samples. More precisely, for each sample, we booted a clean Windows XP SP3 VM with 256 MB of RAM and one virtual processor and loaded and executed it. We ran .exe's directly. We ran .dll's with the help of a standard executable that loads a dll and causes its DllMain function to be executed. We loaded drivers (.sys) using the service control manager (sc.exe). After launching the sample, we waited for one minute, then took a memory snapshot of the VM, converted it into a Windows crash dump and ran MAS over the crash dump.

In order to gain additional insight into the events that take place in the VM, we wrote a driver that makes most of the kernel address space of the VM not executable (by setting the corresponding bits in the page tables) and catches and records any non-execute (NX) page faults. The driver also records the loading and unloading of kernel modules and the allocation and deallocation of pool

blocks. We loaded the driver in the VM before launching the sample.

We used a 25 node compute cluster to evaluate all 154,768 samples. The cluster nodes were running Windows Server 2008 R2. We used Hyper-V as our Virtual Machine Monitor. On each cluster node, we ran between 4 and 8 VMs, running a total of 164 VMs simultaneously at any time. Each job ran for 2 to 3 minutes. Since the VM jobs were I/O bound we took a number of measures to manage disk traffic: The VMs used differencing disks based on a single base image. We interleaved the startup of VMs such that the I/O intensive phases at the beginning and end of some jobs coincided with the one minute idle period of other jobs. All 154,768 jobs completed in less than 48 hours.

MAS reported kernel behaviors for only 89,474 of the samples. We analyzed the events recorded by our driver for the remaining 65,294 samples for which MAS had output no results. The driver logs showed that, in all but 1286 cases, neither module loading nor non-executable page faults were recorded. For the 1286 samples, the driver logs showed that no non-executable page faults were detected, and some modules were loaded after the sample was launched but all of the modules had been unloaded before the memory snapshot was taken. Based on this evidence, it appears that the memory snapshots for which MAS reported no results did not contain any data that MAS should have reported.

There are several potential reasons for the relatively large number of samples without reportable kernel behaviors. As stated above, some of the samples may simply not have been malware. Also, the crude way in which we launch the samples may have caused samples to fail to execute. It may also have caused malware not to become active. Techniques for reliably triggering malware have been studied elsewhere [5, 8] and are not the focus of this paper. The rest of this section presents the results of our analysis for the 89,474 samples for which MAS reported kernel behaviors.

8.1 General Behavior Statistics

Table 2 displays counts on the different categories of kernel behavior we observed. The count for a category is the number of samples that displayed behavior in that category. Some samples displayed behaviors in more than one category. Most categories correspond to modifications of static data structures that can be detected with existing tools. *IDT* represents modifications to the function pointers in the processor's interrupt descriptor table. *Sysenter* represents modifications to the hardware register that determines the target address of a *sysenter* instruction. *Callgate* represents similar modifications to function pointers in hardware-defined call gate structures.

Category	Count
IDT	20
Sysenter	1
Callgate	23
Syscall Table (SSDT)	3652
Hidden Process	1476
Hidden Module	43828
Code Hooks	17744
Module Imports and Exports	103
Function Pointer	84051

Table 2: Distribution of malware behaviors.

The next group of categories represents static software-defined function pointers. The system call table (SSDT) is a table of function pointers to the individual system call handler functions. *Hidden process* and *hidden module* stand for attempts to hide processes or modules by removing them from the data structures Windows maintains to keep track of processes and loaded modules. *Code Hooks* represent modifications of legitimate executable code. *Module Imports and Exports* represent tampering with the function pointers in the import and export lists of loaded modules.

Finally, the *Function Pointer* category includes modifications to function pointers in data objects found in MAS’s memory traversal. Most of the objects are dynamic data (i.e., reside in the kernel pool) and some of them are from global variables. This is by far the most frequent category. About 94% of the samples display this behavior in some form. Since this is also the one category for which existing tools provide at best limited information, we examined it in more detail.

8.2 Function Pointer Hooking

We found that the samples were hooking a total of 191 unique function pointer fields from 31 different data structures belonging to the Windows kernel and five drivers (ntfs, fastfat, ndis, fltmgr, null). Figure 8 shows the number of samples that hooked each of the 191 function pointer fields. We observe a high concentration on a small set of pointers and a long tail. The two plateaus between 0 and 60 correspond mostly to function pointers from `nt!_DRIVER_OBJECT` and `nt!_FAST_IO_DISPATCH`. Almost 50% of the function pointers were hooked by only one or two samples.

We also counted the number of distinct dynamic function pointers hooked by each sample. The distribution is displayed in Figure 9. It is highly concentrated. Almost half the samples hook exactly 32 function pointers. There is a smaller concentration around the value 4. This high concentration suggests that versions or exact copies

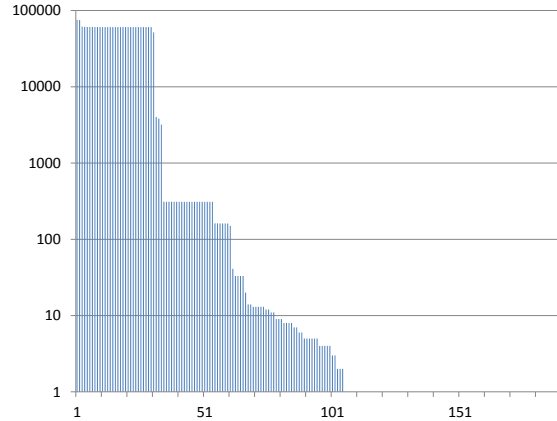


Figure 8: Number of samples that hooked each of the 191 different function pointers for which MAS detected hooking.

of the same underlying malware are present in a large number of samples. We further investigated this observation by clustering the samples.

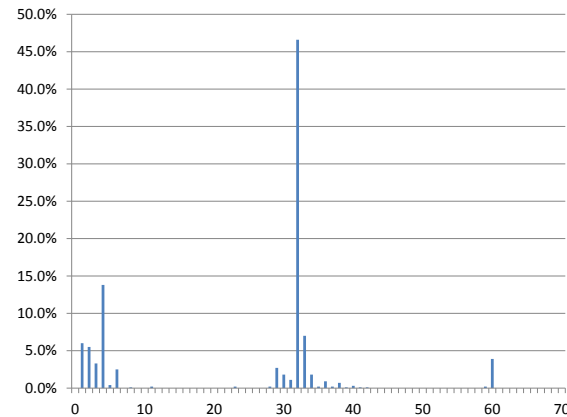


Figure 9: Distribution of the number of dynamic function pointers hooked by each sample

8.3 Clustering

To cluster samples, we first extracted the following information from MAS’s report as a sample’s footprint. For each suspicious function pointer, we use a tuple including “FUNCPTR” (indicating this tuple is about function pointers), function pointer field name, and data structure name. To differentiate the cases when different known drivers are hooked, we replaced the data structure name (“`nt!_DRIVER_OBJECT`”) with a driver name (e.g., “`\Driver \disk`”) for known drivers. For each code hook, we use a tuple including “CODEHOOK”, module name, function name, offset, and the number of

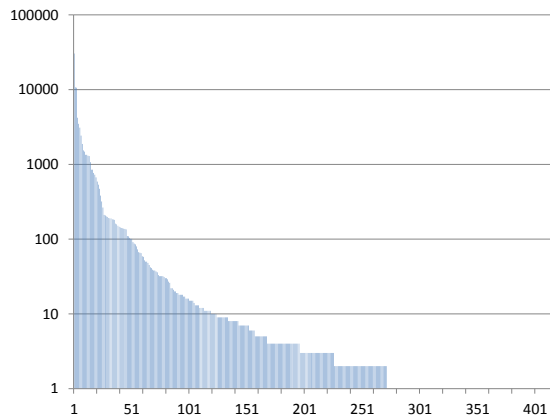


Figure 10: Sizes of clusters of samples with identical MAS footprints.

bytes that were modified. For hidden modules or processes, we simply used a tuple “HIDDEN_MODULE” or “HIDDEN_PROC”. We handled other behaviors similarly. Note that we carefully chose not to include any names or values that are easily modifiable by malware (e.g., a malicious driver’s name or a hidden module’s name). The tuples in each sample’s footprint are sorted so that we can easily compare two samples’ signatures.

We assigned samples into the same cluster if they had identical footprints. This mapped the 89,474 samples into 414 clusters whose sizes ranged from 1 to 30,411. A total of 272 clusters contained at least two samples. Figure 10 shows the distribution of cluster sizes.

To understand whether all samples in a cluster used a single kernel driver, we counted the number of different sized drivers loaded by samples in each cluster (see Figure 11). A total of 209 clusters have at least two different sized drivers loaded. This indicates that different malicious kernel drivers have shown identical MAS footprints. Thus we can potentially use MAS’s footprints to automatically detect new malware samples. We leave the investigation of this approach to future work.

9 Crash Dump Study

In this section we report our experience in using MAS to detect kernel rootkits in real-world crash dumps. Since the white list of trusted code is incomplete for the end user machines from which the crash dumps were collected, we cannot automate the process of rootkit detection entirely. However, we can leverage the findings from our kernel malware study to identify suspicious crash dumps before manually inspecting them.

From Table 2 we can see that the three most common behaviors of rootkits are hooking function pointers, hid-

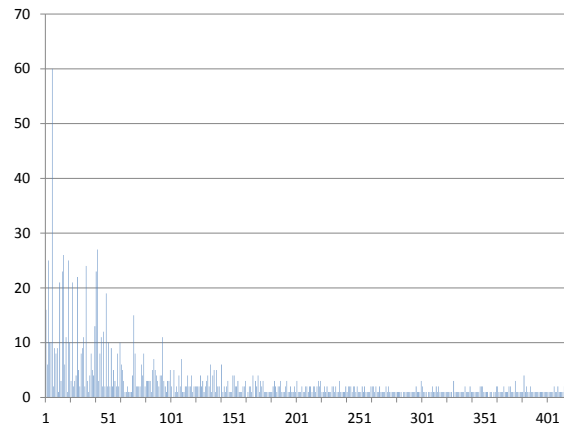


Figure 11: The numbers of different sized drivers loaded by samples of each cluster.

ing modules, and placing code hooks. Since many suspicious function pointers reported by MAS point to benign third-party drivers that are not on our white list, simply using the existence of suspicious function pointers is not an effective way to identify suspicious crash dumps. For rootkits that hook both function pointers and hide modules, the hooked function pointers usually do not point to a loaded module but either a pool block, a hidden module or some other memory region. We used this observation to ignore function pointers whose targets fall into loaded modules. We are aware that this may cause us to miss *non-stealthy* kernel malware that simply installs a driver. To handle such cases, we would need to either grow the white list or do more manual analysis. We also ignore function pointers whose targets do not appear to be the beginning of a function since they do not allow us to differentiate reliably between buggy rootkits and memory corruptions. In our study we used these conditions to do initial filtering to identify suspicious dumps. This initial filtering was done automatically.

For the eleven Windows Vista SP1 crash dumps, we found seven of them to be suspicious after the initial filtering. Our manual investigation confirmed that five crash dumps contain rootkits (e.g., hooking several driver’s dispatch routines, hiding its own driver). The other two were benign because the code hooks were placed by two anti-virus systems. Each of them hooked one of two very frequently called functions, *KiFastCallEntry* and *SwapContext*. We concluded that a code hook was placed by anti-virus software if the hook’s target falls into a module and internet search results indicated that the module belongs to an anti-virus vendor based on the module’s name.

For the 837 Windows 7 crash dumps, we found 177 suspicious dumps after the initial filtering. We quickly verified that 85 dumps that contain hidden modules were

all infected by kernel rootkits. Out of the remaining 92 crash dumps, 82 dumps only contain code hooks, and the other ten contain suspicious function pointers that do not point to a loaded module. We manually analyzed these ten dumps and found that five of them contain rootkits and the other five have corrupted global function tables which let them pass the initial filtering. We cannot decide if the corruptions were due to a rootkit or a kernel bug. The 82 dumps with only code hooks have 37 different hooking patterns. For each hooking pattern, we picked one dump and manually inspected it with MAS's report. Surprisingly, all the code hooks appeared to be placed by anti-virus software.

In summary, with the process described above, we were able to quickly identify five Windows Vista SP1 dumps and 90 Windows 7 dumps that contain kernel rootkits. All the manual inspections described in this section took a total of less than one hour. This demonstrates that MAS is an effective tool for identifying rootkit footprints in real-world systems.

10 Related Work

MAS is not the first system that attempts to identify a kernel rootkit's footprint in a memory snapshot. But it is the first practical system that can do so with high accuracy, robustness and performance.

Our work was inspired by KOP [3]. While KOP is the first system to type dynamic data in a kernel memory snapshot with very high coverage, it lacks in robustness and performance. Our evaluation has shown that MAS is an order of magnitude faster than KOP in both static analysis and memory traversal. More importantly, when analyzing real-world crash dumps of systems running Windows Vista SP1, we observed no errors in MAS's output. In contrast, up to 40% of the function pointers reported by KOP appeared to be incorrect.

Kernel integrity checking has been studied in a large body of work. SBCFI [19] and Gibraltar [2] both leverage type definitions and manual annotations to traverse memory and inspect function pointers. Both fall short in data coverage as a result of not handling generic pointers [3]. A recent system called OSck [7] also discovers kernel rootkits by detecting modifications to kernel data. Instead of memory traversal, OSck identifies kernel data and their types by taking advantage of the slab allocation scheme used in Linux. It provides per-type allocations and enables direct identification of kernel data types. The slab allocator is unavailable on Windows operating systems, which makes OSck less useful for Windows. This problem cannot be solved by the mapping between pool tags and data types since it is *not* a one-to-one mapping. Worse, a pool tag may correspond to different types, and several data structures may be stored in one pool block.

MAS leverages source code and program-defined types to identify dynamic data and their types. Several other systems have tried to solve this problem without access to source code and type definitions. Laika [4] uses Bayesian unsupervised learning to infer data structures and their instances. REWARDS [11] recognizes dynamic data and their types when they are passed as parameters to known APIs at runtime. TIE [10] reverse engineers data type abstractions from binary programs based on type reconstruction theory and is not limited to a single execution trace. These reverse engineering tools are more effective for analyzing small to medium scale programs than for large-scale programs like the Windows kernel. Both MAS and KOP demonstrate that source code is critical for achieving high data coverage when analyzing kernel memory snapshots.

WhatsAt [20] is a tool for dynamic heap type inference. It uses type information embedded in debug symbols and attempts to assign a compatible program-defined type to each heap block by checking type constraints. If a block is untypable, WhatsAt uses it as a hint for heap corruptions and type safety violations. The main difference between WhatsAt and MAS is that whatsat cannot scale to large programs such as the Windows kernel.

MAS leverages a new demand-driven pointer analysis algorithm to enable precise but fast analysis for identifying type candidates for generic pointers in large-scale C/C++ programs. The key idea behind the demand-driven analysis is to formulate the pointer analysis problem as a Context-Free Language (CFL) reachability problem, which was explored in previous work [21, 24, 23, 27]. In [21], Reps first introduced the concept of transforming program analysis problems to graph-reachability problems. In [24], Sridharan *et al.* apply this idea to demand-driven points-to analysis for Java. In [23], Sridharan and Bodik present a refinement-based algorithm for demand-driven context-sensitive analysis for Java. In [27], Zheng and Rugina describe a demand-driven alias analysis algorithm for C. We adopt their algorithm and extend it to support field-sensitivity. We also achieve context-sensitivity in a way similar to [23]. In KOP [3], Carbone *et al.* extend the algorithm presented in [6] to be context- and field-sensitive. The key advantage of MAS over KOP is that MAS's static analysis can run in parallel.

MAS works on memory snapshots to analyze kernel rootkit behavior. Several other systems [9, 22, 26] have used virtualization-based dynamic tracing for the same purpose. Soft-timer based attacks [25] are detectable by MAS since the callback function pointer injected by the malware is always in memory and can potentially be detected by MAS.

11 Limitations

A key limitation faced by MAS is that an attacker who is familiar of MAS's design can potentially disrupt MAS's memory traversal by manipulating the kernel memory. MAS checks several constraints (see Section 4) before adding a new data object. If an attacker were able to find some pointer or enum fields in a data structure that may take arbitrary values without crashing the OS, he could potentially mislead MAS to reject instances of such a data structure by changing them to violate the pointer or enum constraints. The impact of this limitation remains unclear because we are not aware of such data structures. Moreover, even when such data structures exist, it is unclear if they will affect the identification of security sensitive data (e.g., hooked function pointers).

Another limitation of MAS is due to the existing implementation in Windows. Currently an attacker can modify the tag of a pool block without crashing Windows, and thus use it to mislead MAS. However, this limitation can be eliminated if the pool manager checks the tag of a pool block against the expected pool tag passed as a function argument when the pool block is freed.

12 Conclusions

We have presented MAS, a practical memory analysis system that can accurately and quickly identify a rootkit's memory footprint. We applied MAS to analyze 848 crash dumps collected from end user machines and 154,768 potential malware samples obtained from a major anti-virus vendor. Our experiments show that MAS was able to quickly analyze all memory snapshots with typical running times between 30 and 160 seconds per snapshot and with near perfect accuracy. With MAS, we were able to quickly identify 95 crash dumps that contain rootkits. Our kernel malware study shows that rootkits hooked 191 different function pointers in 31 different data structures. Furthermore, it demonstrates that many malware samples installed different kernel drivers but had identical memory footprints, which suggests a future research direction on leveraging memory footprints to automatically detect new malware samples.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful feedback. We are very grateful to many colleagues for their valuable feedback, suggestions and help throughout the effort of making MAS real: Alex Moshchuk, Anil Francis Thomas, Barry Bond, Bryan Parno, Chris Hawblitzel, David Molnar, Dennis Batchelder, Eddy Hsia, Galen Hunt, Gloria Mainar-Ruiz,

Helen Wang, Jay Stokes, Jeffrey Chung, Jen-Lung Chiu, Jim Jernigan, Pat Winkler, Randy Treit, Reuben Olinsky, Rich Draves, Ryan Kivett, Scott Lambert, Tim Shoultz, YongKang Zhu.

References

- [1] The Alureon rootkit. <http://en.wikipedia.org/wiki/Alureon>.
- [2] BALIGA, A., GANAPATHY, V., AND IFTODE, L. Automatic inference and enforcement of kernel data structure invariants. In *Proceedings of the 24th Annual Computer Security Applications Conference (2008)*.
- [3] CARBONE, M., CUI, W., LU, L., LEE, W., PEINADO, M., AND JIANG, X. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS) (November 2009)*.
- [4] COZZIE, A., STRATTON, F., XUE, H., AND KING, S. T. Digging for data structures. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (Berkeley, CA, USA, 2008)*, OSDI'08, USENIX Association, pp. 255–266.
- [5] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS 2008) (October 2008)*.
- [6] HEINTZE, N., AND TARDIEU, O. Ultra-fast aliasing analysis using CLA - a million lines of C code in a second. In *SIGPLAN Conference on Programming Language Design and Implementation (2001)*.
- [7] HOFMANN, O. S., DUNN, A. M., KIM, S., ROY, I., AND WITCHEL, E. Ensuring operating system kernel integrity with OSck. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (New York, NY, USA, 2011)*, ASPLOS '11, ACM, pp. 279–290.
- [8] KOLBITSCH, C., KIRDA, E., AND KRUEGEL, C. The power of procrastination: Detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011) (October 2011)*.
- [9] LANZI, A., SHARIF, M., AND LEE, W. K-tracer: A system for extracting kernel malware behavior. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (2009)*.
- [10] LEE, J., AVGERINOS, T., AND BRUMLEY, D. Tie: Principled reverse engineering of types in binary programs. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (Feb. 2011)*, pp. 251–268.
- [11] LIN, Z., ZHANG, X., AND XU, D. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10) (San Diego, CA, February 2010)*.
- [12] MICROSOFT. Debug interface access SDK. [http://msdn.microsoft.com/en-us/library/x93ctkx8\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/x93ctkx8(v=vs.71).aspx).
- [13] MICROSOFT. Debugging Tools for Windows. <http://www.microsoft.com/whdc/devtools/debugging/default.aspx>.
- [14] MICROSOFT. PREfast. [http://msdn.microsoft.com/en-us/library/ff550543\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ff550543(v=vs.85).aspx).

- [15] MICROSOFT. Symbols and symbol files. [http://msdn.microsoft.com/en-us/library/windows/hardware/ff558825\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/hardware/ff558825(v=vs.85).aspx).
- [16] MICROSOFT. Windows driver kit. <http://msdn.microsoft.com/en-us/windows/hardware/gg487428.aspx>.
- [17] MICROSOFT. Windows HPC Server 2008 R2. <http://www.microsoft.com/hpc>.
- [18] MICROSOFT. Windows kernel pool tags. <http://msdn.microsoft.com/en-us/windows/hardware/gg463213.aspx>.
- [19] NICK L. PETRONI, J., AND HICKS, M. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)* (October 2007).
- [20] POLISHCHUK, M., LIBLIT, B., AND SCHULZE, C. W. Dynamic heap type inference for program understanding and debugging. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2007), POPL '07, ACM, pp. 39–46.
- [21] REPS, T. Program analysis via graph reachability. In *Proceedings of the 1997 International Logic Programming Symposium* (October 1997).
- [22] RILEY, R., JIANG, X., AND XU, D. Multi-aspect profiling of kernel rootkit behavior. In *Proceedings of the 4th ACM SIGOPS/EuroSys Conference on Computer Systems* (April 2009).
- [23] SRIDHARAN, M., AND BODIK, R. Refinement-based context-sensitive points-to analysis for java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2006).
- [24] SRIDHARAN, M., GOPAN, D., SHAN, L., AND BODIK, R. Demand-driven points-to analysis for Java. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems Languages, and Applications (OOPSLA)* (October 2005).
- [25] WEI, J., PAYNE, B. D., GIFFIN, J., AND PU, C. Soft-timer driven transient kernel control flow attacks and defense. In *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC 2008)* (December 2008).
- [26] YIN, H., SONG, D., MANUEL, E., KRUEGEL, C., AND KIRDA, E. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)* (October 2007).
- [27] ZHENG, X., AND RUGINA, R. Demand-driven alias analysis for C. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (January 2008).

TACHYON: Tandem Execution for Efficient Live Patch Testing

Matthew Maurer

maurer@cmu.edu

Carnegie Mellon University

David Brumley

dbrumley@cmu.edu

Carnegie Mellon University

Abstract

The vast number of security incidents are caused by exploits against vulnerabilities for which a patch is already available, but that users simply did not install. Patch installation is often delayed because patches must be tested manually to make sure they do not introduce problems, especially at the enterprise level.

In this paper we propose a new *tandem execution* approach for automated patch testing. Our approach is based on a patch execution consistency model which maintains that a patch is safe to apply if the executions of the pre and post-patch program only differ on attack inputs. Tandem execution runs both pre and post-patch programs simultaneously in order to check for execution consistency. We have implemented our techniques in TACHYON, a system for online patch testing in Linux. TACHYON is able to automatically check and verify patches without source access.

1 Introduction

Most attacks target known vulnerabilities for which there are already patches. For example, Microsoft reported that only 0.12% activity in the first half of 2011 involved a zero-day attack for which a patch has been available for a month or less [19]. For the other 99.88%, exploits were successful simply because available patches were not installed. These statistics indicate that one of the best ways to reduce security incidents due to exploits is to simply patch vulnerable systems.

The need to rapidly deploy security patches in enterprise environments is hampered by the need to also test patches for problems. Bad patches often have more business risk than a security breach, suggesting that the ability to test patches and guard against such problems might result in faster deployment of security patches. Measures currently deployed in cloud environments deal with random failures, rather than systematic ones caused by bad

patches. As a result, current best practices amount to manual testing, which is slow, error-prone, and expensive. For example, NIST best practices recommend manual patch testing (which is slow) on pre-production environments (which are expensive to acquire and maintain) when available, or simply waiting to see if others report a problem or not [18]. While such approaches prevent bad patches from being applied, they increase the vulnerability window. Even when a pre-production environment is provided through virtualization, reducing the cost substantially, auxiliary services, such as databases, must be simulated. This leads to excessive administration overhead and compute overhead. Additionally, effects are captured in an often ad-hoc manner (e.g. by recording network traces), which can miss changes the administrator did not think to look for. For example, the US Air Force implements a centralized patch testing procedure for their half million managed machines, but as a result, delay patch rollout by up to a quarter year [14].

If we could automatically test patches, then we could shorten the vulnerable time window between when a patch is released until it is installed. However, automated patch testing faces several challenges.

First, in order to faithfully check that functionality is preserved in a patch, we should be able to test a patch on the system it will ultimately protect. Second, in order to be widely applicable, we should be able to test patches in the common scenario where the patch is a new binary program, as source is often not available. Third, we want to minimize manual effort. As patches can change the semantics of a program, a human will likely always need to be in the loop to determine if the semantic changes are meaningful. However, we still wish to automate the system as much as possible. Unfortunately, there is very little work on automated patch testing, and no previous work addresses all these requirements.

In this paper we propose the first techniques for live patch testing via tandem execution. Our insight is that current manual testing checks whether the executions of

a pre-patch and post-patch binary produce different outputs on known inputs. We call this *observational equivalence* between pre and post-patch.

Tandem execution uses this insight to automate patch testing by simultaneously running both programs on the same input. More specifically, in tandem execution one program runs live on the system (e.g., the patched program), with all system calls (syscalls) being serviced by the kernel. The second version of the program (e.g., the unpatched program) runs in tandem, but with each syscall to the kernel simulated by replaying the side effects from the corresponding calls of the live version. The replay prevents duplicating side-effects, such as writing to the same file twice.

If the two programs deviate on the syscalls issued, or the arguments to syscalls, then they are not observationally equivalent. We record the deviation and inform the user of the potential problem. At this point, the actions that can be taken are to halt the program, or to specify that the deviation is permitted. In order to continue testing, the user provides a rewrite rule that specifies how to handle the deviation, and automated patch testing continues. In our experiments we show that rewrite rules are small when needed, and often completely unnecessary for security-related fixes.

We implement our approach in a system called TACHYON. TACHYON is based upon syscall replay techniques for binary programs, but with a new twist. Existing system call replay schemes are designed to record system calls from one run of a binary for replay against *exactly the same binary*, e.g., [1, 11, 22]. Since both record and replay are against the same binary, the record step only needs to conceptually keep a snapshot of the memory cells affected by the syscall. The affected memory cells are typically determined by differencing the pre and post-syscall memory state. During replay, the memory cells at exactly the same addresses are replayed with the recorded data.

The twist in our setting is we want to replay syscalls to a *different binary*. Typically the two binaries will have a different memory layout, and may make different syscalls. For example, the system may have ASLR enabled, or a patch may change a buffer from the too-small size of 1024 bytes to the just-right size of 4096. In either case, all pointers are likely to have different addresses between the patched and unpatched versions. As a result, previous raw memory snapshot and replay approaches do not work.

To address our twist, TACHYON takes a semantic-based approach to replay only the semantically meaningful information from a syscall in a recording during replay, rather than capturing details like pointer values which may change between patches. Our approach is enabled by three techniques. First, we extend the C

type system to include a full description of the syscall side-effects. The description enables TACHYON to identify semantically meaningful arguments and results in syscalls instead of relying on blind memory differencing. The work to annotate the system calls needs to be performed once, and can be reused for all programs. Second, TACHYON utilizes a rewrite rule system to compare syscall sequences for equivalence and rewrite if necessary. The rewrite system gives the end-user the ability to say when deviations are permitted in a systematic manner. Third, TACHYON uses syscall interposition techniques to record the effects of syscalls on a live program, and replay those effects to simulate syscalls on the tandem program.

Tandem execution makes patch testing in new scenarios possible. For example, a test administrator can run the pre-patch binary live and the post-patch in-tandem on the same system. Any deviation reported is either (a) a bug in the patch, or (b) an exploit against the buggy program that is averted in the patched program, or (c) a permitted change that changes IO behavior. In all cases, the administrator should be informed of the deviation. Alternately, a security-conscious administrator may run the patched version live, noting deviations with the pre-patched version. In either case, the tandem execution achieves live patch testing without duplicating side-effects. The closest current best practices come to similar results requires mirroring a production environment with a pre-production environment, which is expensive and requires significant effort to maintain. We discuss other possible applications such as creating honeypots in § 7.

We have implemented and evaluated TACHYON on a number of security patches, and demonstrated that our techniques can successfully detect deviations. We have also performed micro-benchmarks that show our implementation is efficient with respect to the amount of I/O performed. We show our implementation records full syscall information faster than `strace`, a tracing tool targeted at binary programs, and is efficient in comparison to an untraced run.

Contributions. The main contribution of this paper is techniques for live tandem execution for patch testing. These techniques automate a large part of patch testing, thus reducing the vulnerability window for unpatched systems. In particular:

- We are the first to propose techniques for automated patch testing that address all the above challenges; we have implemented them in TACHYON. We demonstrate where we run both the unpatched and patched binary and use tandem execution to detect deviations. An additional benefit of tandem execution is that it can utilize extra cores for the security purpose of patch testing.

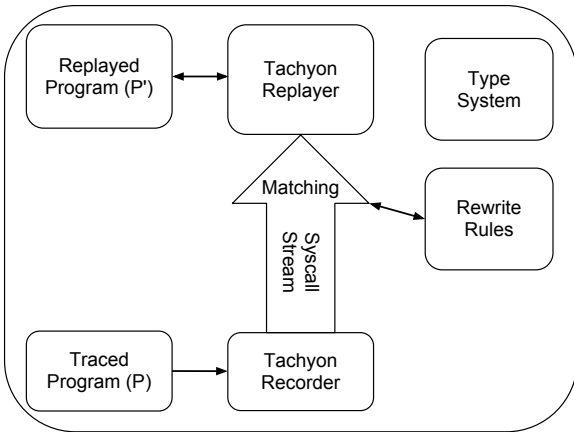


Figure 1: Tachyon System Overview

- We develop a type system that fully encapsulates the side effects of syscalls necessary for replay. Our type system is similar to [11], but does not require source code access or explicit developer cooperation.
- We propose a light-weight rule-based system for checking syscall stream equivalence (and rewriting if necessary) when the sequence of syscalls between the patched and unpatched binary are not exactly the same.
- We have implemented our techniques in TACHYON using Haskell (a type-safe language) and validated the techniques experimentally. Our system is robust enough to handle single-threaded and multi-threaded programs. We evaluate our approach on several real-world patches, as well as synthetic benchmarks, to show the effectiveness and performance of TACHYON.

2 Design of TACHYON

2.1 Overview of TACHYON and Challenges

The overall architecture of TACHYON is shown in Figure 1. In this paper we call the program running live P (e.g., the patched program) and the program running in-tandem with simulated syscalls P' (e.g., the unpatched program). TACHYON is a user-land program that utilizes the Linux ptrace facility to interpose on syscalls issued by P and P' . Like replay schemes, TACHYON has a recorder and a replay module. The recorder records the stream of system calls issued by P and outputs a stream of tuples $\langle C, \vec{I}, \vec{O} \rangle$ where C is the system call number, \vec{I} is a list of inputs to the system call, and \vec{O} is a list of outputs. The replay module interposes on P' , and for each syscall C with arguments \vec{I} made by P' , simulates the OS by returning \vec{O} .

Listing 1: Example patch

```

1 -int fd = open("/tmp/fileA", O_RDONLY);
2 +int fd = open("/tmp/fileB", O_RDWR);
3 -int *storage = malloc(...);
4 -/* ... Do some processing with storage.. */
5 +fstat(fd, statBuf);
6 char* incoming = malloc(chunksize);
7 ssize_t size = read(fd, incoming, chunksize);
8 if (size != -1)
9     write(fdOut, incoming, size);
  
```

Consider the example shown in Listing 1, with the patch difference being displayed in diff style with full context. The first edit changes the file opened from `/tmp/fileA` to `/tmp/fileB`. The next few edits remove an unneeded call to `malloc`, and add `fstat`. The rest of the program is the same. Note that since a `malloc` call was removed, the returned memory chunk for `incoming` will be at a different address, even on systems with a deterministic memory layout. Overall, this patch example illustrates three challenges: patches may change arguments to system calls, may change system calls issued, may change memory allocation patterns, and any of these changes may have effects on subsequent execution.

The above challenges motivate three main requirements of live patch testing as distinguished from a normal replay system. First, instead of offline replay, a live patch testing solution should be *online* where P produces the syscall stream that P' should consume. Second, a live patch tester should not depend upon pointers because absolute memory addresses may change between runs. For example, P' and P may issue calls to `malloc` for different amounts or ASLR may be enabled. Either case prevents patch testing. As a result, we cannot determine \vec{O} by simply diffing the memory state before and after a syscall, as in previous syscall replay schemes [11, 22]. Additionally, memory diffing does not allow us to determine the inputs \vec{I} to system calls. As a good live patch tester should verify the inputs as well, we need some way to extract all inputs of a system call. Without a semantic model, we will be unable to both locate all the relevant components of the input, and to avoid capturing irrelevant components. Thus, we need a semantic model of the inputs. Third, since a patch may remove or add system calls, the live patch testing scheme should allow for the syscall stream to be rewritten during replay. This can be accounted for by allowing rewriting of the tuple stream $\langle C, \vec{I}, \vec{O} \rangle$.

2.2 System Calls and Side-Effects

TACHYON needs to determine what the semantic inputs and outputs to a syscall are in order to record and replay them. Specifically, it needs to (1) determine the types of arguments to a syscall, (2) differentiate input from output, and (3) pointers from the pointed-to data. While existing C syscall prototypes are sufficient for (1), they do not provide enough information for (2) and (3). Consider the `read` syscall declaration:

```
1 ssize_t read(int fd, void *buf, size_t count);
```

This C declaration misses crucial information. First, it gives no clue how the void pointer `buf` works. How big is it? Is it null-terminated? Are the contents relevant before the call, after, or both? We need to answer all these questions in order to copy the appropriate semantic data. We can see that even the assertion that a pointer points at some data before or after the system call is not the case, as with `sbrk` (pointer points at the end of your address space) and `mmap` (one pointer is only a suggestion). `read` is one of the simple cases; several syscalls have complicated dependencies between input and output parameters, as will be discussed later in §3.

TACHYON addresses the challenges associated with understanding the semantics of syscalls by adding type annotations, as described in §3. The TACHYON annotation language is a light-weight dependent type system that says how to parse the inputs and arguments into semantic data at runtime. These type annotations only need to be written once per system call, and are portable across systems with the same syscall signatures.

2.3 Syscall Stream Rewriting

Many patches also change the sequence of syscalls made in addition to the actual parameters. Consider Listing 1. The system call stream when executing the patched program is $\langle \dots, \text{open}, \text{fstat}, \text{read}, \text{write}, \dots \rangle$. However, the call after `open` in the unpatched program is `read`, not `fstat`.

New patched versions often have new system call patterns that cause the program to behave differently at an IO level. It is not possible to tell whether a particular change in system call patterns is valid without a human to validate it. For example, if it turns out the above code is just shifting a few things around and adding a new inconsequential call to `fstat`, then the user may want to ignore the deviation. However, the `fstat` may have been inserted for security, and a deviation may indicate an attack. When opening files in `/tmp/` a common security practice is to then call `fstat` to obtain the user ID and group ID of the file to make sure they are correct in

order to detect race conditions. TACHYON should report the deviation and halt execution in such instances.

Although we cannot automatically decide which deviations matter, TACHYON does *automate finding deviations*, as well as provide a mechanism to ignore such deviations when found to continue testing. The rewriting engine relies upon rules that are created for each patched program that detail how to handle semantic differences. For example, if a system administrator decides the above deviation is inconsequential a rule can be written to ignore the `fstat` call. Alternatively, patch creators could write such rules and distribute them with their patches to aid testing.

2.4 Road Map

In the rest of this paper, we first describe the TACHYON type system in detail. We then discuss how TACHYON rewrites system calls, as well as some common rules we have found in patches we have tested. We next describe our implementation and evaluation. We finally discuss several applications of TACHYON outside automated patch testing.

3 System Call Types

The C function declarations for syscalls do not describe all side-effects. TACHYON proposes an extension to the C type declarations to encode all semantic information necessary to record which parameters are inputs, which are outputs, and how to identify all bytes for each parameter. While this problem has been attacked before [11], our particular needs are different due to the binary nature of our approach, as we discuss in §9.

The TACHYON type system takes advantage of the user-space/kernel-space barrier for interposition. The barrier provides a clean separation that can be monitored without requiring source to the program. In addition, the barrier allows TACHYON to not monitor internal kernel state. The reason is that the only way P' and P can interact with the underlying system is via TACHYON, and TACHYON's mechanism ensures that both programs see an identical state. This is a vital complexity reduction.

TACHYON uses a limited form of lightweight dependent types (types which depend on values). Our lightweight use avoids pitfalls such as undecidability normally associated with dependent types. In the rest of this section, we first provide an intuition on the issues and how dependent types are used, and then describe the full system.

3.1 Intuition

A basic approach for recording syscalls is to decorate C types with information about which parameters should be treated as inputs, outputs, or both. We call such annotations the IO class for each parameter. In order to specify how to copy output parameters, we also need to know the size of their values. The size information is needed because we need to copy all output bytes from `buf` in the monitored program P to the address space of P' . For example, we could imagine annotating the `read` call as:

```
1 ssize_t read(int fd, void output* buf{ret},
              size_t count);
```

The parameter `buf` has been annotated as an output parameter, thus should be copied and replayed to P' . The annotation also specifies that `ret` bytes should be copied, where `ret` is the return value.

Unfortunately, such simple annotations are insufficient with many data structures, such as vectors. A prime example of a difficult system call is `readv`, which provides vectored reads of a file descriptor. Its C type declaration is:

```
1 ssize_t readv(int fd, const struct iovec *iov,
               int iovcnt);
2 struct iovec {
3     void *iov_base; /* Starting address */
4     size_t iov_len; /* Num bytes in iov_base */
5 };
```

The main issue demonstrated is that a complete description of the IO behavior of parameters may refer to other parameters. The `iov_base` length is determined by `iov_len`, and the total number of `iov` items is given by `iovcnt`. `readv` is not alone: it has many friends such as `writev`, `preadv`, and `pwritev`. In order to handle such cases, we need a type system that allows us to express *relationships* between parameter values.

TACHYON uses lightweight dependent types that can express relationships between the value of one parameter and the value of another. We view types as a tree, and use dependent types to walk the tree to determine a value.

The types allow us to walk from the top of the tree, or from the current parameter. In TACHYON, we specify `readv` as:

```
1 ssize_t readv(int fd, const struct inputoutput
               iovecin *iov{iovcnt}, int iovcnt);
2 struct iovecin {
3     void input *iov_base{undo(self).iov_len};
4     size_t iov_len;
5 }
```

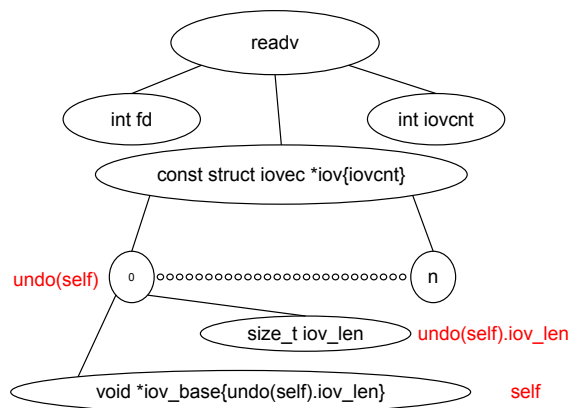


Figure 2: A lookup in action

We now call the struct `iovecin`, because while both `readv` and `writev` take an `iovec`, they are used differently, and so are assigned differing types (specifically, in one case the buffers inside are output, while in the other they are input). The only new annotations compared to before are `undo` and `self`, which are used to walk the type tree to reference other fields. The semantic meaning is that `iov_base` is `iov_len` bytes. `self` refers to the location at which the current value is being read from. `undo` simply says to step back along whatever indexing step was done to get there. In this case, this means that `self` represents the tree traversal up through that instance of `iov_base`. The “undo” brings us up a level, to be looking at the struct. Then, we index the struct to `iov_len` and are done. Figure 2 graphically shows the type tree for `readv` and how the syntax expresses fields in the tree.

3.2 The Tachyon Type System

The full TACHYON dependent type system is shown in Figure 3, and is taken directly from the TACHYON source code in Haskell. The language is similar to BNF, where non-terminals are to the left of the equal sign, and brackets denote a list (e.g., `[A]` is a list with elements of type `A`).

In TACHYON’s language, `IOC` represents an IO class, that is, whether the pointer is input, output, or both. `T` represents some form of termination, to allow us to include null-terminated data. `NT` is for null-terminated data; `UT` is for unterminated data. If a pointer is null-terminated, reading will cease when a 0 is hit, if this happens prior to the end of the buffer. The index operation is used on both arrays and structs, where the i ’th index refers to the i ’th field (counting from 0).

The types available are

- Small - These correspond to basic integer C types,


```

1 data SysSig = SysSig Type [Type]
2 data Type = Small Int
3           | Struct [Type]
4           | Ptr IOC Type Bound T
5 data T = NT | UT
6 data IOC = In | Out | InOut
7 data Bound = Const Int | Lookup Lookup
8 data Lookup = Ret | Arg Int | Index Int Lookup
           | Self | Undo Lookup

```

Figure 3: The TACHYON annotation language

like `char` or `long`, and indicate values that should not be treated as pointers. The type parameter is the number of bytes of the type, e.g., `Small(1)` is a 1-byte value corresponding to a `char`.

- **Struct** - an aggregate of other types. Note that previous replay work treated such types as raw buffers because they could determine size by simply diffing memory before and after a syscall. In live replay, we explicitly lay out all fields because the underlying types may yield further information.
- **Ptr** - a pointer annotated with an IO class, the type of element it is pointing to, a way to tell how many elements it points to, and whether or not it respects a null termination convention.

We introduce the concept of a “lookup”. This is just a series of steps that can be performed from either the arguments of a function in the case of an input or in/out class pointer, or the arguments and return value in the case of an output pointer, to arrive at a memory location or register. This is demonstrated in Figure 2. The `Ret` and `Arg` constructs for a lookup are used to allow us to reference the return value or various arguments in a system call, respectively. This is just the generalization of the tree walking described earlier.

Given this, the encoding of a bound as either a constant or a lookup is rather natural. It is the use of this bound that makes us lightly dependently typed—the type depends on the data in question.

Finally, we can build the fundamental structure all this is for—the system signature. A system signature, indicated by `SysSig`, is what is assigned to each system call in order to allow the tracer to record and play back its effects. The first parameter is the type of the return value, and the second is a list of the types of its arguments.

Type Checking TACHYON Declarations. The TACHYON types need only be written once for each system call, and can be reused for any program. However, since they are written manually, we would like to prevent mistakes. In order to achieve this, TACHYON

also provides type-checking to make sure the annotations make sense. In particular, TACHYON checks:

1. Bounds are numbers, not pointers or something else.
2. Bounds use only information which is available for the IO class of the pointer (e.g., input class may not use the return value as a size).
3. Output pointers do not contain structure; they are raw data.
4. Types are potentially compatible with the original C type.

These checks ensure annotations which are usable, self-consistent, and match the C type.

4 System Call Stream Deviation Detection and Rewriting

Patches often add, delete, or modify new system calls in the original buggy program. Our example in Listing 1 shows all three cases. When the streams of syscalls differ, then the two programs are semantically different. While this means we cannot automatically tell if the differences are meaningful, we can (a) automatically detect deviations and (b) rewrite deviations when informed by the user that the semantic differences are permitted. The heart of detection and rewriting is TACHYON’s syscall stream matching and rewriting engine.

4.1 Stream Matching

TACHYON uses a rule-based system for rewriting system call streams during execution, designed to be employed by a user of the tracing software to explain to the system what behaviors it should consider equivalent. The rules must consume a sequence of system calls by P , and produce a corresponding set of system calls for P' to make in order to allow for writing call results into P' and checking that P' indeed matches the particular equivalence rule.

As we execute, we have two streams of tuples. TACHYON represents the stream from P as $\langle C_i, \vec{I}_i, \vec{O}_i \rangle$, and the stream from P' as $\langle C'_i, \vec{I}'_i, \vec{O}'_i \rangle$. The easy case is when the two programs are semantically equivalent by issuing the same system calls, i.e., $\forall i : C_i = C'_i \wedge \vec{I}_i = \vec{I}'_i$. In this case no rule is needed, and TACHYON will send the corresponding \vec{O}_i for each \vec{I}_i to P' .

Any time the syscall input arguments do not line up, TACHYON reports a semantic deviation. In order to permit some deviations, TACHYON provides the ability to rewrite the system call stream. The rewrite engine takes in a set of rewriting rules f . Each rewrite rule f_k is a function which takes in $\langle C_i, \vec{I}_i, \vec{O}_i \rangle$ and $\langle C'_i, \vec{I}'_i \rangle$. The rule uses pattern matching to decide if it applies, and if so, returns a pair of equivalent syscall streams to perform a

substitution with. After a match, the stream continues to be consumed by the simulated program P' .

The overall mechanism can be used for:

- Determining roughly equivalent syscalls, e.g., many small writes being patched to be one big rewrite.
- Ignoring syscalls, e.g., the P program issues a call that is not needed by P' .
- Limited reordering, e.g., allowing for syscalls to be switched.

4.2 Rewriting Rules

Each rewrite rule f takes a system call (the one made by P) and the input to a potential system call made by P' , and returns a substitution in the stream. The substitution is implemented as a pair of lists, where the left list indicates the syscalls consumed by the rule, and the right list indicates the corresponding substitution produced by the rule. The type signature for f in TACHYON is:

Syscall \rightarrow SysReq \rightarrow Maybe ([Syscall], [Syscall])

where the “Maybe” indicates that the rule may also return that no substitution was performed.

The rewriting rules are pure functions, which means they have no access to outside resources like the current syscall stream or application state. By being pure we ensure that rewrite rules can be applied in any order. In addition, it ensures that the rule engine itself will not continually accumulate state, i.e., while individual rewrites may take substantial space, the space used will remain constant in the number of system calls which have gone through, which is vital to an online system.

During execution, the matching engine maintains a queue of syscalls executed by the live program P . Suppose the queue contains any syscall x that is not `write`, but the simulated program P' issues a `write` syscall. The simplest rule is to ignore the `write`. This is accomplished by adding a `write` to the queue before x . When the matching engine re-examines the queue, it will match the still-pending `write` to the one in the queue, and not report a deviation.

In TACHYON, the rule is written as:

```

1 ignoreWrite :: Syscall
2   -> SysReq
3   -> Maybe ([Syscall], [Syscall])
4 ignoreWrite x (Write 2 buf sz) =
5   Just ([x],
6         [Syscall (Write 2 buf sz) sz, x])
7 ignoreWrite _ _ = Nothing

```

This rule fires when line 4 is matched. This occurs when P issues a syscall x that doesn't match P' 's syscall `write`. On line 5 the rule directs it to consume whatever

is on the stream at the moment, and replace it on line 6 with the stream of `Write` followed by x . This can be thought of as “faking” the call for `Write` to the stream matcher so that it does not report a deviation. On line 7, we catch the case where our conditions are not met, and indicate we did not modify the stream.

The simple, no-look-behind method of replaying with this equivalence is to replay the stream normally until a match fails. At this point, the two syscalls that failed to match are fed into all rewrite rules, and their replacement list for the original stream is checked. If there is still more than one rewrite rule remaining, one is chosen arbitrarily. In future work, checkpointing could be used here to allow for the ability to rewind if the wrong replacement was chosen. In practice, the rules we tested have only yielded one matching rewrite.

A more complex example is what we call write splitting, which occurs when a larger write in the original program is translated into two smaller writes in the replayed program. This is useful if the buffer size used in a transmission was decreased during the patch, as it allows for a roughly equivalent operation—writing one part of the message, then the other—to be treated the same as the original system call writing the entire message. A concrete example would be the difference between the program fragments:

```

1 #define CHUNK 4096
2 #define CHUNK 1024
3 while(buf < end) {
4   buf += write(fd, buf, min(end - buf,
5     CHUNK));
6 }

```

In the patched case here, we will see on average 4 times as many system calls, but fundamentally, the same thing is happening. A rewriting rule for write splitting says that a sequence of previous writes can be used to fill on big write request:

```

1 writeSplit :: Syscall
2   -> SysReq
3   -> Maybe ([Syscall], [Syscall])
4 writeSplit (Syscall (Write fd buf sz0) sz)
5   --sz is return size
6   (Write fd' buf' sz')
7 | (fd == fd')
8 && (sz' < sz)
9 && ((take sz' buf) == buf')
10 = Just ([Syscall (Write fd buf sz0) sz],
11         [Syscall (Write fd' buf' sz') sz',
12           Syscall (Write fd' (drop sz' buf)
13             (sz0 - sz'))
14           (sz - sz')])
14 writeSplit _ _ = Nothing

```

This rule states that if we have a write call in the original stream, and the replayed program is trying to make a non-matching write call, but it matches on the file descriptor, and has a smaller size, and the write it is trying to make is a prefix of the original write, then we can replace the original write with two smaller writes, the first of which is the target write, and the second of which is set up to represent the rest of the write.

In line 6, we do a sanity check that the file descriptors we are writing to are the same, followed by a similar check in line 7 that the request from P' has a smaller size than the original from P . Finally, in line 8 we make sure that what is trying to be written is a prefix of the appropriate length of the original write. Given these conditions, we know that we can provide a replacement rule which will allow the trace to proceed. In line 9, it tells its caller to consume the most recent call, asserting that it matches the call passed into us. In line 10, we see the first call that is going to end up on the new stream, which matches the input vector we've received from P' , and so will allow the trace to continue. In the 11,12, and 13, the function returns an additional item for the system call stream, which represents the rest of the write that has been split. In 14, we catch the case where we don't apply, and simply return no pattern.

5 Architecture

TACHYON is built to target Linux for the x86-64 architecture via the `ptrace` call, written in Haskell. An abstraction barrier is in place around `ptrace`, to ensure the technique's generality and portability to other systems with similarly powered tracing libraries. Haskell was chosen for abstract data type support, multi-OS portability, relative speed, and a monadic abstraction layer that proved useful for our tracing environment.

ptrace TACHYON uses the `ptrace` system call to accomplish system interposition. Use of `ptrace` starts with initialization, in which options are set and the remote process either volunteers itself for tracing, or is traced via a command to attach to its pid. Then, `wait` and `wait4` are used in an event loop to get status information about processes (which are paused when generating one of these messages) and are then resumed at a later time.

The wakeup and sleep powers are implemented by selectively choosing to not resume or resume threads at system call boundaries. While this only enables us to support putting the currently running thread to sleep, we never needed to stop any thread for which we were not currently processing an event.

Trace Abstraction. The trace abstraction layer is designed to expose only primitives we believe to be constructible on all platforms for portability purposes. Additionally, the interface was higher level than the tracing interface directly available on most platforms, enabling easier authoring of the code. Fundamentally, a handler is provided for events which the tracing interface detects and sends back. The potential events currently supported are pre/post syscall, and process split. Available to the callback is the ability to put threads to sleep, wake them up, read and write registers, and read or write memory in the target process.

Multithreading tolerance. Up until now, we have considered only programs using one thread. However, many modern programs use multiple threads in their normal operation. For example, `curl` in § 6 uses them during DNS lookup.

To deal with threads, TACHYON employs techniques inspired by the field of deterministic multithreading (DMT)[2, 3, 15]. A DMT mechanism is one that makes a program insensitive to the scheduler as an input. That is, given the same inputs other than kernel scheduler action, it will yield the same outputs.

In TACHYON, we enforce an ordering over system call events. We differentiate from a mismatched system call in need of rewriting and a thread being early or late by choosing to block the thread if the thread IDs on the syscalls don't match, and invoke the rewrite engine if they do match, but the system calls don't. This forms a looser notion of consistency than is used in regular DMT, but is sufficient for our purposes. However, applying a real DMT system in addition to our techniques would likely yield an even more robust treatment of threading able to deal with shared memory data transfers and other such intricacies, as we discuss in § 8.

Special Syscalls. While in general the simulated application P' uses the effects of syscalls from the live P , TACHYON does have a few exceptions. The exceptions occur when a syscall result from P cannot be emulated in P' . This usually occurs when there is something which is part of the thread life cycle or virtual memory system, which are not facilities that can be directly accessed by TACHYON. Luckily, there are only a limited number of cases.

The first is `sbrk()`, which is the syscall responsible for dynamic memory allocation. Luckily, this particular call can be passed through, as it does not modify OS state, it only serves to modify the process's VM system.

When a `clone()` occurs, we match input arguments like any other system call, and then allow it through. `ptrace` feeds us an event notifying us as soon as the new thread exists, and pauses that thread before it can do

anything so that we can attach to it. This event is also part of the synchronized system call stream. TACHYON then registers that new thread and its pair between emulated tid and real tid, and proceeds with normal operation.

The `exit_group()` system call works like any other, except that it acts as an end-of trace marker. We currently only accept full application exit, in which all threads are simultaneously terminated, but there is no reason our techniques could not be extended to individual thread destruction.

The most complicated is `mmap()`, which maps a file or device into memory. Our implementation depends upon the operations performed on the region. The read/write case would be extremely expensive to monitor, as all writes are effects, so we would have to interpose on every memory write to that page, and so this case is disallowed entirely. It might be possible to deal with writable mapped files or shared regions shared outside the program, using page faults and slow execution, or some form of snapshot trick. We leave this as future work.

In the read only case however, the translation is straightforward. Rather than issuing the mapping as requested, we instead simply ask for a private mapping of the same size this thread received during recording, and fill that buffer with the contents that memory contained after the initial map. Private mappings of anonymous memory are also easy to support, and are be simply passed through. Finally, in the case of shared memory, we can allow it if it is both anonymous and our multi-threading system is in place. This could be extended to allow for non-anonymous shared memory by spawning fake file descriptors to the region, but is left as future work.

Other system calls for which we would need to add special implementation to allow them to be serviced by the kernel, but still safely matched include `munmap`, `mprotect`, `fork`, `sigaction`, `sigreturn`, and `exit`. These were unneeded for our test cases and were not implemented.

6 Evaluation

We have evaluated TACHYON with respect to three main questions. First, can TACHYON detect deviations where the patched program is semantically different than the unpatched, and how hard is it to write rules to ignore deviations that do not matter? Second, what is the performance factors for TACHYON, including best and worst case settings? Third, what is the performance on real programs? In this section, we describe our results.

Program	Issue ID	Description
cURL	CVE-2011-2192	Improper key delegation
mplayer	EDB-ID 11792	Table index out of bounds
php5	CVE-2012-0832	Bad Argument handling
php5	CVE-2011-1938	Buffer Overflow
ncompress	CVE-2001-1413	Buffer overflow
htget	CVE-2004-0852	Buffer overflow
gs	CVE-2010-1869	Buffer Overflow
glftpd	EDB-ID-476	Buffer Overflow
socat	CVE-2004-1484	Format String
corehttp	CVE-2009-3586	Off-by-one Buffer

Figure 4: Successfully Detected Deviations for Security Patches

6.1 Detecting Deviations

To test the effectiveness of TACHYON, we used it on real patches to detect known deviations. The patches we tested are shown in Figure 4. In this experiment, we tested the program on normal inputs, and verified that TACHYON did not report a deviation. We then tested on inputs that triggered known deviations, e.g., exploits in the original program or bugs in the patch.

For cURL, the patched vulnerability was an information disclosure bug. In the unpatched version, Kerberos credentials were (accidentally) forwarded instead of just a proof the user was authorized. We verified that the unpatched program would send credentials, and the patched program did not. In order to test the patch and allow normal operation of safe inputs, we had to write two rules for cURL that totaled 11 lines. The rules were necessary because cURL added a non-security feature that affected file descriptors in their patch.

CVE-2011-4885 addresses a problem in PHP where hash collisions are easy to find, which can be used to launch a remote denial of service attack. TACHYON required no rewrite rules to run the patch on safe inputs. The patch, however, broke the argument handling for arrays after loading many arguments. CVE-2012-083 addressed this problem. For CVE-2012-083, we again required no rewrite rules for safe inputs.

CVE-2011-4885 and CVE-2012-0832 demonstrate a patch that is broken, and provide motivation for TACHYON. Since CVE-2011-4885 fixed a purported vulnerability, it should be applied immediately. However, after applying CVE-2011-4885, a *new* vulnerability is introduced. TACHYON detects those new exploits as deviations immediately. In particular, we checked exploits (addressed in CVE-2012-0832) that were unknown in

2011, and verified that they caused detected deviations. Thus, if an administrator had been running TACHYON, and immediately applied the patch, they would detect exploits immediately against the vulnerability introduced.

The EDB-ID 11792, CVE-2001-1412, and CVE-2004-0852 all patch typical security bugs by adding in-line checks. These checks did not change the system call pattern or arguments, thus no rules were needed for patch testing.

For CVE-2010-1869, `gs`'s memory problems required a rewrite rule to admit additional or skipped calls to `brk`. 8 lines were required for these rewrite rules. Three lines were required for EDB-ID-476 to allow for rewriting of the format of a usage message. Four were required to deal with the new `lstats` in the patch for CVE-2009-3586.

6.2 False Positive Testing

To show that TACHYON is fairly precise, we tested it on the most recent 207 patches to `coreutils`. (The number 207 was chosen because that was how far backwards we could go easily with an automated building system.) From this, we found that in 18 cases out of 1656 executions, a deviation was reported, or TACHYON crashed. Looking at the output, 16 of these were TACHYON bugs, but are not systematic, so that re-running the test produced correct results. 2 of these were actual deviations. In the first, a call to `fadvise()` was introduced into `cp`. An equivalence can be reached with a one-line rewrite rule. In the second, a buffer size was changed. The read/write splitting/coalescing rules described earlier in this paper allow an equivalence to be reached. Overall, this indicates that while TACHYON is not perfectly bug free, it never reported a deviation when one had not happened, and deviations that should be acceptable could be easily expressed in the rule system.

We also ran TACHYON on patches for two common utilities with no known vulnerabilities: `/bin/ls` and `/bin/cat`, and used them interactively. In the one month testing period, TACHYON was able to use tandem execution on these utilities for normal day-to-day use with no perceived slowdown. Further, TACHYON reported no deviations (i.e., had no false positives).

6.3 Micro-benchmarks

TACHYON has three main sources of overhead: our approach to syscall interposition, transferring bytes from the source to the sync application, and running both P' and P in tandem. Overall we measured a linear overhead for both data transfer and system calls, and 0% of CPU time, as detailed below.

Syscall interposition overhead. TACHYON is a user-space system call interposition scheme, which imposes additional context switches but provides for a clean separation of interposition, kernel, and user-space application. Our user-space interposition has 4 context switches per call issued by the live application P . For each syscall in P , TACHYON context switches from P to the kernel, from the kernel to TACHYON, from TACHYON back to the kernel, and finally back to P . Normal operation only has two context switches: from user to kernel space, and back again.

In order to test the effect of these two extra switches, we wrote a simple program that executed `getpid()` in a tight loop. We were sure to call the system call directly, as the standard version of `getpid()` in C actually caches its result.

Data copy overhead. TACHYON needs to copy the output data from P to P' . A first naive implementation actually incurred 6 copies. TACHYON originally copied output data from P into TACHYON for syscall rewriting, and then copied it to P' . Each copy between systems was actually two copies: one from the user-space into kernel space, and one from kernel-space into user-space. This led to a huge slowdown in early benchmarks (over 6x). To reduce this, we added the ability to the Linux kernel to map a remote process's memory via `/proc/pid/mem` under most situations, making the normal case of reading from the remote process only have one copy.

In Figure 5, we see the overhead is in a linear relationship. The overhead here is the total overhead time for the system. While they make a difference for small data transfers, they are rapidly dominated. This can be seen by the rapid transition to a tight grouping around a linear relationship.

In Figure 6, we again observe a nice linear relationship, showing no residual effects on performance from processing a system call. It shows that it takes less than a second of overhead to process 60,000 syscalls.

When varying the CPU load of the traced program, no noticeable difference in execution time was noticed, as we do not intercept regular computation, only system calls.

Given this, if it is known how many system calls are used, how much data is being transferred, and how much time is being spent on the CPU, we can model how long a given workload would take under our tracer. TACHYON previously incurred a large number of copies and control transfers to move buffers around in comparison with the register fetching it does for simple system calls, and the remote memory fetch path is not optimized in the OS. However, a kernel patch allowing for `mmap` to be used on the special file `/proc/pid/mem` considerably ameliorate this, resulting in the new statistics above.

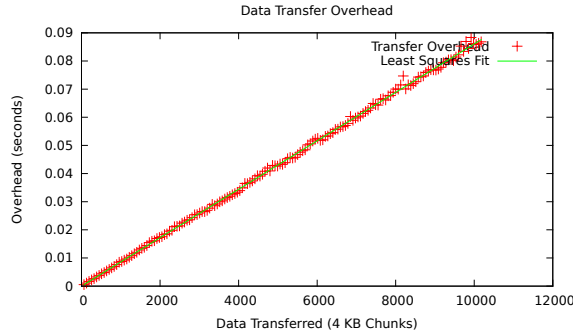


Figure 5: Data Transfer Overhead

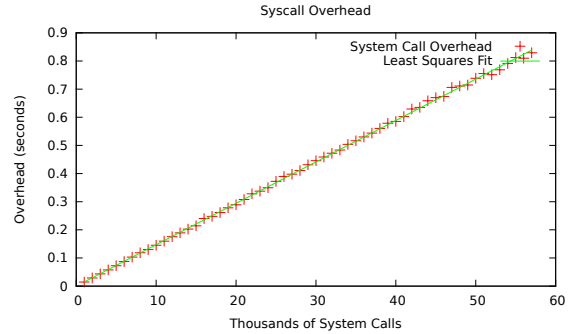


Figure 6: Syscall Overhead

Tandem execution CPU time. The final source of overhead is in CPU operations. Since TACHYON sleeps when system calls are not being issued, it does not slow down applications that are CPU-intensive. This is a significant advantage over instruction-level interposition tools such as Pin [16] and Valgrind [21], which typically suffer at least several times overhead.

However, we are running both P' and P . We verified that TACHYON can utilize independent cores to run both programs with no additional overhead (other than the memory transfer and syscall overhead measured above). Thus, we conclude that TACHYON can utilize multi-core to test patches.

Efficiency on real programs. To test the efficiency of TACHYON interposition, we measured its tandem execution against `strace` and `gdb`'s reverse execution. `strace` is a tool built on top of `ptrace` used to monitor system calls. `gdb` allows programs to back-step through operations.¹ Note these tools have different goals than TACHYON; we only use them to evaluate performance.

`gdb`'s replay mechanism derived from its reversible debugging support. However, it proved wholly unsuitable for regions of more than a few instructions. Due to a lack of SSE support, `memcpy` would be improperly rewind and replayed. Additionally, even then the recording overhead was more than 100x native execution, and built up a huge memory data structure, making it impractical to benchmark.

The results compared to `strace` are shown in Figure 7. Overall, TACHYON was faster, sometimes by a large margin, than a comparable syscall interposition scheme. This is partially because TACHYON can and does process some of its overhead while the traced program is doing work, but mostly due to the massively

¹We were unable to test against what is likely the most similar system, R2 [11], as it is both Windows only and requires build-time support (as well as not being public).

Program	Load	TACHYON	strace
compress	32M Random	1.41	19.78
primegaps	First 35	1.00	1.00
mencoder	h264	1.07	1.12

Figure 7: Tracing Performance (relative to native execution)

improved facility for retrieving remote memory via our patch to `mmap /proc/pid/mem`.

Web Server Tests. We also tested the throughput of `lighttpd` and `thttpd` when monitored under TACHYON. In this test, we use ApacheBench (`ab`) configured to make 1000 requests in two threads, downloading 4096 byte web page. We ran the experiment in two scenarios: on localhost and across the Internet.

In the network experiment, the web servers ran at one university, and requests were made from another university on the opposite US coast. There was no detectable degradation for `thttpd`, and only about a 30% slowdown for `lighttpd`. Essentially, what this shows is that while the system does not deal well with applications whose progress is primarily based on the rate at which they can issue system calls, when we move closer to a real deployment, applications do not tend to have that as their primary limiting factor.

In the second experiment, we ran `ab` on localhost. This is a worst-case test because both web servers spin in a tight loop on a syscall (`lighttpd` spins on `epoll`, and `thttpd` on `poll`). This creates a pathological case for TACHYON, because the application spends most of its time neither doing IO, nor doing computation, but instead spends most of its time moving across the system call barrier.

TACHYON took 8.9 times longer on `lighttpd`

(throughput decreased to 10% of original values) and 14.2 times longer on `thttpd` (throughput decreased to 7% of original values).

To round things out, we also ran a test over a few hops on the local network. As expected, intermediate results were measured to be between the two, with 1.17 times untraced completion to complete the test with `thttpd`, and 3.72 times untraced completion to complete the test with `lighttpd`.

With a more real network (or a more complicated webapp), we can see the slowdown is lessened. With a real network, `epoll` will spend more time waiting, diminishing the perceived effects.

7 Discussion

Other Patch Testing Scenarios. While throughout this paper we have focused on online patch testing where the patched version is run live, we could also run the unpatched version live. We note that the live program can continue executing after a deviation, but currently the syscall sync application cannot. Thus, by running the patched live, we are assuming that after a deviation the right thing is to continue executing the patched version. However, by running the unpatched version live, we can check for incompatibilities while allowing for the original program to continue executing after a deviation.

Honeypots. TACHYON can also be used as a type of lightweight honeypot. Let P be a patch for a security vulnerability, and P' be the vulnerable program. Observe that P and P' differ on exploits by definition. By running P and P' in-tandem, TACHYON will report a deviation on attacks.

A clever approach to running a honeypot is to run P as the live program, with P' as the sync. In this setting an attacker only seeing the buggy program. TACHYON will report attacks, e.g., by logging a deviation when shellcode tries to execute `/bin/sh`. However, the system is safe from a real compromise because TACHYON can be configured to abort execution after the deviation.

Debugging. One of the most difficult to debug classes of bugs is commonly known as heisenbugs. These are bugs which will seemingly randomly occur or not occur with all of the inputs the programmer knows about held constant. These traces, and the associated replay mechanism, provide a way to step through the program in a completely deterministic way, so that once a heisenbug has been caught with tracing on, it has been captured and the sequence leading to it can be carefully explored and debugged. As we capture all inputs, this also makes it possible for the programmer to debug a crash that took

place on another machine, without having to try to replicate the OS state to reproduce the crash.

Efficiency. Recall TACHYON uses user-land syscall interposition, and has our approach to syscall interposition as its primary source of overhead. Currently, interposing on each system call on the live program requires 4 context switches. TACHYON context switches from P to the kernel, from the kernel to TACHYON, from TACHYON back to the kernel, and finally back to P . Normal operation only has two context switches: from user to kernel space, and back again. A kernel-space interposition scheme would also have only two switches.

Recall from § 6 the overhead from copying data is almost linear in the amount data transferred between P and P' . A basic in-kernel approach would still have a linear overhead (since data has to be copied into both virtual memory spaces), but likely with a smaller constant factor.

Our user-land approach was chosen because it offers a clean separation of functionality, isn't kernel dependent, and offers an easier development environment. Moving the system call interposition into the kernel would not have these advantages, but would likely improve performance. We leave further study of in-kernel tandem execution schemes as future work.

8 Limitations and Future Work

TACHYON could be extended to provide better determinism for shared memory. At the moment, because TACHYON does not schedule individual memory operations, multithreaded programs which have concurrency bugs could run differently under TACHYON. (Non-concurrency bugs are not a problem.) One approach would be to incorporate recent advances in DMT, e.g., [2, 3, 8, 15] into TACHYON. This would also allow for effectful shared memory.

TACHYON currently does not support virtual dynamically-linked shared objects (vDSO), a section of kernel memory mapped into the user-space process to allow for more efficient calls. Unfortunately, some system calls made through a vDSO do not trigger the `ptrace` trap. However, vDSOs are known to provide increased efficiency, so being able to trap that interface could be an improvement, and limit host system modification.

9 Related Work

Our approach is motivated by existing replay systems. At a high level, previous work in this area has focused on system call replay (e.g., [11, 22]), virtual-machine level

replay (e.g., [12, 17, 24]), and instruction-level replay schemes (e.g., [1, 10, 22]). These systems address the related but different problem of replay against the same binary, e.g., for debugging, while we want to replay to a different binary for patch testing.

Delta Execution [23] uses a similar insight to us for testing, namely that patches tend to change very little, and the majority of the program should remain the same. To accomplish this, they structure execution so that it splits every place in which the patch modified the code, and attempts to merge the execution afterwards, checking that the overall state change during the split matched appropriately. Their approach has the additional advantage of avoiding duplicate computation. TACHYON differentiates itself from this work primarily in its generality; specifically, it works at a binary-only level, it allows matching global effects (e.g. heap changes and IO), can be configured to allow specific non-matching global effects, and allows for structure size changing. Fundamentally, Delta Execution attacks the problem at the level of matching internal state, while TACHYON attacks it from the point of view of observational equivalence from the outside world.

Where Delta Execution places their consistency level inside the application state, which is more specific than us, Capo [20] attacks it from the point of what signals are coming into and going out of the computer. This wins them several benefits, namely the ability to deal with fewer effects and a lesser need for a rewrite or matching system (for example, coalescing or splitting reads or writes is free). Unfortunately, this system also needs specialized hardware and in-kernel code to operate. While we have used kernel code to accelerate TACHYON under Linux, it is not required or fundamental to the technique.

Like TACHYON, R2 [11] designed a type system to express all side-effects, but for the purpose of describing intercepted APIs at the source level. R2 [11] differs from TACHYON in that it targets developers, interposes at the function level (thus requires source), and replays recorded syscalls against the same binary. Although the problem setting is different, there are numerous good ideas that could be borrowed for live replay if source was available. For example, R2 proposed analyzing the call graph to find efficient cut points at which to perform interposition, while we always interpose at every syscall. Unfortunately, we at the binary only level cannot easily prove that a set of interposition points form a complete cut. Additionally, R2's annotation language is actually too powerful, and allows for the expression of types that we cannot appropriately interact with at a binary-only level without compile-time assistance, as it allowed for the computation of arbitrary expressions. We instead limited ourselves to navigating trees of pointers, which turns out to be sufficient for the vast majority

of system calls.

We record system calls and discover divergences when system call requests do not line up. Another approach would be to perform replay at the instruction level, which would be useful for pinpointing the first point of divergence. This could be done by augmenting instruction-level replay systems like PinPlay [22], and gdb[10] 7.0 and above to take into account differences in memory layout. In undodb [1], memory snapshots and individually optimized system calls are used to accomplish reverse execution.

Alternatively, one could utilize VM playback mechanisms [9, 12, 24] to simulate patches at the whole-machine level. However, testing then requires accurately replaying very low level events. We chose system calls over instruction level or whole-machine level because system call interposition is significantly cheaper, thus more amenable to end-user deployment scenarios. Additionally, rewriting system calls is much more reasonable than rewriting low level events.

Another recent idea in interacting with multiple programs which should meet the same specification, similar to our patched and unpatched pair, is the idea of N-Variant systems [7]. It intends to increase reliability by forcing any exploit or otherwise bad input sent as input to cause the same bad effect in other versions of the software in order to actually occur. There are some similarities here, but our techniques are aimed at fundamentally differing process images, while theirs are aimed at the same underlying code put together in different ways.

Our system does not find new inputs for patch testing. While we assume live or recorded inputs, one could replay both systems on automatically generated inputs as well. For example, we could use test cases produced by automated systems such as KLEE [6], BitBlaze [4], and BAP [13].

Brumley et al. have previously proposed deviation detection at the binary level [5]. The main goal in this work is to automatically *generate* likely deviations, which is a different problem than tandem execution. Once a candidate deviation is generated, the deviation was manually validated (Section 3.3 [5]). Our approach could be used to validate deviations automatically at the syscall level.

10 Conclusion

In this paper, we presented TACHYON, a system for testing binary-only patches on real inputs in a live system. We have demonstrated an efficient way to describe interface boundaries on C-style declarations using a lightweight dependent type system. Our experiments show TACHYON is able to automatically detect deviations on real programs. We also suggest our techniques may apply to other problem domains, such as honeypots.

References

- [1] undodb. <http://undo-software.com>, Nov. 2011.
- [2] AVIRAM, A., WENG, S.-C., HU, S., AND FORD, B. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–16.
- [3] BERGAN, T., ANDERSON, O., DEVIETTI, J., CEZE, L., AND GROSSMAN, D. Coredet: a compiler and runtime system for deterministic multithreaded execution. *SIGARCH Comput. Archit. News* 38 (March 2010), 53–64.
- [4] BitBlaze binary analysis project. <http://bitblaze.cs.berkeley.edu>, 2007.
- [5] BRUMLEY, D., CABALLERO, J., LIANG, Z., NEWSOME, J., AND SONG, D. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of the USENIX Security Symposium* (Boston, MA, Aug. 2007).
- [6] CADAR, C., DUNBAR, D., AND ENGLER, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation* (2008).
- [7] COX, B., EVANS, D., FILIPI, A., ROWANHILL, J., AND HU, W. N-variant systems: A secretless framework for security through diversity. In *USENIX Security Symposium* (2006), no. August, pp. 1–16.
- [8] CUI, H., WU, J., GALLAGHER, J., GUO, H., AND YANG, J. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 337–351.
- [9] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)* (2002), pp. 211–224.
- [10] FSF. *Documentation for GDB*, 7.3.1 ed., Sept. 2011.
- [11] GUO, Z., WANG, X., TANG, J., LIU, X., XU, Z., WU, M., KAASHOEK, M. F., AND ZHANG, Z. R2: An application-level kernel for record and replay. In *OSDI* (2008), pp. 193–208.
- [12] HERROD, S. The amazing vm record/replay feature in vmware workstation 6. <http://communities.vmware.com/community/vmtn/cto/steve/blog/2007/04/18/the-amazing-vm-recordreplay-feature-in-vmware-workstation-6>, Apr. 2007.
- [13] JAGER, I., AVGERINOS, T., SCHWARTZ, E., AND BRUMLEY, D. BAP: A binary analysis platform. In *Proceedings of the Conference on Computer Aided Verification* (2011).
- [14] LANE, B. A. The USAF standard desktop configuration (SDC). download.microsoft.com, June 2007.
- [15] LIU, T., CURTSINGER, C., AND BERGER, E. D. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 327–336.
- [16] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (June 2005).
- [17] MAGNUSSON, P. S., CHRISTENSSON, M., ESKILSON, J., FORSGREN, D., HÄLLBERG, G., HÖGBERG, J., LARSSON, F., MOESTEDT, A., AND WERNER, B. Simics: A full system simulation platform. *Computer* 35 (February 2002), 50–58.
- [18] MELL, P., BERGERON, T., AND HENNING, D. *Creating a Patch and Vulnerability Management Program*. National Institution of Standards and Technology, Nov. 2005.
- [19] MICROSOFT. Microsoft security intelligence report vol. 11. Tech. rep., Microsoft, 2011.
- [20] MONTESINOS, P., HICKS, M., KING, S. T., AND TORRELLAS, J. Capo. *ACM SIGPLAN Notices* 44, 3 (Feb. 2009), 73.
- [21] NETHERCOTE, N., AND SEWARD, J. Valgrind: A program supervision framework. In *Proceedings of the Third Workshop on Runtime Verification* (Boulder, Colorado, USA, July 2003).
- [22] PATIL, H., PEREIRA, C., STALLCUP, M., LUECK, G., AND COWNIE, J. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization* (New York, NY, USA, 2010), CGO '10, ACM, pp. 2–11.
- [23] TUCEK, J., XIONG, W., AND ZHOU, Y. Efficient online validation with delta execution. *ACM SIGPLAN Notices* 44, 3 (Feb. 2009), 193.
- [24] XU, M., MALYUGIN, V., SHELDON, J., VENKITACHALAM, G., WEISSMAN, B., AND INC, V. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS* (2007).

Privacy-Preserving Social Plugins

Georgios Kontaxis,[†] Michalis Polychronakis,[†] Angelos D. Keromytis,[†] Evangelos P. Markatos^{*}

[†]*Columbia University*, ^{*}*FORTH-ICS*

{kontaxis,mikepo,angelos}@cs.columbia.edu, markatos@ics.forth.gr

Abstract

The widespread adoption of social plugins, such as Facebook’s Like and Google’s +1 buttons, has raised concerns about their implications to user privacy, as they enable social networking services to track a growing part of their members’ browsing activity. Existing mitigations in the form of browser extensions can prevent social plugins from tracking user visits, but inevitably disable any kind of content personalization, ruining the user experience.

In this paper we propose a novel design for *privacy-preserving social plugins* that decouples the retrieval of user-specific content from the loading of a social plugin. In contrast to existing solutions, this design preserves the functionality of existing social plugins by delivering the same *personalized* content, while it protects user privacy by avoiding the transmission of user-identifying information at load time. We have implemented our design in *SafeButton*, an add-on for Firefox that fully supports seven out of the nine social plugins currently provided by Facebook, including the Like button, and partially due to API restrictions the other two. As privacy-preserving social plugins maintain the functionality of existing social plugins, we envisage that they could be adopted by social networking services themselves for the benefit of their members. To that end, we also present a pure JavaScript design that can be offered transparently as a service without the need to install any browser add-ons.

1 Introduction

Social plugins enable third-party websites to offer personalized content by leveraging the social graph, and allow their visitors to seamlessly share, comment, and interact with their social circles [12]. For example, Facebook’s Like button, probably the most widely deployed social plugin [33], enables users to leave positive feedback for the web page in which it has been embedded, share the page with their social circle, and view their

like-minded friends. Google’s “+1” button [16] offers almost identical features to the Like button, while similar widgets are also available from other popular social networking services (SNSs) such as Twitter and LinkedIn.

Social plugins offer multifaceted benefits to both content providers and members of SNSs, a fact that is reflected by the tremendous growth in their adoption. Indicatively, as of June 2012, more than two million websites have incorporated some of Facebook’s social plugins, while more than 35% of the top 10,000 websites include Like buttons—a percentage three times higher than just one year ago [33]. Unfortunately, as the number of websites that incorporate social plugins increases, so does the portion of their visitor’s browsing history that gets exposed.

To personalize the content of third-party web pages, social plugins connect to the SNS and transmit a unique user identifier—usually contained in an HTTP cookie—along with the URL of the visited page. Consequently, the SNS receives detailed information about every visit of its members to any page with embedded social plugins. Considering the increasing adoption rate of social plugins, a constantly growing part of its members’ browsing history can be precisely tracked.

More importantly, the cookies used in social plugins are linked to user profiles that typically contain the person’s name, email address, and other private information. Although third-party tracking cookies as used by advertising networks and traffic analytics services also aim to track the pages visited by a specific user [43], in essence they track the pages opened using a particular browser instance running on a device with a given IP address. While this can already be considered as personally identifying information to some extent, in addition to that information, social plugins reveal much more: the browsing history of *individuals*.

The important implications of social plugins to user privacy were identified soon after their release [34, 54], and concerns have been intensifying [34, 37]. As avoid-

ing becoming a member of any SNS is often rather difficult (even users that are not interested in the social aspects of a service can be affected, e.g., Gmail users can still be tracked through Google’s “+1” buttons), privacy-conscious users can resort to browser extensions that block user-identifying information from reaching the SNS through social plugins [27, 15, 7, 4, 28, 45].

Depending on the subtlety of their approach, ranging from stripping cookies and headers from the plugin’s requests to preventing the plugin from loading, some or none of its user interaction functionality may be preserved. However, as user-identifying information never reaches the SNS, all these solutions completely disable any kind of content personalization. As an example, for a Like button, even logged in members will be viewing just the total number of “likes” for the page (Fig. 1a), instead of the names and pictures of their friends who have liked the page (Fig. 1c).

We believe that the majority of users are not even aware of the privacy issues stemming from the prevalence of social plugins. For this reason, we argue that any solution can be effective only if it can be deployed by SNSs themselves, so as to protect *all* users without requiring any action on their behalf. Crucially, content personalization and user interaction are two key features of existing social plugins. Any solution that lacks either of them, or introduces even a slight compromise in user experience, is not likely to be adopted by SNSs.

Driven by these two observations, in this paper we propose a novel design for *privacy-preserving social plugins*, which fulfills two seemingly contradicting goals: it protects user privacy by avoiding the transmission of user-identifying information at load time, while it offers identical functionality to existing social plugins by providing the same personalized content. The main idea is to decouple the retrieval of private information from the loading of a social plugin by prefetching all data from the user’s social circle that might be needed in the context of a social plugin. Any missing non-private data is retrieved on demand without revealing the identity of the user to the SNS. Local (private) and server-side (public) data are then combined to render a pixel-by-pixel identical version of the same personalized content that would have been rendered by existing social plugins.

To demonstrate the feasibility of our design, we have implemented *SafeButton*, an add-on for Firefox that provides privacy-preserving versions of existing social plugins, as they are provided by the major SNSs. Based on our experimental evaluation, the local disk space consumed by SafeButton for storing the private data required for handling the nine different social plugins currently provided by Facebook is in the order of a few megabytes for typical users, and 145MB for the extreme case of a user with 5,000 friends. At the same time, due to re-

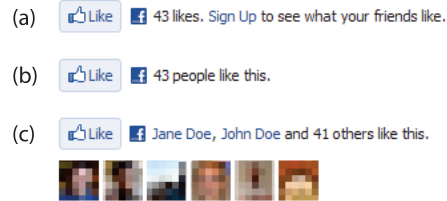


Figure 1: Different states of Facebook’s Like button for a user that (a) has never logged in on Facebook from this particular browser or is not a member of Facebook at all, (b) has previously logged in but is currently logged out, (c) is currently logged in (personalized view).

duced network overhead, SafeButton renders social plugins 64% faster compared to their original versions. Our design can be readily adopted by existing SNSs, and be offered transparently as a service to their members without the need to install any additional software.

Our work makes the following main contributions:

- We propose a novel design for privacy-preserving social plugins that i) prevents the SNS from tracking its members’ browsing activities, and ii) provides the same functionality as existing social plugins with no compromises in content personalization.
- We have implemented SafeButton, a Firefox extension that currently provides privacy-preserving versions of Facebook’s social plugins.
- We evaluate our implementation and demonstrate the feasibility of the proposed design in terms of functionality, effectiveness, and performance.
- We describe in detail a pure JavaScript implementation of our design that can be offered by existing SNSs as a transparent service to their members.

2 User Tracking through Social Plugins

2.1 Social Plugins

Social plugins are provided by the major social networking services in the form of “widgets” that can be embedded in any web page, usually in the form of an IFRAME element. After downloading the page, the browser issues a subsequent request to fetch and load the content of the plugin, as shown in Fig. 2 (step 2). The domain that serves the social plugins is the same as the one that hosts the SNS itself, and thus any state that the browser maintains for the SNS in the form of HTTP cookies [17] is transmitted along with the request for the social plugin.

Assuming the user has an active session with the SNS, the site will associate the request with the user’s profile,

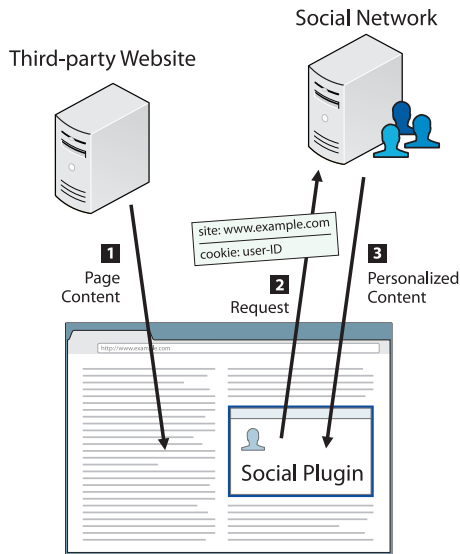


Figure 2: Loading phase of social plugins. After a page is fetched (1), the browser loads the IFRAME of the social plugin (2). If the user is logged in on the SNS, the plugin receives and displays personalized information (3). Users are identified (and can be tracked) through the HTTP cookies included in the request.

and respond with personalized content tailored to that particular user and visited web page (step 3 in Fig. 2). Otherwise, if the user has not logged in on the SNS from that particular browser before, or has never registered at all, the social plugin will display only generic, publicly accessible information for that page.

For instance, Fig. 1 shows the different modes of the Like button depending on the browser’s cookies for `facebook.com`. If a user does not have an account or has not logged in on Facebook using that browser, the plugin displays only the total number of “likes” and prompts the user to sign up (a). If a user is currently logged in, the plugin displays personalized information, including some of the names and pictures of the user’s friends that have liked the page (c). Interestingly, while a user is logged out (b), the plugin does not prompt for sign-up; depending on how cookies are cleared, some user-identifying information may persist even upon user exit.

2.2 Privacy Issues

With publishers reporting multifold increases in traffic [35], and the continuous addition of new gestures and social features by the major social networking services [36], it is expected that the explosive popularity of social plugins will only continue to grow. As more sites employ social plugins, the potential for broader user tracking increases. With more than 35% of the

top 10,000 most visited websites having Like buttons in their pages (as of June 2012) [33], a good part of the daily browsing history of 901 million active Facebook users [8] is technically available to Facebook. We should stress that the same issue holds for all other major social networking services that provide social plugins, including Google and Twitter.

The privacy issues related to the use of HTTP cookies are a well-known problem. Since their introduction in 1995, cookies have been extensively used by advertising networks for building user profiles and tracking the browsing activity of users across the web [51]. Although user tracking through social plugins resembles this kind of cross-site tracking through third-party cookies [43], there is one key difference.

An advertising network uses cookies to track the same user across all affiliate sites that host the network’s advertisements, but cannot easily link the derived activity pattern to the actual identity of the user. In contrast, social plugins use cookies associated with real user profiles on the respective social networking site, which typically contain an abundance of personally identifiable information [47]. In essence, instead of tracking anonymous users, social plugins enable tracking of named *persons*.

Advertising agencies can also potentially associate a user profile with a person’s identity by combining information from other sources, e.g., in cooperation with one or more affiliate websites on which users provide contact information for registration. Social networking services, though, do not have to collude with another party because they already have access to both extensive personally identifiable information, as well as to a broad network of sites that host social plugins.

2.3 Preventing Privacy Leaks

One might think that if users diligently log out of the social networking site, they will be safe from the privacy leaks caused by its social plugins. Unfortunately, this seems a rather daunting task for users that rely daily on Google, Facebook, Twitter, and other popular SNSs for their personal and professional communication and social interaction activities. To provide convenience for frequent use, these sites follow a single sign-on approach for all offered services, and prompt users to stay logged in indefinitely through “keep me logged in” features. Consequently, users typically remain logged in throughout the whole duration of their online presence.

In some cases, even after a user logs out, the cookies of the SNS might not be cleared completely, and personally identifiable information may still persist [9]. For example, even after logging out of Facebook, a cookie with a user identifier remains in the browser, enabling features such as pre-filling a returning user’s email address in the

log in form, or avoiding to unnecessarily prompt existing members to sign up, as shown in Fig. 1(b).

Blocking of third-party cookies could be considered a mitigation to this problem, since most of the major web browsers (Chrome, Firefox, Internet Explorer) have adapted their security policy to prevent third parties from reading (in addition to writing) cookies. Therefore, even though the SNS's domain appears both as a first party (when a user visits the site directly) and as a third party (when a social plugin is embedded in a page), in the latter case the SNS no longer receives any cookies. However, with the exception of Internet Explorer, blocking of third-party cookies is not enabled by default. Internet Explorer will do so, but white-lists same-domain cookies set by first parties that return a P3P header [30] (even a dummy one), which both Facebook and Google [25] appear to be doing. Moreover, even if a user chooses explicitly to block third-party cookies, there are known bypass techniques [2], such as faking an interaction with the embedded page through a script-initiated form submission in Safari, or opening the embedded page in a pop-up window that gets treated by the browser as a first party [53], which interestingly in Chrome is not hindered by pop-up blocking [3].

The Do Not Track HTTP header [5] is an encouraging recent initiative that allows users to opt out of tracking by advertising networks and analytics services. Although currently not supported by any SNS, if it were adopted, Do Not Track could allow users to choose whether they want to opt in for the personalized versions of social plugins or not. However, users who would opt in for the personalized versions (or who would not opt out, depending on the default setting) could still be tracked.

This situation drives privacy-conscious users towards browser extensions that block the transmission of user-identifying information through social plugins [27, 15, 7, 4, 28, 45]. For instance, Facebook Blocker [7] removes completely the IFRAME elements of social plugins from visited web pages. Instead of blocking social plugins completely, ShareMeNot [28] simply removes the sensitive cookies from the social plugin's requests at load time. When a user explicitly interacts with a plugin, the cookies are then allowed to go through, enabling the action to complete normally. Although this approach strikes a balance between usability and privacy, it still completely disables any content personalization.

3 Design

3.1 Requirements

The design of privacy-preserving social plugins is driven by two key requirements: *i) provide identical functionality to existing social plugins in terms of content personaliza-*

tion and user interaction, and ii) avoid the transmission of user-identifying information to the social networking service before any user interaction takes place. The first requirement is necessary for ensuring that users receive the full experience of social plugins, as currently offered by the major SNSs. Existing solutions against user tracking do not provide support for content personalization, and thus are unlikely to be embraced by SNSs and content providers. The second requirement is mandatory for preventing SNSs from receiving user-identifying information whenever users merely view a page and do not interact with a social plugin.

We consider as user-identifying information any piece of information that can be used to *directly* associate a social plugin instance with a user profile on the SNS, such as a cookie containing a unique user identifier. The IP address of a device or a browser fingerprint can also be considered personally identifying information, and could be used by a shady provider for user tracking. However, the accuracy of such signals cannot be compared with the ability of directly associating a visit to a page with the actual *person* that visits the page, due to factors that introduce uncertainty [52], such as DHCP churn, NAT, proxies, multiple users using the same browser, and other aspects that obscure the association of a device with the actual person behind it. Users can mitigate the effect of these signals to their privacy by browsing through an anonymous communication network [38], and ensuring that their browser has a non-unique fingerprint [39].

When viewed in conjunction, the two requirements seem contradicting. Content personalization presumes knowledge of the person for whom the content will be personalized. Nevertheless, the approach we propose satisfies both requirements, and enables a social plugin instance to render personalized content without revealing any user-identifying information to the SNS.

3.2 Overall Approach

Social plugins present the user with two different types of content: *private* information, such as the names and pictures of friends who like a page, and *public* information, such as the total number of "likes." The main idea behind our approach is to maintain a local copy of all private information that can possibly be needed for rendering any personalized content for a particular user, and query the social networking service only for public information that can be requested anonymously.

This approach satisfies our first requirement, since all the required private information for synthesizing and presenting personalized content is still available to the social plugin locally, while any missing public information can be fetched on demand. User interaction is not hindered in any way, as user actions are handled in the same way

as in existing social plugins. Our second requirement is also accomplished, because all communication of a privacy-preserving social plugin with the SNS for loading its content *does not* include any user-identifying information. Only public information about the page might be requested, which can be retrieved anonymously.

The whole process is coordinated by the *Social Plugin Agent*, which runs in the context of the browser and has three main tasks: i) upon first run, gathers all private data that might be needed through the user’s profile and social circle, and stores it in a local *DataStore*, ii) periodically, synchronizes the *DataStore* with the information available online by adding or deleting any new or stale entries, and iii) whenever a social plugin is encountered, synthesizes and presents the appropriate content by combining private, personalized information from the local *DataStore* and public, non-personalized information through the SNS. Maintaining a local copy of the user’s social information is a continuous process, and takes place transparently in the background. Once all necessary information has been mirrored during the bootstrapping phase, the *DataStore* is kept up to date periodically.

Going back to the example of the Like button, the private information that must be stored locally for its privacy-preserving version should suffice for properly rendering *any possible* instance of its personalized content for *any third-party page the user might encounter*. This can be achieved by storing locally all the “likes” that all of the user’s friends have ever made, as well as the names and thumbnail pictures of the user’s friends. Note that all the above information is available through the profile history of the user’s friends, which is always accessible while the user is logged in.

Although keeping all this state locally might seem daunting at first, as we demonstrate in Sec. 5.2, the required space for storing all the necessary private information for privacy-preserving versions of *all* Facebook’s existing social plugins is just 5.4MB for the typical case of a user with 190 friends, and 145MB for an extreme case of a user with 5,000 friends. No information that is not accessible under the user’s credentials is ever needed, and daily synchronization typically requires the transmission of a few kilobytes of data.

Continuing with the Like button as an example, Fig. 3 illustrates the process of rendering its privacy-preserving version. Upon visiting a third-party page, the Social Plugin Agent requests from the SNS the total number of “likes” for that particular page, without providing any user-identifying information (step 3). In parallel, it looks up the URL of the page in the *DataStore* and retrieves the names and pictures of the friends that have liked the page (if any). Once the total number of “likes” arrives (step 4), it is combined with the local information and the unified personalized content is presented to the user (5).

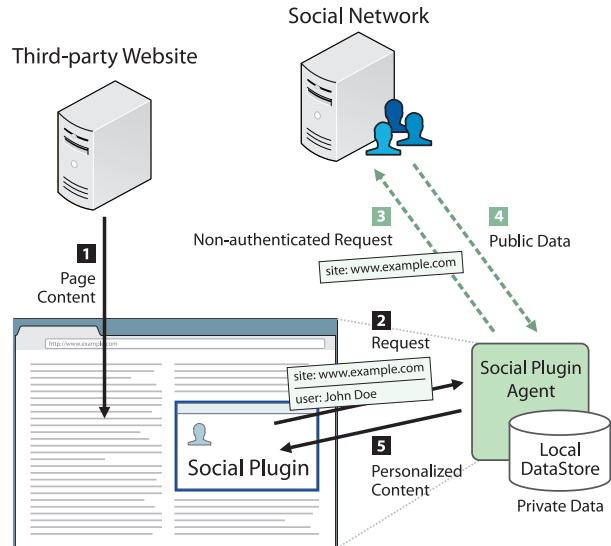


Figure 3: The loading phase of privacy-preserving social plugins. When a social plugin is encountered (1), the Social Plugin Agent intervenes between the plugin and the SNS (2). The agent requests (3) and receives (4) only publicly accessible content, e.g., the page’s total number of “likes,” without revealing any user-identifying information to the SNS. The agent then combines this data with personalized information that is maintained locally, and presents the unified content to the user (5).

Further optimizations are possible for avoiding querying for non-personalized content at load time. Depending on the plugin and the kind of information it provides, public information for frequently visited pages can be cached, while public information for highly popular pages can be prefetched. For example, information such as the total number of “likes” for a page that a user visits several times a day can be updated only once per day without introducing a significant inconsistency, allowing the Social Plugin Agent to occasionally serve the Like button using *solely* local information. Similarly, the SNS can regularly push to the agent the total number of “likes” for the top 10K most “liked” pages. In both cases, the elimination of any network communication on every cache hit not only reduces the rendering time, but also protects the user’s browsing pattern even further.

4 Implementation

To explore the feasibility of our approach we have implemented *SafeButton*, an add-on for Firefox (version 7.0.1) that provides privacy-preserving versions of existing social plugins. *SafeButton* is written in JavaScript and XUL [23], and relies on the XPCOM interfaces of Firefox to interact with the internals of the browser. Figure 4

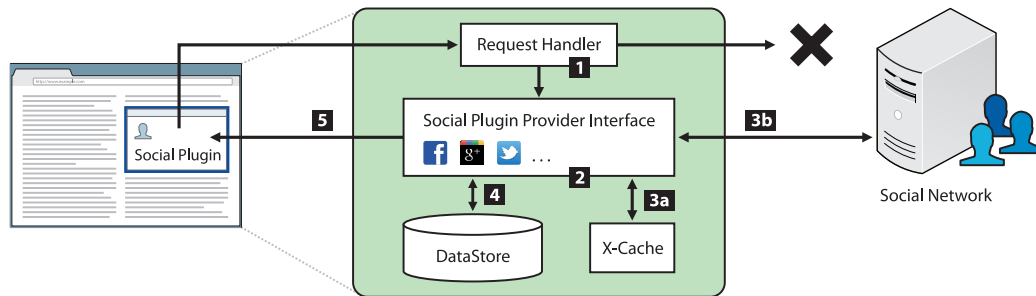


Figure 4: Overall architecture of SafeButton. A Request Handler (1) intercepts the HTTP requests of social plugins. Privacy-preserving implementations of the supported plugins (2) combine public remote data (3b), which can be cached in the X-Cache for improving network performance (3a), and private data from the user’s social circle, which are maintained locally in the DataStore (4), and deliver the same personalized content (5) as the original plugins.

provides an overview of SafeButton’s main components, which are described below. A detailed description of how the components are put together to handle a Like button is provided at the end of this section.

Request Handler The main task of the Request Handler is to intercept the HTTP requests of a social plugin at load time, and hand off the event to an appropriate callback handler function. The requests are intercepted using a set of filters based on signatures that capture the target URL of each plugin. These signatures are received from the Social Plugin Provider Interface, along with the callback handlers that should be invoked whenever a filter is triggered. The Request Handler provides as an argument to these callbacks a reference to the DOM of the page that contains the social plugin that triggered the filter.

We have implemented the Request Handler by registering an observer for HTTP requests (`http-on-modify-request` notification) using XPCOM’s `nsIObserverService`. This allows the inspection code to lie inline in the HTTP request creation process, and either intercept and modify requests (e.g., by stripping HTTP cookies or other sensitive headers), or drop them entirely when necessary.

Social Plugin Provider Interface The Social Plugin Provider Interface serves as an abstraction between the Request Handler and different *Provider Modules* that support the social plugins offered by different social networking services. This extensible design enables more networks and plugins to be supported in the future. In the current version of SafeButton, we have implemented a Provider Module for the social plugins offered by Facebook. We take advantage of the Graph API [10] to download the user’s private social information that needs to be stored locally, and access any other public content on demand. We should stress that, although an option, we do

not employ any kind of web scraping to acquire information from pages accessible through the user’s profile.

A Provider Module for a SNS consists of: i) the signatures that will be used by the Request Handler for intercepting the HTTP requests of the platform’s social plugins, ii) the callback handler functions that implement the core functionality of each social plugin based on local and remote social information, and iii) the necessary logic for initializing the DataStore and keeping it up to date with the information that is available online.

Each callback function implements the core functionality for rendering a particular social plugin. Its main task is to retrieve the appropriate private social data from the DataStore, request any missing public data from the SNS (without revealing any user-identifying information), and compile the two into the personalized content that will be displayed. The function then updates the DOM of the web page through the page reference that was passed by the Request Handler.

DataStore The DataStore keeps locally all the private social data that might be required for rendering personalized versions of any of the supported social plugins. All information is organized in a SQLite database that is stored in the browser’s profile folder for the user that has installed SafeButton. Upon first invocation, SafeButton begins the process of prefetching the necessary data. This process takes place in the background, and relies on the detection of browser idle time and event scheduling to operate opportunistically without interfering with the user’s browsing activity.

In our implementation for Facebook, data retrieval begins with information about the user’s friends, including each friend’s name, thumbnail picture, and unique identifier in Facebook’s social graph. Then, for each friend, SafeButton retrieves events of social activity such as the pages that a friend has liked or shared, starting with the oldest available event and moving onward. In case the

download process is interrupted, e.g., if the user turns off the computer, it continues from where it left off the next time the browser is started.

Updating the DataStore is an incremental process that takes place periodically. Fortunately, the current version of the Graph API offers support for incremental updates. As we need to query for any new activity using a separate request for each friend (a Graph API function for multiple user updates would be welcome), we do so gracefully for each friend every two hours, or, if the browser is not idle, in the next idle period. We have empirically found the above interval to strike a good balance between the timeliness of the locally stored information and the incurred network overhead. In our future work, we plan to employ a more elaborate approach based on an exponential backoff algorithm, so that a separate adaptive update interval can be maintained for different friend groups according to their “chattiness.”

Note that we also need to address the consistency of the locally stored data with the corresponding data that is available online. For instance, friends may “like” a page and later on “unlike” it, thereby deleting this activity from their profile. Unfortunately, the Graph API currently does not offer support for retrieving any kind of removal events. Nevertheless, SafeButton periodically fetches the entire set of activities for each friend (at a much slower pace than the incremental updates), and removes any stale entries from the DataStore.

X-Cache The *X-Cache* holds frequently used public information and meta-information, such as the total number of “likes” for a page or the mapping between page URLs and objects in the Facebook graph. A hit in the X-Cache means that no request towards the social networking service is necessary for rendering a social plugin. This improves significantly the time it takes for the rendering process to complete, and at the same time does not reveal the IP address of the user to the SNS.

Use Case: Facebook Like Button Here we enrich the running case of the Facebook Like button from Sec. 3 with the technical details of the behavior of SafeButton’s components, as shown by the relevant steps in Fig. 4.

Upon visiting a web page with an embedded Like button in the form of an IFRAME, the browser will issue an HTTP request towards Facebook to load and subsequently render the contents of that IFRAME. The Request Handler intercepts this request and attempts to match its URL against the set of signatures of the supported social plugins, which will trigger a match for the regular expression `http[s]?://www\.facebook\.com/plugins/like\.php`. Subsequently, the handler invokes the callback associated with this signature and pass as an

argument the plugin’s URL and a reference to the DOM of the page that contains the social plugin (step 1).

The first action of the callback function is to query X-Cache for any cached non-personalized information about the button and the page it is referring to. This includes the mapping between the page’s URL and its ID in the Facebook graph, along with the global count of users who have “liked” the page (step 3a). In case of a miss, a request made through the Graph API retrieves that information (step 3b). The request is stripped from any Facebook cookies that the browser unavoidably appends to it. The response is then added to X-Cache for future reference. After retrieving the global count of users, the names (and if the developer has chosen so, the thumbnail pictures) of the user’s friends that have liked the page are retrieved from the LocalStore (step 4).

Finally, the reference to the DOM of the embedding page (passed by the handler in step 1), is used to update the IFRAME where the original Like button would have been with exactly the same content (step 5).

5 Experimental Evaluation

5.1 Supported Facebook Plugins

In this section we discuss the social plugins offered by Facebook and evaluate the extent to which SafeButton can support them in respect to two requirements: i) user privacy, and ii) support for personalized content. Table 1 lists the nine social plugins currently offered by Facebook. For each plugin, we provide a brief categorization of its “view” functionality, i.e., the content presented to the user according to whether it is based on public (non-personalized) or private (personalized) information, as well as its “on-click” functionality, i.e., the type of action that a user can take.

Although SafeButton interferes with the “view” functionality of existing social plugins, it does not affect their “on-click” functionality, allowing users to interact normally as with the original plugins. As shown in Table 2, SafeButton currently provides complete support for seven out of the nine social plugins currently offered by Facebook.

The Like button and its variation, the Like Box, are fully functional; the count, names, and pictures of the user’s friends are retrieved from the DataStore, while the total number “likes” is requested on demand anonymously. The Recommendations plugin presents a list of recommendations for pages from the same site, with those made by friends appearing first. Recommendations from the user’s friends are stored locally, so SafeButton can render those that are relevant to the visited site on top. The list is then completed with public recommendations by others, which are retrieved on demand.

Facebook Social Plugin	Public Content	Personalized Content	User Action
Like Button	Total number of people that have liked the page	Names and pictures of friends that have liked the page	Like page
Send Button	-	-	Send content/page URL
Comments	List of user comments	Friends' comments appear on top	Post comment
Activity Feed	List of user activities (likes, comments, shared pages)	Friends' activities appear on top	-
Recommendations	List of user recommendations (likes)	Friends' recommendations appear on top	-
Like Box	Total number of people that have liked the Facebook Page, names and pictures of some of them, list of recent posts from the Page	Names and pictures of friends that have liked the page are shown first	Like page
Registration	-	User's Name, picture, birthday, gender, location, email (prefilled in registration form)	Register
Facepile	-	Names and pictures of friends that have liked the page	-
Live Stream	User messages	-	Post message

Table 1: Public vs. Personalized content in Facebook's social plugins [12].

Facebook Social Plugin	Exposed information during loading		Personalized Content with SafeButton
	Original	SafeButton	
Like Button	IP addr. + cookies	IP addr.	Complete
Send Button	IP addr. + cookies	None	Complete
Comments	IP addr. + cookies	IP addr.	Partial ¹
Activity Feed	IP addr. + cookies	IP addr.	Partial ²
Recommendations	IP addr. + cookies	IP addr.	Complete
Like Box	IP addr. + cookies	IP addr.	Complete
Registration	IP addr. + cookies	None	Complete
Facepile	IP addr. + cookies	IP addr.	Complete
Live Stream	IP addr. + cookies	IP addr.	Complete

¹ When all comments are loaded at once, all personalized content is complete. In case they are loaded in a paginated form, some of the friends' comments (if any) might not be shown in the first page.

² Some of the friends' comments (if any) might be omitted (access to comments is currently not supported by Facebook's APIs).

Table 2: For 7 out of the 9 Facebook social plugins, SafeButton provides exactly the same personalized content without exposing any user-identifying information.

Similarly to the Like button, Facepile presents pictures of friends who have liked a page, and that information is already present in the DataStore. The Send, Register, and Login buttons do not present any kind of dynamic information, and thus can be rendered instantly without issuing any network request.

Similarly to the Recommendations plugin, content personalization in the Comments plugin consists of giving priority to comments made by friends. SafeButton retrieves the non-personalized version of the plugin, and reorders the received comments so that friends' comments are placed on top. When all comments for a page are fetched at once, the personalized information pre-

sented by SafeButton is *fully consistent* with the original version of the plugin. However, when comments are presented in a paginated form, only the first sub-page is loaded. The current version of the Graph API does not support the retrieval of comments (e.g., in contrast to “likes”), and thus in case friends' comments appear deeper than the first sub-page, SafeButton will not show them on top (a workaround would be to download all subsequent comment sub-pages, but for popular pages this would result in a prohibitive amount of data).

The Activity Feed plugin is essentially a wrapper for showing a mix of “likes” and comments by friends, and thus again SafeButton's output lacks any friends' comments. Note that our implementation is based solely on the functionality provided by the Graph API [10], and we refrain from scraping of web content for any missing information. Ideally, future extensions of the Graph API will allow SafeButton to fully support the personalized content of all plugins. We discuss this and other missing functionality that would facilitate SafeButton in Sec. 7.

5.2 Space Requirements

To explore the local space requirements of SafeButton, we gathered a data set that simulates the friends a user may have. Starting with a set of friends from the authors' Facebook profiles, we crawled the social graph and identified about 300,000 profiles with relaxed privacy settings that allow unrestricted access to all profile information, including the pages that person has liked or shared in the past. From these profiles, we randomly selected a set of 5,000—the maximum number of friends a person can have on Facebook [6].

Data	190 Friends	5,000 Friends
Names, IDs of Friends	10.5KB	204.8KB
Photos of Friends	463.4KB	11.8MB
Likes of Friends	4.6MB	126.7MB
Shares of Friends	318.4KB	7.0MB
Total	5.4MB	145.7MB
Average (per friend)	29.2KB	29.7KB

Table 3: Storage space requirements for the average case of 190 friends and the borderline case of 5,000 friends.

To quantify the space needed for storing the required data from a user’s social circle, we initialized SafeButton using the above 5,000 profiles. In detail, SafeButton prefetches the names, IDs, and photos of all friends, and the URLs of all pages they have liked or shared. Although we have employed a slow-paced data retrieval process (5sec delay between consecutive requests), the entire process for all 5,000 friends took less than 10 hours. For typical users with a few hundred friends, bootstrapping completes in less than a hour. As already mentioned, users are free to use the browser during that time or shut it down and resume the process later.

Table 3 shows a breakdown of the consumed space for the average case of a user with 190 friends [58] and the extreme case of a user with 5,000 friends, which totals 5.4MB and 145.7MB, respectively. Evidently, consumed space is dominated by “likes,” an observation consistent with the prevailing popularity of the Like button compared to the other social plugins. To gain a better understanding of storage requirements for different users, Fig. 5 shows the consumed space as a function of the number of friends, which as expected increases linearly.

We should note that the above results are specific for the particular data set, and the storage space might increase for users with more “verbose” friends. Furthermore, the profile history of current members will only continue to grow as time passes by, and the storage space for older users in the future will probably be larger. Nevertheless, these results are indicative for the overall magnitude of SafeButton’s storage requirements, which can be considered reasonable even for current smartphones, while the storage space of future devices can only be expected to increase.

To further investigate the distribution of “likes,” the factor that dominates local space, we plot in Fig. 6 the CDF of the number of “likes” of each user in our data set. The median user has 122 “likes,” while there are some users with much heavier interaction: about 10% of the users have more than 504 “likes.” The total number of “likes” was 1,110,000, i.e., 222 per user on average. This number falls within the same order of mag-

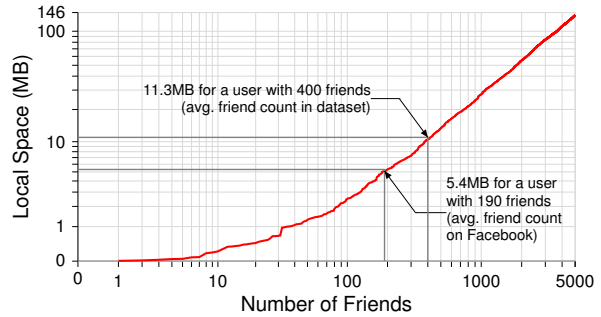


Figure 5: Local space consumption for the required information from a user’s social circle as a function of the number of friends. For the average case of a user with 190 friends, SafeButton needs just 5.4MB.

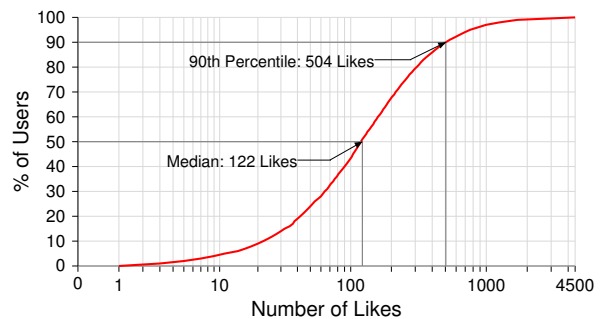


Figure 6: CDF of the number of “likes” of each user.

nitude as previously reported statistics, which suggest that there are about 381,861 “likes” per minute on Facebook [31]. With a total population of about 901 million active users [8], this results in about 217 “likes” per user per year. These results indicate that our data set is not particularly biased towards excessively active or inactive profiles.

Besides the storage of social information, SafeButton maintains the X-Cache for quick access to frequently used non-personalized information about a social plugin. To get an estimate about its size requirements, we visited the home pages of the first 1,000 of the top websites according to alexa.com that contained at least one Facebook social plugin. About 82.4% of the identified plugins corresponded to a Like Button or Like Box, 14% to Facebook Connect, 3% to Recommendations, 0.5% to Send Button, and 0.1% to Facepile and Activity Box. After visiting all above sites, X-Cache grew to no more than 850KB, for more than 2,500 entries.

5.3 Speed

In this experiment, we explore the rendering time of social plugins with and without SafeButton. Specif-

ically, we measured the time from the moment the HTTP request for loading the IFRAME of a Like button is sent by the browser, until its content is fully rendered in the browser window. To do so, we instrumented Firefox with measurement code triggered by `http-on-modify-request` notifications [20] and `pageshow` events [21]. We chose to measure the rendering time for the IFRAME instead of the entire page to eliminate measurement variations due to other remote elements in the page. This is consistent with the way a browser renders a page, since IFRAMEs are loaded in parallel with the rest of its elements.

We consider the following three scenarios: i) Firefox rendering a Like button unobstructed, and Firefox with SafeButton rendering a Like button when there is ii) an X-Cache miss or iii) an X-Cache hit. For the original Like button, we used a hot browser cache to cancel out loading times for any required external elements, such as CSS and JavaScript files. Using SafeButton, visiting a newly or infrequently accessed webpage will result in a miss in the X-Cache. For a Like button, this means that besides looking up the relevant information in the local DataStore, SafeButton must (anonymously) query Facebook to retrieve the total number of “likes.” For frequently accessed pages, such personalized information will likely already exist in the X-Cache, and thus SafeButton does not place any network request at all.

Using a set of the first 100 among the top websites according to `alexa.com` that contain a Like button, we measured the loading time of the Like button’s IFRAME for each site (each measurement was repeated 1,000 times). Figure 7 shows the median loading time across all sites for each scenario, as well as its breakdown according to the events that take place during loading. The rendering time for the original Like button is 351ms, most of which is spent for communication with Facebook. In particular, it takes 130ms from the moment the browser issues the request for the IFRAME until the first byte of the response is received, and another 204ms for the completion of the transfer. In contrast, SafeButton is much faster, as it needs 127ms for rendering the Like button in case of an X-Cache miss (2.8 times faster than the original), and just 24ms in case of an X-Cache hit (14.6 times faster), due to the absence of any network communication.

The difference in the response times for the network requests placed by the original Like button and SafeButton in case of an X-Cache miss can be associated with the different API used and amount of data returned in each case. SafeButton uses the Graph API to retrieve just the total number of “likes,” which is returned as a raw ASCII value that is just a few bytes long. In contrast, the original plugin communicates with a different endpoint from the side of Facebook, and fetches a full HTML page with embedded CSS and JavaScript content. While these two

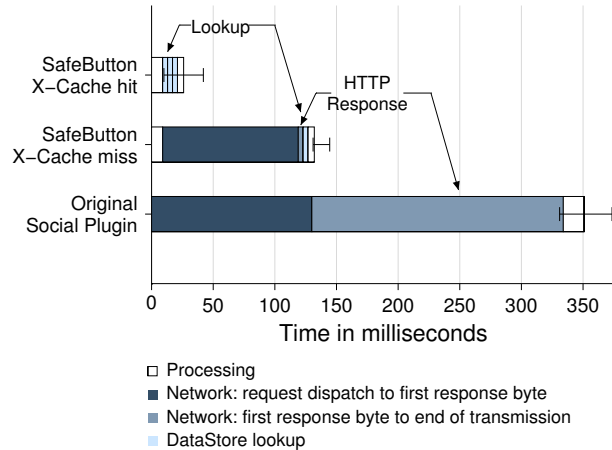


Figure 7: Loading time for Like button with and without SafeButton. Even when the total number of “likes” is not available in the X-Cache, SafeButton is 2.8 times faster.

requests need a similar amount of time from the moment they are placed until the first response byte is received from the server, they differ by two orders of magnitude in terms of the time required to complete the transfer. Even if Facebook optimizes its own plugins in the future, we expect the rendering speed of SafeButton to be comparable in case of an X-Cache miss, and still much faster in case of an X-Cache hit.

5.4 Effectiveness

As presented in Sec. 3, we rely on a set of heuristics that match the target URL of each supported social plugin to intercept and treat them accordingly so as to protect the user’s privacy. To evaluate the effectiveness and accuracy of our approach, we carried out the following experiment. Using `tcpdump`, we captured a network trace of all outgoing communication of a test PC in our lab while surfing the web for a week through Firefox equipped with SafeButton. We then inspected the trace and found that no cookie was ever transmitted in any HTTP communication with `facebook.com` or any of its sub-domains.

This was a result of the following “fail-safe” approach. Besides the signatures of the supported social plugins, SafeButton inspects all communication with `facebook.com` and strips any cookies from requests initiated by third-party pages. Next, we performed the reverse experiment: using the same browser equipped with SafeButton, we surfed `www.facebook.com` and interacted with the site’s functionality without any issues for a long period. Careful inspection of the log generated by SafeButton proved that no in-Facebook communication was hindered at any time.

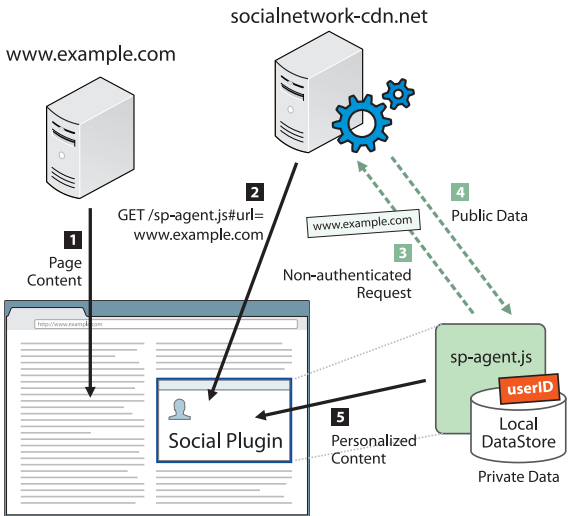


Figure 8: Privacy-preserving social plugins serviced by a SNS. Here: the loading of a social plugin in a third-party page. The code of the social plugin agent is always fetched from a secondary domain to avoid leaking cookies set by the primary domain of the SNS. The URL of the target page is passed via a fragment identifier, so it is never transmitted to the SNS. The agent synthesizes and renders the personalized content of the social plugin.

6 Privacy-preserving Social Plugins as a Service: A Pure JavaScript Design

As many users are typically not aware of the privacy issues of social plugins, they are not likely to install any browser extension for their protection. For instance, NoScript [27], a Firefox add-on which blocks untrusted JavaScript code from being executed, has roughly just 2 million downloads, and Adblock [1], an add-on which prevents advertisement domains from loading as third parties in a web page, has been downloaded 14 million times. At the same time, Firefox has 450 million active users [24], which brings the adoption rate of the above security add-ons to 0.4% and 3.1%, respectively. For this reason, in this section we present a pure JavaScript implementation of privacy-preserving social plugins that could be employed by social networking services themselves for the protection of their members.

The use case would not be much different from now: web developers would still embed an IFRAME element that loads the social plugin from the SNS. However, instead of serving a traditional social plugin, the SNS serves a JavaScript implementation of a social plugin agent in respect to the design presented in Sec. 3. The agent then fetches personalized information from the browser's local storage, requests non-personalized information from the SNS, and renders the synthesized con-

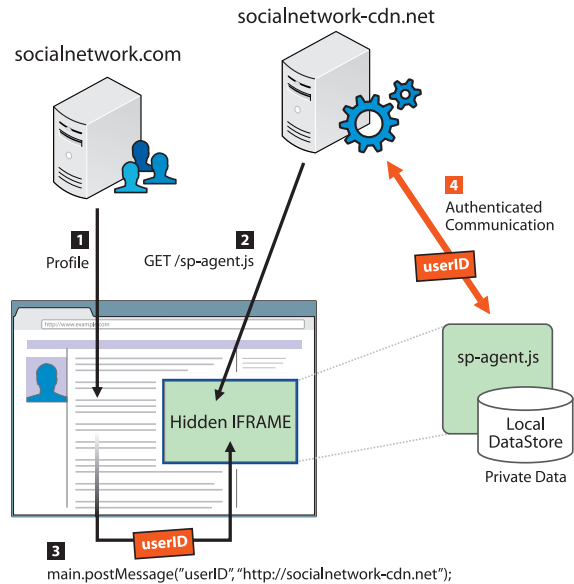


Figure 9: Privacy-preserving social plugins serviced by a SNS. Here: securely communicating the user's session identifier to the social plugin agent when logging in on the SNS. Although the agent is hosted on a secondary domain, it receives and stores the identifier from the primary domain through the `postMessage` function, allowing it to place asynchronous authenticated requests for accessing the user's profile information.

tent according to the specified social plugin. The feasibility of the above design is supported by existing web technologies such as IndexedDB [19], which provide a JavaScript API for managing a local database, similar to the DataStore used in SafeButton.

The most challenging aspect of this implementation is to prevent the leakage of user-identifying information during the loading of a social plugin. If the IFRAME of the social plugin agent is hosted on the same (sub)domain as the SNS itself (e.g., `socialnetwork.com`), then the request for fetching its JavaScript code would also transmit the user's cookies for the SNS. At the same time, the agent would need to know the URL of the embedding page for which it has personalized the social plugin's content. If the URL is passed as a parameter to that initial request, the situation is obviously as problematic as in current social plugins.

A solution would be to leave out the URL of the page from the request for loading the social plugin agent. However, there should be a way to communicate this information to the agent once its JavaScript code has been loaded by the browser. This can be achieved through a fragment identifier [32] in the URL from which the agent is loaded. Fragment identifiers come as the last part of a URL, and begin with a hash mark (#) char-

acter. According to the HTTP specification [18], fragment identifiers are never transmitted as part of a request to a server. Thus, during the loading of a social plugin in a third-party page, instead of passing an explicit parameter with the URL of the embedding page, as in `www.socialnetwork.com/sp-agent.js?url=<URL>`, it can be passed through a fragment identifier, as in `www.socialnetwork.com/sp-agent.js#<URL>`. The information about the URL of the visited page never leaves the browser, and remains accessible to the JavaScript code of the agent, which can then parse the hypertext reference of its container and extract the fragment identifier.

Unfortunately, this approach is still not secure in practice. The URL of the embedding page is usually also transmitted as part of the HTTP Referer [sic] header by most browsers. Therefore, even if we omit the target URL from the HTTP parameters of the request, the server will receive it anyway, allowing the SNS to correlate this information with the user's cookies that are transmitted as part of the same request.

To overcome this issue, the social plugin agent can be hosted on a secondary domain, different than the primary domain of the SNS, as also proposed by Do Not Track [5]. For instance, in this design the agent could be hosted under `socialnetwork-cdn.net` instead of `socialnetwork.com`, as shown in Fig. 8. This prevents the browser from appending the user's cookies whenever a social plugin is encountered (step 2), since its IFRAME will be served from a different domain than the one for which the cookies were set. The rest of the steps are analogous to Fig. 3.

Still, the social plugin agent must be able to issue authenticated requests towards the SNS for accessing the user's profile and retrieving the necessary private social information that must be maintained locally. This requires access to the user's cookies, and specifically to the identifier of the authenticated session that the user has with the SNS.

A solution to this problem can be achieved by taking advantage of the `windows.postMessage` [22] JavaScript API, which allows two different origins to communicate. When the user logs in on the SNS, the login page contains a hidden IFRAME loaded through HTTPS from the secondary domain on which the social plugin agent is hosted, as shown in Fig. 9 (step 2). The login page then communicates to the agent's IFRAME the session identifier of the user through `postMessage` (step 3). The IFRAME executes JavaScript code that stores locally the user identifier under its own domain, making it accessible to the plugin agent. The agent can then read the session identifier from its own local storage, and place authenticated requests towards the SNS for accessing the user's profile (step 4) and synchronizing the required information with the locally stored data. When the user ex-

PLICITLY logs out from the social networking site, the log out page follows a similar process to erase the identifier from the local storage of the agent.

In respect to supporting multiple users per browser instance and protecting the personal information stored locally, encryption can be employed to shield any sensitive information, such as the names or identifiers of a user's friends. In accordance with the communication of the session identifier described above, a user-specific cryptographic key can be communicated from the SNS to the social plugin agent. The plugin can then use this key to encrypt sensitive information locally. The key is kept only in memory. Each time the plugin agent loads, it spawns a child IFRAME towards the social networking site. The request for the child IFRAME will normally have the user's cookies appended. Finally, that child IFRAME, once loaded, can communicate via `postMessage` the encryption key back to the plugin agent.

7 Discussion

Strict Mode of Operation Although SafeButton does not send any cookies to the social networking service, it still needs to make non-authenticated requests towards the SNS to fetch public information for some social plugins (e.g., for Facebook plugins, the information shown in column "Public Content" in Table 1). These requests unavoidably expose the user's IP address to the SNS.

Some users might not feel comfortable with exposing their IP address to the SNS (even when no cookies are sent), as this information could be correlated by the SNS with other sources of information, and could eventually lead to the exposure of the users' true identity. For such privacy-savvy users, we consider a "paranoid" mode of operation in which SafeButton does not reveal the user's IP address to the social networking service when encountering a social plugin in a third-party page, by simply not retrieving any public information about the page. Unavoidably, some social plugins are then rendered using solely the locally available personalized information, e.g., for the Like button, the total number of "likes" for the page will be missing.

Alternatively, given the very low traffic incurred by SafeButton's non-authenticated queries to the SNS, these can be carried out transparently by SafeButton through an anonymous communication network such as Tor [38]. Given that social plugins are loaded in parallel with the rest of the page's elements, this would minimally affect the browsing experience (compared to browsing solely through Tor).

Potential Challenges with Future Social Plugins. Although SafeButton currently supports all social plugins

offered by Facebook, and our approach is extensible so as to handle the plugins of other social networking services, we consider two potential challenges with future plugins [44]. First, future personalization functionality could include social information from a user's second degree friends, i.e., the friends of his friends, or rely on the analysis of data from the entire user population of the social network. Second, this type of personalization could involve proprietary algorithms not available to the client-side at run-time.

We believe that our approach could be adapted to support such developments. We find it realistic that such extended analysis will take place offline, and result in the calculation of a product that will be stored and taken into account in real-time during content personalization. Therefore, it will not be necessary to have at the client side neither the analysis algorithms nor the entire dataset. The stored outcome of the analysis, e.g., some extra weight on the social graph or additional meta-data, could be available to through the developer's API, and be taken into account by SafeButton during content personalization. At the same time, the social networking service is not deprived of the data necessary to carry out such analysis. Our approach protects user privacy when accessing the "view" functionality of social plugins, but when users explicitly interact with them, their actions and any corresponding data are transmitted to the SNS.

Profile Management As users may access the web via more than one devices, it reasonable to assume that they will require a practical way to use SafeButton in all of them. Although installing SafeButton on each browser should be enough, this will result to the synchronization of the locally stored information with the SNS for each instance separately. In our future work, we will consider the use of cloud storage for keeping fully-encrypted copies of the local DataStore and X-Cache, and synchronizing them across all the user's browser instances, in the same spirit as existing settings and bookmark synchronization features of popular browsers [29, 14].

Keeping a local copy of private information that is normally accessible only through the social networking service might be considered a security risk, as it would be made readily available to an attacker that gains unauthorized access to the user's system. At that point, though, the attacker would already have access to the user's credentials (or could steal them by installing a keylogger on the compromised host) and could easily gather this information from the SNS anyway.

In any case, users could opt-in for keeping the DataStore encrypted, although this would require them to provide a password to SafeButton (similarly to the above mentioned settings synchronization features). For the pure JavaScript implementation, though, as discussed in

Section 6, the cryptographic key can be supplied by the SNS upon user login, making the process completely transparent to the user.

Security in Multi-user Environments We now consider the operation of SafeButton in a multi-user environment where more than one users share the same browser instance. In general, sharing the same browser instance is a bad security practice, because after users are done with a browsing session they may leave sensitive information behind, such as stored passwords, cookies, and browsing history. Ideally, users should maintain their own browser instance or accounts in the operating system.

SafeButton retrieves private information when users are logged in the SNS, and stores it locally even after they log out, as it would be inefficient to erase it every single time. Multiple users are supported by monitoring the current cookies for that domain of the SNS, and serving personalized content only for the user that is currently logged in. Local entries that belong to a user ID that does not match the one currently logged in are never returned. Obviously, users that share the same OS account can access each other's locally stored data, since they are contained in the same DataStore instance, unless they have opted in for keeping their data encrypted, as discussed earlier.

Shortcomings of the Graph API Throughout this paper we have briefly mentioned some obstacles we have encountered, namely shortcomings in the developer API provided by Facebook, in respect to our objective of protecting the user's privacy while maintaining full functionality for the social plugins. We summarize these issues here and discuss how the social networks in general could support us.

User Activity Updates through the API. Currently the Facebook API [10] offers access to the social graph but there is no way to receive updates or "diffs" when something changes. For instance, we retrieve a friend's "likes" through the API, we are also able to fetch only new "likes" from a point forward, but are unable to receive notice when that friend "unlikes." A friend "activity" or "history" function could significantly aid our implementation in keeping an accurate local store.

Accuracy of the Provided Information. Sometimes, the API calls and documentation offered to developers differ slightly from the actual behavior of a plugin when it is offered by the SNS itself [11]. This creates a predicament for developers wishing to replicate the functionality.

Support for All Social Information that is Otherwise Accessible. We consider it reasonable for the API to provide access to information that is accessible via the social plugins offered by the SNS itself or via the profile pages of its users. For instance, there is no API call to

access the comments of a specific user, although they appear in the user's profile page. Scrapping could retrieve them, but this practice is not ideal. Therefore, in our case, we have to resolve to practices that result in reduced accuracy, such as anonymously retrieving a sample of the comments of a page and placing the comments of a user's friends at the top, if present in the sample. Retrieving the entire set of comments could be inefficient for pages with too many comments.

Alternatively, Facebook could provide a call for retrieving just the user IDs of all the commenters, and another call for specifying a set of IDs for which to retrieve the actual comments. In that case, we could hide the IDs of a user's friends among a group of k strangers and request their comments for that page [56].

8 Related Work

Do Not Track [5] is a browser technology which enables users to signal, via an HTTP header, that they do not wish to be tracked by websites they do not explicitly visit. Unfortunately there are no guarantees that such a request will be honored by the receiving site or not.

Krishnamurthy et al. [46] studied privacy leaks in online social networking services. They identified the presence of embedded content from third-party domains in the interactions of a user with the SNS itself and stress that the combination with personal information inside an SNS could pose a significant threat to user privacy.

There has been significant work in the interplay between SNSs and privacy. For example, there has been some focus on protecting privacy in SNS against third-party applications installed in a user's profile within the social network [41, 40, 55]. Facecloak [49] shields a user's personal information from a SNS and any third-party interaction, by providing fake information to the SNS and storing actual, sensitive information in an encrypted form on a separate server. The authors in Fly-ByNight [48] propose the use of public key cryptography among friends in a SNS so as to protect their information from a curious social provider and potential data leaks.

Recent work has focused on how to support personalized advertisements without revealing the user's personal information to the providing party. Adnostic [57] offers targeted advertising while preserving the user's privacy by having the web browser profile the user, through the monitoring of his browsing history, and inferring his interests. It then downloads diverse content from the advertising server and selects which part of it to display to the user. Similarly, RePriv [42] enables the browser to mine a user's web behavior to infer guidelines for content personalization, which are ultimately communicated to interested sites. Our approach differs in principle as the model of these previous systems prevents a web site

from building a profile for the user while we decouple the identification step the user undergoes, to access his already existing social profile, with his subsequent requests for content personalization.

Mayer et al. [50] highlight the threats against user privacy by the cross-site tracking capabilities of third-party web services. The authors detail a plethora of tracking technologies used by the embedded pages of advertisement, analytics, and social networking services. Their work demonstrates the high level of sophistication in web tracking technologies, and their resiliency against browser countermeasures.

Roesner et al. [53] study the tracking ecosystem of third-party web services and discuss current defenses, including third-party cookie blocking. They identify cases where tracking services actively try to evade such restrictions by bringing themselves in a first party position, e.g., by spawning pop-up windows. Moreover, the authors present cases in which services are treated as first parties when visited directly and intentionally by the users, and at the same time appear embedded as third parties in web sites, as is the case with social networking services and their social plugins. Overall, they conclude that current restrictions imposed by browsers against third-party tracking are not fool-proof, and at the same time find more than 500 tracking services, some with the capability to capture more than 20% of a user's browsing behavior.

A series of browser add-ons exist [7, 26] that block social plugins from the web pages a user visits by removing them or preventing them from loading, in a manner similar to what Adblock [1] does for advertisements. However, they come at the cost of full loss of functionality as social plugins are completely removed from a page. Note that some of these add-ons are poorly implemented and naively remove the social plugins only after they have appeared on a page, meaning that the corresponding HTTP request containing user-identifying information has already been issued towards the server.

ShareMeNot [28, 53] is a Firefox add-on that strips user cookies from a series of HTTP requests that the web browser issues to load social plugins. As a result, no user-identifying information is sent to the social networking service until the user explicitly interacts with the social plugin. The downside of this approach is that users are deprived of any personalized information offered by the plugin, e.g., the number and names of any of their friends that might have already interacted with a page. In other words, users view these social plugins as if they were logged out from the respective SNS (or browsing in "incognito" mode). Our approach differs from ShareMeNot in that it focuses on providing the full content personalization of existing social plugins while protecting user privacy.

9 Conclusion

Concerns about the interplay between social plugins and privacy are mounting rapidly. Tensions have reached the point that even governments consider to outlaw Facebook's Like button [13]. Recently, in an official response to questions regarding user privacy asked by the government of Norway, it was stated that "*Facebook does not use cookies to track people visiting websites using the Like button*" [37]. The current design of social plugins, as provided by all major social networking services, combined with empirical evidence [9], stresses the need for changes so that words align with actions. We want to believe that SNSs treat the privacy of their members as an issue of the utmost importance, and we hope that they are willing to ensure it through technical means.

In this paper, we have presented a novel design for privacy-preserving social plugins, which provide exactly the same user experience as existing plugins, and at the same time prevent SNSs from being able to track the browsing activities of their users. We have described in detail how this design can be offered transparently as a service to users of existing SNSs without the need to install any additional software, and thus envisage that it could be adopted for the protection of their member's privacy. SafeButton, our proof-of-concept implementation of this design as a browser add-on for Firefox, demonstrates the practicality of our approach. SafeButton is publicly available, and currently supports full content personalization in a privacy-preserving way and with minimal space overhead for seven out of the nine social plugins offered by Facebook, while it loads them 2.8 times faster compared to their original versions.

Availability

SafeButton is publicly available as an open source project at <http://www.cs.columbia.edu/~kontaxis/safebutton/>

Acknowledgements

This work was supported in part by the FP7-PEOPLE-2009-IOF project MALCODE and the FP7 project SysSec, funded by the European Commission under Grant Agreements No. 254116 and No. 257007. This work was also supported by the National Science Foundation through Grant CNS-09-14312, with additional support from Google. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government or the NSF.

References

- [1] Adblock Plus. <https://addons.mozilla.org/en-US/firefox/addon/adblock-plus/>.
- [2] Browser Security Handbook - Third-party cookie rules. http://code.google.com/p/browsersec/wiki/Part2#Third-party_cookie_rules.
- [3] Chromium - Don't play plugin instances inside suppressed popups? <http://code.google.com/p/chromium/issues/detail?id=3477>.
- [4] Disconnect. <http://disconnect.me/>.
- [5] Do Not Track - Universal Web Tracking Opt Out. <http://donottrack.us/>.
- [6] Facebook - How many Pages can I like? <https://www.facebook.com/help/?faq=116603848424794>.
- [7] Facebook Blocker. <http://webgraph.com/resources/facebookblocker/>.
- [8] Facebook Fact Sheet. <http://newsroom.fb.com/content/default.aspx?NewsAreaId=22>.
- [9] Facebook fixes logout issue, explains cookies. <http://nikcub.appspot.com/facebook-fixes-logout-issue-explains-cookies>.
- [10] Facebook Graph API. <http://developers.facebook.com/docs/reference/api/>.
- [11] Facebook Like Button Count Inaccuracies. <http://faso.com/fineartviews/21028/facebook-like-button-count-inaccuracies>.
- [12] Facebook Plugins. <http://developers.facebook.com/docs/plugins/>.
- [13] Facebook's Like button illegal in German state. http://news.cnet.com/8301-1023_3-20094866-93/facebook-s-like-button-illegal-in-german-state/.
- [14] Firefox Sync. <http://www.mozilla.org/en-US/mobile/sync/>.
- [15] Ghostery. <http://www.ghostery.com/>.
- [16] Google +1 button. <http://www.google.com/+1/button/>.
- [17] HTTP state management. <http://www.ietf.org/rfc/rfc2109.txt>.
- [18] Hypertext Transfer Protocol 1.1. <http://www.ietf.org/rfc/rfc2616.txt>.
- [19] Indexed Database API. <http://www.w3.org/TR/IndexedDB/>.
- [20] MDN - Intercepting Page Loads. https://developer.mozilla.org/en/XUL_School/Intercepting_Page_Loads.
- [21] MDN - Pageshow Event. https://developer.mozilla.org/en/using_firefox_1.5_caching#pageshow.
- [22] MDN - window.postMessage. <https://developer.mozilla.org/en/DOM/window.postMessage>.
- [23] MDN - XML User Interface Language. <https://developer.mozilla.org/En/XUL>.
- [24] Mozilla At a Glance. <http://blog.mozilla.org/press/ataglance/>.
- [25] MSDN Blogs - Google Bypassing User Privacy Settings.

- <http://blogs.msdn.com/b/ie/archive/2012/02/20/google-bypassing-user-privacy-settings.aspx>.
- [26] No Llike. <https://chrome.google.com/webstore/detail/pockodjapmojcccdpgfhkjldcnbhenjm>.
- [27] NoScript. <https://addons.mozilla.org/en-US/firefox/addon/noscript/>.
- [28] ShareMeNot. <http://sharemenot.cs.washington.edu/>.
- [29] The Chromium projects - Sync. <http://www.chromium.org/developers/design-documents/sync>.
- [30] The Platform for Privacy Preferences Specification. <http://www.w3.org/TR/P3P/>.
- [31] Time Magazine - One Minute on Facebook. http://www.time.com/time/video/player/0,32068,711054024001_2037229,00.html.
- [32] Uniform Resource Identifier. <http://www.ietf.org/rfc/rfc3986.txt>.
- [33] Widgets Distribution. <http://trends.builtwith.com/widgets>.
- [34] An Open Letter to Facebook CEO Mark Zuckerberg, June 2010. https://www.eff.org/files/filenode/social_networks/OpenLettertoFacebook.pdf.
- [35] Facebook + Media - The Value of a Liker, Sept. 2010. https://www.facebook.com/note.php?note_id=150630338305797.
- [36] 5 ways Facebook's new features will fuel social shopping, Sept. 2011. <http://mashable.com/2011/09/29/facebook-social-shopping/>.
- [37] Facebook's response to questions from the Data Inspectorate of Norway, Sept. 2011. <http://www.datatilsynet.no/upload/Dokumenter/utredningeravDatatilsynet/FromFacebook-Norway-DPA.pdf>.
- [38] R. Dingledine, N. Mathewson, and P. Syverson. Tor: the second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, pages 303–320. USENIX Association, 2004.
- [39] P. Eckersley. How unique is your web browser? In *Proceedings of the 10th international conference on Privacy Enhancing Technologies*, pages 1–18. Springer, 2010.
- [40] M. Egele, A. Moser, C. Kruegel, and E. Kirda. PoX: Protecting users from malicious facebook applications. In *Proceedings of the 9th Annual IEEE international conference on Pervasive Computing and Communications (PerCom), Workshop Proceedings*, pages 288–294. IEEE Computer Society, 2011.
- [41] A. Felt and D. Evans. Privacy protection for social networking platforms. In *Proceedings of the 2008 IEEE Workshop on Web 2.0 Security and Privacy*, 2008.
- [42] M. Fredrikson and B. Livshits. RePriv: Re-envisioning in-browser privacy. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, pages 131–146. IEEE Computer Society, 2011.
- [43] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *Proceedings of the 15th international World Wide Web Conference (WWW)*, pages 737–744. ACM, 2006.
- [44] A. Kobsa. Privacy-enhanced personalization. *Communications of the ACM*, 50:24–33, August 2007.
- [45] G. Kontaxis, M. Polychronakis, and E. P. Markatos. SudoWeb: Minimizing information disclosure to third parties in single sign-on platforms. In *Proceedings of the 14th Information Security Conference*, pages 197–212. Springer, 2011.
- [46] B. Krishnamurthy and C. E. Wills. Characterizing privacy in online social networks. In *Proceedings of the 1st Workshop on Online Social Networks*, pages 37–42. ACM, 2008.
- [47] B. Krishnamurthy and C. E. Wills. On the leakage of personally identifiable information via online social networks. *SIGCOMM Computer Communication Review*, 40, 2010.
- [48] M. M. Lucas and N. Borisov. FlyByNight: mitigating the privacy risks of social networking. In *Proceedings of the 7th ACM workshop on Privacy in the Electronic Society (PETS)*, pages 1–8. ACM, 2008.
- [49] W. Luo, Q. Xie, and U. Hengartner. FaceCloak: An architecture for user privacy on social networking sites. In *Proceedings of the international conference on computational science and engineering*, pages 26–33. IEEE Computer Society, 2009.
- [50] J. R. Mayer and J. C. Mitchell. Third-Party Web Tracking: Policy and Technology. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2012.
- [51] L. I. Millett, B. Friedman, and E. Felten. Cookies and web browser design: toward realizing informed consent online. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2001.
- [52] M. A. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. My botnet is bigger than yours (maybe, better than yours): why size estimates remain challenging. In *Proceedings of the first workshop on Hot topics in understanding Botnets (Hot-Bots)*. USENIX Association, 2007.
- [53] F. Roesner, T. Kohno, and D. Wetherall. Detecting and defending against third-party tracking on the web. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2012.
- [54] A. Roosendaal. Facebook tracks and traces everyone: Like this! <http://ssrn.com/abstract=1717563>.
- [55] K. Singh, S. Bhola, and W. Lee. xbook: Redesigning privacy control in social networking platforms. In *Proceedings of the 18th USENIX Security Symposium*, pages 249–266. USENIX Association, 2009.
- [56] L. Sweeney. k-anonymity: a model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10:557–570, October 2002.
- [57] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy preserving targeted advertising. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS)*. IEEE Internet Society, 2010.
- [58] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the facebook social graph. *CoRR*, abs/1111.4503, 2011.

Social Networking with Frienteegrity: Privacy and Integrity with an Untrusted Provider

Ariel J. Feldman, Aaron Blankstein, Michael J. Freedman, and Edward W. Felten

Princeton University

Abstract

Today's social networking services require users to trust the service provider with the confidentiality and integrity of their data. But with their history of data leaks and privacy controversies, these services are not always deserving of this trust. Indeed, a malicious provider could not only violate users' privacy, it could *equivocate* and show different users divergent views of the system's state. Such misbehavior can lead to numerous harms including surreptitious censorship.

In light of these threats, this paper presents Frienteegrity, a framework for social networking applications that can be realized with an *untrusted* service provider. In Frienteegrity, a provider observes only encrypted data and cannot deviate from correct execution without being detected. Prior secure social networking systems have either been decentralized, sacrificing the availability and convenience of a centralized provider, or have focused almost entirely on users' privacy while ignoring the threat of equivocation. On the other hand, existing systems that are robust to equivocation do not scale to the needs social networking applications in which users may have hundreds of friends, and in which users are mainly interested the latest updates, not in the thousands that may have come before.

To address these challenges, we present a novel method for detecting provider equivocation in which clients collaborate to verify correctness. In addition, we introduce an access control mechanism that offers efficient revocation and scales logarithmically with the number of friends. We present a prototype implementation demonstrating that Frienteegrity provides latency and throughput that meet the needs of a realistic workload.

1. Introduction

Popular social networking sites have hundreds of millions of active users [20]. They have enabled new forms of communication, organization, and information sharing; or, as Facebook's prospectus claims, they exist "to make the world more open and connected" [60]. But by now, it is widely understood that these benefits come at the cost of having to trust these centralized services with the privacy of one's social interactions. The history of these services is rife with unplanned data disclosures (*e.g.*, [22, 40]), and

these services' centralization of information makes them attractive targets for attack by malicious insiders and outsiders. In addition, social networking sites face pressure from government agencies world-wide to release information on demand, often without search warrants [24]. Finally and perhaps worst of all, the behavior of service providers themselves is a source of users' privacy concerns. Providers have repeatedly changed their privacy policies and default privacy settings, and have made public information that their users thought was private [46, 47].

Less recognized, however, is the extent to which users trust social networking sites with the *integrity* of their data, and the harm that a malicious or compromised provider could do by violating it. Prior work on secure social networking has focused primarily on privacy and largely neglected integrity, or at most employed digital signatures on users' individual messages [5, 53, 54, 56]. But a malicious provider could be more insidious. For example, bloggers have claimed that Sina Weibo, a Chinese microblogging site, tried to disguise its censorship of a user's posts by hiding them from the user's followers but still showing them to the user [51]. This behavior is an example of server *equivocation* [34, 39], in which a malicious service presents different clients with divergent views of the system state. We argue that to truly protect users' data, a secure social networking service should defend against this sort of attack.

To address the security concerns surrounding social networking, numerous prior works (*e.g.*, [5, 17, 56]) have proposed decentralized designs in which the social networking service is provided not by a centralized provider, but by a collection of federated nodes. Each node could either be a service provider of a user's choice or the user's own machine or those of her friends. We believe that decentralization is the wrong, or at least an insufficient, approach, however, because it leaves the user with an unenviable dilemma: either sacrifice availability, reliability, and convenience by storing her data on her own machine, or entrust her data to one of several providers that she probably does not know or trust any more than she would a centralized provider.

In light of these problems, we present Frienteegrity, a framework for building social networking services that protects the *privacy* and *integrity* of users' data from a

potentially malicious provider, while preserving the availability, reliability, and usability benefits of centralization. Frientegrity supports familiar social networking features such as “walls,” “news feeds,” comment threads, and photos, as well as common access control mechanisms such as “friends,” “friends-of-friends,” and “followers.” But in Frientegrity, the provider’s servers only see encrypted data, and clients can collaborate to detect server equivocation and other forms of misbehavior such as failing to properly enforce access control. In this way, Frientegrity bases its confidentiality and integrity guarantees on the security of users’ cryptographic keys, rather than on the service provider’s good intentions or the correctness of its complex server code. Frientegrity remains highly scalable while providing these properties by spreading system state across many shared-nothing servers [52].

To defend against server equivocation, Frientegrity enforces a property called *fork* consistency* [33]. A fork*-consistent system ensures that if the provider is honest, clients see a strongly-consistent (linearizable [27]) ordering of updates to an object (*e.g.*, a wall or comment thread). But if a malicious provider presents a pair of clients with divergent views of the object, then the provider must prevent the clients from ever seeing each other’s subsequent updates lest they identify the provider as faulty.

Prior systems have employed variants of fork* consistency to implement network file systems [33, 34], key-value stores [7, 38, 50], and group collaboration systems [21] with untrusted servers. But these systems assumed that the number of users would be small or that clients would be connected to the servers most of the time. As a result, to enforce fork* consistency, they presumed that it would be reasonable for clients to perform work that is linear in either the number of users or the number of updates ever submitted to the system. But these assumptions do not hold in social networking applications in which users have hundreds of friends, clients are Web browsers or mobile devices that connect only intermittently, and users typically are interested only in the most recent updates, not in the thousands that may have come before.

To accommodate these unique scalability challenges, we present a novel method of enforcing fork* consistency in which clients *collaborate* to detect server equivocation. This mechanism allows each client to do only a small portion of the work required to verify correctness, yet is robust to collusion between a misbehaving provider and as many as f malicious users, where f is a predetermined security parameter per object.

Access control is another area where social networking presents new scalability problems. A user may have hundreds of friends and tens of thousands of friends-of-friends (FoFs) [19]. Yet, among prior social networking systems that employ encryption for access control (*e.g.*,

[5, 9, 37]), many require work that is linear in the number of friends, if not FoFs, to revoke a friend’s access (*i.e.*, to “un-friend”). Frientegrity, on the other hand, supports fast revocation of friends and FoFs, and also gives clients a means to efficiently verify that the provider has only allowed writes from authorized users. It does so through a novel combination of *persistent authenticated dictionaries* [12] and *key graphs* [59].

To evaluate the scalability of Frientegrity, we implemented a prototype that simulates a Facebook-like service. We demonstrate that Frientegrity is capable of scaling with reasonable performance by testing this prototype using workloads with tens of thousands of updates per object and access control lists containing hundreds of users.

Roadmap In §2, we introduce Frientegrity’s goals and the threat model against which it operates. §3 presents an overview of Frientegrity’s architecture using the task of fetching a “news feed” as an example. §4 delves into the details of Frientegrity’s data structures and protocols for collaboratively enforcing fork* consistency on an object, establishing dependencies between objects, and enforcing access control. §5 discusses additional issues for untrusted social networks such as friend discovery and group administration. We describe our prototype implementation in §6 and then evaluate its performance and scalability in §7. We discuss related work in §8 and then conclude.

2. System Model

In Frientegrity, the service provider runs a set of servers that store *objects*, each of which corresponds to a familiar social networking construct such as a Facebook-like “wall”, a comment thread, or a photo or album. Clients submit updates to these objects, called *operations*, on behalf of their users. Each operation is encrypted under a key known only to a set of authorized users, such as a particular user’s friends, and not to the provider. Thus, the role of the provider’s servers is limited to storing operations, assigning them a canonical order, and returning them to clients upon request, as well as ensuring that only authorized clients can write to each object. To confirm that servers are fulfilling this role faithfully, clients collaborate to verify their output. Whenever a client performs a read, it checks whether the response is consistent with the responses that other clients received.

2.1 Goals

Frientegrity should satisfy the following properties:

Broadly applicable: If Frientegrity is to be adopted, it must support the features of popular social networks such as Facebook-like walls or Twitter-like feeds. It must also support both the symmetric “friend” and “friend-of-friend” relationships of services like Facebook and the asymmetric “follower” relationships of services like Twitter.

Keeps data confidential: Because the provider is untrusted, clients must encrypt their operations before submitting them to the provider's servers. Frientegrity must ensure that all and only the clients of authorized users can obtain the necessary encryption keys.

Detects misbehavior: Even without access to objects' plaintexts, a malicious provider could still try to forge or alter clients' operations. It could also equivocate and show different clients inconsistent views of the objects. Moreover, malicious users could collude with the provider to deceive other users or could attempt to falsely accuse the provider of being malicious. Frientegrity must guarantee that as long as the number of malicious users with permission to modify an object is below a predetermined threshold, clients will be able to detect such misbehavior.

Efficient: Frientegrity should be sufficiently scalable to be used in practice. In particular, a client that is only interested in the most recent updates to an object should not have to download and check the object in its entirety just so that it can perform the necessary verification. Furthermore, because social networking users routinely have hundreds of friends and tens of thousands of friends-of-friends [19], access control list changes must be performed in time that is better than linear in the number of users.

2.2 Detecting Server Equivocation

To prevent a malicious provider from forging or modifying clients' operations without detection, Frientegrity clients digitally sign all their operations with their users' private keys. But as we have discussed, signatures are not sufficient for correctness, as a misbehaving provider could still equivocate about the history of operations.

To mitigate this threat, Frientegrity employs *fork* consistency* [33].¹ In *fork**-consistent systems, clients share information about their individual views of the history by embedding it in every operation they send. As a result, if clients to whom the provider has equivocated ever communicate, they will discover the provider's misbehavior. The provider can still *fork* the clients into disjoint groups and only tell each client about operations by others in its group, but then it can never again show operations from one group to the members of another without risking detection. Furthermore, if clients are occasionally able to exchange views of the history out-of-band, even a provider which forks the clients will not be able to cheat for long.

¹Fork* consistency is a weaker variant of an earlier model called *fork consistency* [39]. They differ in that under *fork consistency*, a pair of clients only needs to exchange one message to detect server equivocation, whereas under *fork* consistency*, they may need to exchange two. Frientegrity enforces *fork* consistency* because it permits a one-round protocol to submit operations, rather than two. It also ensures that a crashed client cannot prevent the system from making progress.

Ideally, to mitigate the threat of provider equivocation, Frientegrity would treat all of the operations performed on all of the objects in the system as a single, unified history and enforce *fork** consistency on that history. Such a design would require establishing a total order on all of the operations in the system regardless of the objects to which they belonged. In so doing, it would create unnecessary dependencies between unrelated objects, such as between the "walls" of two users on opposite sides of the social graph. It would then be harder to store objects on different servers without resorting to either expensive agreement protocols (e.g., Paxos [31]) or using a single serialization point for all operations.

Instead, like many *scale-out* services, objects in Frientegrity are spread out across many servers; these objects may be indexed either through a directory service [1, 23] or through hashing [15, 30]. The provider handles each object independently and only orders operations with respect to the other operations in the same object. Clients, in turn, exchange their views of each object to which they have access separately. Thus, for efficiency, Frientegrity only enforces *fork** consistency on a per-object basis.

There are situations, however, when it is necessary to make an exception to this rule and specify that an operation in one object happened after an operation in another. Frientegrity allows clients to detect provider equivocation about the order of such a pair of operations by supplying a mechanism for explicitly entangling the histories of multiple objects (see §3.4).

2.3 Threat Model

Provider: We assume that the provider may be actively malicious. It may not only attempt to violate the confidentiality of users' social interactions, but also may attempt to compromise their integrity through either equivocation or by directly tampering with objects, operations, or access control lists (ACLs).

Although Frientegrity makes provider misbehavior detectable, it does not prevent a malicious provider from denying service, either by blocking all of a client's updates or by erasing the encrypted data it stores. To mitigate this threat, clients could replicate their encrypted operations on servers run by alternate providers. Furthermore, if provider equivocation creates inconsistencies in the system's state, clients can resolve them using *fork-recovery* techniques, such as those employed by SPORC [21]. We argue, however, that because Frientegrity allows provider misbehavior to be detected quickly, providers will have an incentive to avoid misbehaving out of fear of legal repercussions or damage to their reputations.

The provider does not have access to the contents of objects or the contents of the individual operations that clients upload, because they are encrypted under keys that it does not know. In addition, because users' names are

also encrypted, the provider can only identify users by pseudonyms, such as the hash of the public keys they use within the system. Nevertheless, we do not seek to hide social relationships: we assume that the provider can learn the entire pseudonymous social graph, including who is friends with whom and who interacts with whom, by analyzing the interconnections between objects and by keeping track of which pseudonyms appear in which objects, (e.g., by using social network deanonymization techniques [4, 43]).

Preventing the provider from learning the social graph is likely to be impossible in practice because even if users used a new pseudonym for every new operation, the provider would still be able to infer a great deal from the size and timing of their operations. After all, in most social networking applications, the first thing a user does when she signs in is check a “news feed” which is comprised of her friends’ most recent updates. In order to construct the news feed, she must query each of her friend’s feed objects in succession, and in so doing reveal to the provider which feed objects are related.

Users and Clients: We assume that users may also be malicious and may use the clients they control to attempt to read and modify objects to which they do not have access. In addition, malicious users may collude with the provider or with other users to exceed their privileges or to deceive honest users. They may also attempt to falsely accuse the provider of misbehavior. Finally, we assume that some clients may be controlled by Sybil users, created by the provider to subvert the clients’ defenses against server equivocation.

Frientegrity’s security is based on the assumption, however, that among the users which have access to a given object, no more than some constant f will be malicious (Byzantine faulty). We believe that this assumption is reasonable because a user can only access an object if she has been explicitly invited by another user with administrator privileges for the object (e.g., Alice can only access Bob’s wall if he explicitly adds her as a friend). As we describe in §4.1, this assumption allows clients to collaborate to detect provider misbehavior. If a client sees that at least $f + 1$ other users have vouched for the provider’s output, the client can assume that it is correct.

Client code: We assume the presence of a code authentication infrastructure that can verify that the application code run by clients is genuine. This mechanism might rely on code signing or on HTTPS connections to a trusted server (different from the untrusted service provider used as part of Frientegrity’s protocols).

3. System Overview

As discussed above, to ensure that the provider is behaving correctly, Frientegrity requires clients to verify the

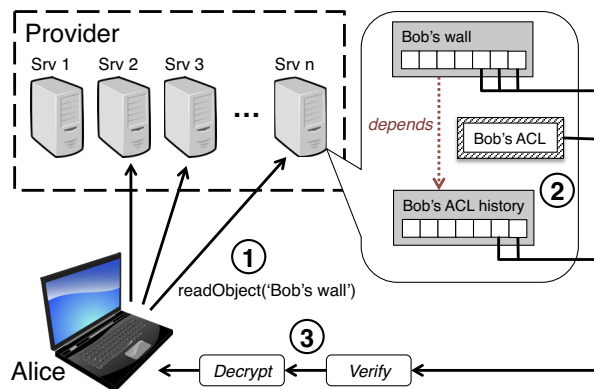


Figure 1: A client fetches a news feed in Frientegrity by reading the latest posts from her friends’ walls, as well as information to verify, authenticate, and decrypt the posts.

output that they receive from the provider’s servers. As a result, whenever clients retrieve the latest updates to an object, the provider’s response must include enough information to make such verification possible. In addition, the provider must furnish the key material that allows authorized clients with the appropriate private keys to decrypt the latest operations. Thus, when designing Frientegrity’s protocols and data structures, our central aim was to ensure that clients could perform the necessary verification and obtain the required keys *efficiently*.

To explain these mechanisms, we use the example of a user Alice who wants to fetch her “news feed” and describes the steps that her client takes on her behalf. For simplicity, in this and subsequent examples throughout the paper, we often speak of users, such as Alice, when we really mean to refer to the clients acting on their behalf.

3.1 Example: Fetching a News Feed

Alice’s news feed consists of the most recent updates to the sources to which she is subscribed. In Facebook, for example, this typically corresponds to the most recent posts to her friends’ “walls”, whereas in Twitter, it is made up of the most recent tweets from the users she follows. At a high level, Frientegrity performs the following steps when Alice’s fetches her news feed, as shown in Figure 1.

1. For each of Alice’s friends, Alice’s sends a `readObject` RPC to the server containing the friend’s wall object.
2. In response to a `readObject` RPC for a friend Bob, a well-behaved server returns the most recent operations in Bob’s wall, as well as sufficient information and key material for Alice to verify and decrypt them.
3. Upon receiving the operations from Bob’s wall, Alice performs a series of verification steps aimed at detecting server misbehavior. Then, using her private key, she decrypts the key material and uses it to decrypt the operations. Finally, when she has verified and decrypted the recent wall posts from all her

friends, she combines them and optionally filters and prioritizes them according to a client-side policy.

For Alice to verify the response to each `readObject`, she must be able to check the following properties efficiently:

1. **The provider has not equivocated about the wall's contents:** The provider must return enough of the wall object to allow Alice to guarantee that history of the operations performed on the wall is fork* consistent.
2. **Every operation was created by an authorized user:** The provider must prove that each operation from the wall that it returns was created by a user who was authorized to do so at the time that the operation was submitted.
3. **The provider has not equivocated about the set of authorized users:** Alice must be able to verify that the provider did not add, drop, or reorder users' modifications to the access control list that applies to the wall object.
4. **The ACL is not outdated:** Alice must be able to ensure that the provider did not roll back the ACL to an earlier version in order to trick the client into accepting updates from a revoked user.

The remainder of this section summarizes the mechanisms with which Frientegrity enforces these properties.

3.2 Enforcing Fork* Consistency

Clients defend against provider equivocation about the contents of Bob's wall or any other object by comparing their views of the object's history, thereby enforcing fork* consistency. Many prior systems, such as BFT2F [33] and SPORC [21], enforced fork* consistency by having each client maintain a linear hash chain over the operations that it has seen. Every new operation that it submits to the server includes the most recent hash. On receiving an operation created by another client, a client in such systems checks whether the history hash included in the operation matches the client's own hash chain computation. If it does not, the client knows that the server has equivocated.

The problem with this approach is that it requires each client to perform work that is linear in the size of the entire history of operations. This requirement is ill suited to social networks because an object such as Bob's wall might contain thousands of operations dating back years. If Alice is only interested in Bob's most recent updates, as is typically the case, she should not have to download and check the entire history just to be able to detect server equivocation. This is especially true considering that when fetching a news feed, Alice must read all of her friends' walls, and not just Bob's.

To address these problems, Frientegrity clients verify an object's history collaboratively, so that no single client

needs to examine it in its entirety. Frientegrity's collaborative verification scheme allows each client to do only a small portion of the work, yet is robust to collusion between a misbehaving provider and as many as f malicious users. When f is small relative to the number of users who have written to an object, each client will most likely only have to do work that is logarithmic, rather than linear, in the size of the history (as our evaluation demonstrates in §7.5). We present Frientegrity's collaborative verification algorithm in §4.1.

3.3 Making Access Control Verifiable

A user Bob's profile is comprised of multiple objects in addition to his wall, such as photos and comment threads. To allow Bob to efficiently specify the users allowed to access all of these objects (*i.e.*, his friends), Frientegrity stores Bob's friend list all in one place as a separate ACL. ACLs store users' pseudonyms in the clear, and every operation is labeled with the pseudonym of its creator. As a result, a well-behaved provider can reject operations that were submitted by unauthorized users. But because the provider is untrusted, when Alice reads Bob's wall, the provider must *prove* that it enforced access control correctly on every operation it returns. Thus, Frientegrity's ACL data structure must allow the server to construct efficiently-checkable proofs that the creator of each operation was indeed authorized by Bob.

Frientegrity also uses the ACL to store the key material with which authorized users can decrypt the operations on Bob's wall and encrypt new ones. Consequently, ACLs must be designed to allow clients with the appropriate private keys to efficiently retrieve the necessary key material. Moreover, because social network ACLs may be large, ACL modifications and any associated rekeying must be efficient.

To support both efficiently-checkable membership proofs and efficient rekeying, Frientegrity ACLs are implemented as a novel combination of *persistent authenticated dictionaries* [12] and *key graphs* [59]. Whereas most prior social networking systems that employ encryption required work linear in the number of friends to revoke a user's access, all of Frientegrity's ACL operations run in logarithmic time.

Even if it convinces Alice that every operation came from someone who was authorized by Bob at some point, the provider must still prove that it did not equivocate about the history of changes Bob made to his ACL. To address this problem, Frientegrity maintains an *ACL history* object, in which each operation corresponds to a change to the ACL and which Alice must check for fork* consistency, just like with Bob's wall. Frientegrity's ACL data structure and how it interacts with ACL histories are further explained in §4.3.

3.4 Preventing ACL Rollbacks

Even without equivocating about the contents of either Bob's wall or his ACL, a malicious provider could still give Alice an outdated ACL in order to trick her into accepting operations from a revoked user. To mitigate this threat, operations in Bob's wall are annotated with *dependencies* on Bob's ACL history (the red dotted arrow in Figure 1). A dependency indicates that a particular operation in one object *happened after* a particular operation in another object. Thus, by including a dependency in an operation that it posts to Bob's wall, a client forces the provider to show anyone who later reads the operation an ACL that is at least as new as the one that the client observed when it created the operation. In §4.2, we explain the implementation of dependencies and describe additional situations where they can be used.

4. System Design

Clients interact with Frientegrity primarily by reading and writing objects and ACLs via the following four RPCs:²

- `readObject(objectID, k, [otherOps])`. Returns the k most recent operations in object `objectID`, and optionally, a set of additional earlier operations from the object (`otherOps`). But as we explain in the previous section, the provider must also return enough operations from the object to allow the client to verify that the provider has not equivocated and proofs from the ACL that show that every operation came from an authorized user. In addition, it must return key material from the ACL that allows the client to decrypt the object.
- `writeObject(objectID, op)`. Submits the new operation `op` to object `objectID`. Every new operation is signed by the user that created it. To allow clients to enforce fork* consistency, it also includes a compact representation of the submitting client's view of object's state. (This implies that the client must have read the object at least once before submitting an update.)
- `readACL(aclID, [userToAdd] [userToRemove])`. Returns ACL `aclID` and its corresponding ACL history object. As an optimization, the client can optionally specify in advance that it intends to add or remove particular users from the ACL so that the provider only has to return the portion of the ACL that the client needs to change.
- `writeACL(aclID, aclUpdate)`. Submits an update to ACL `aclID`. Only administrator users (*e.g.*, the owner of a Facebook-like profile) can modify the ACL. The objects to which the ACL applies are encrypted under a key that is shared only among currently authorized

²For brevity, we omit RPCs for creating new objects and ACLs and for adding new users to the system.

users. Thus, to add a user, the client must update the ACL so that it includes the encryption of this shared key under the new user's public key. To remove a user, the ACL must be updated with a new shared key encrypted such that all remaining users can retrieve it. (See §4.3.3.)

The remainder of this section describes how Frientegrity makes these RPCs possible. It discusses the algorithms and data structures underlying object verification (§4.1), dependencies between objects (§4.2), and verifiable access control §4.3).

4.1 Making Objects Verifiable

4.1.1 Object Representation

Frientegrity's object representation must allow clients to compare their views of the object's history without requiring any of them to have the entire history. Representing an object as a simple list of operations would be insufficient because it is impossible to compute the hash of a list without having all of the elements going back to the first one. As a result, objects in Frientegrity are represented as *history trees*.

A history tree, first introduced by Crosby *et al.* [11] for tamper-evident logging, is essentially a versioned Merkle tree [41]. Like an ordinary Merkle tree, data (in this case, operations) are stored in the leaves, each internal node stores the hash of the subtree below it, and the hash of the root covers the tree's entire contents. But unlike a static Merkle tree, a history tree allows new leaves (operations) to be added to the right side of the tree. When that occurs, a new version of the tree is created and the hashes of the internal nodes are recomputed accordingly.

This design has two features that are crucial for Frientegrity. First, as with a Merkle tree, subtrees containing unneeded operations can be omitted and replaced by a stub containing the subtree's hash. This property allows a Frientegrity client which has only downloaded a subset of an object's operations to still be able to compute the current history hash. Second, if one has a version j history tree, it is possible to compute what the root hash would have been as of version $i < j$ by pretending that operations $i + 1$ through j do not exist, and by then recomputing the hashes of the internal nodes.

Frientegrity uses history trees as follows. Upon receiving a new operation via a `writeObject` RPC, the server hosting the object adds it to the object's history tree, updates the root hash, and then digitally signs the hash. This server-signed hash is called a *server commitment* and is signed to prevent a malicious client from later falsely accusing the server of cheating.

When Alice reads an object of version i by calling `readObject`, the server responds with a pruned copy of the object's history tree containing only a subset of the opera-

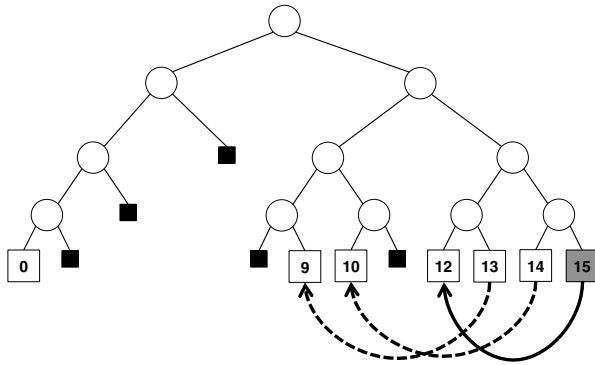


Figure 2: A pruned object history that a provider might send to a client. Numbered leaves represent operations and filled boxes represent stubs of omitted subtrees. The solid arrow represents the last operation’s prevCommitments. Dashed arrows represent other prevCommitments.

tions, along with C_i , the server commitment to version i of the object. If Alice then creates a new operation, she shares her view of the history with others by embedding C_i in the operation’s prevCommitment field. If Bob later reads the object, which by then has version $j \geq i$, he can compare the object he receives with what Alice saw by first computing what the root hash would have been at version i from his perspective and then comparing it to the prevCommitment of Alice’s operation. If his computed value C'_i does not equal C_i , then he knows the server has equivocated.

4.1.2 Verifying an Object Collaboratively

But how many operations’ prevCommitments does a client need to check in order to be confident that the provider has not misbehaved? Clearly, if the client checks every operation all the way back to the object’s creation, then using a history tree provides no advantage over using a hash chain. Consequently, in Frientegrity, each client only verifies a suffix of the history and trusts others to check the rest. If we assume that there are at most f malicious users with write access to an object, then as long as at least $f + 1$ users have vouched for a prefix of the history, subsequent clients do not need to examine it.

To achieve this goal, every client executes the following algorithm to verify an object that it has fetched. In response to a readObject RPC, the provider returns a pruned object history tree that includes all of the operations the client requested along with any additional ones that the client will need to check in order to verify the object. Because the provider knows the client’s verification algorithm, it can determine a priori which operations the client will need. For simplicity, the algorithm below assumes that only one user needs to vouch for a prefix of the history in order for it to be considered trustworthy (*i.e.*, $f = 0$). We relax this assumption in the next section.

1. Suppose that Alice fetches an object, and the provider replies with the pruned object shown in Figure 2. Because the object has version 15, the provider also sends Alice its commitment C_{15} . On receiving the object, she checks the server’s signature on C_{15} , recomputes the hashes of the internal nodes, and then verifies that her computed root hash C'_{15} matches C_{15} . Every operation she receives is signed by the user that created it, and so she verifies these signatures as well.
2. Alice checks the prevCommitment of the last operation (op_{15}), which in this case is C_{12} .³ To do so, Alice computes what the root hash would have been if op_{12} were the last operation and compares her computed value to C_{12} . (She must have op_{12} to do this.)
3. Alice checks the prevCommitment of every operation between op_{12} and op_{15} in the same way.
4. Frientegrity identifies every object by its first operation.⁴ Thus, to make sure that the provider did not give her the wrong object, Alice checks that op_0 has the value she expects.

4.1.3 Correctness of Object Verification

The algorithm above aims to ensure that at least one honest user has checked the contents and prevCommitment of every operation in the history. To see how it achieves this goal, suppose that op_{15} in the example was created by the honest user Bob. Then, C_{12} must have been the most recent server commitment that Bob saw at the time he submitted the operation. More importantly, however, because Bob is honest, Alice can assume that he would have never submitted the operation unless he had already verified the entire history up to op_{12} . As a result, when Alice verifies the object, she only needs to check the contents and prevCommitments of the operations after op_{12} . But how was Bob convinced that the history is correct up to op_{12} ? He was persuaded the same way Alice was. If the author of op_{12} was honest, and op_{12} ’s prevCommitment was C_i , then Bob only needed to examine the operations from op_{i+1} to op_{12} . Thus, by induction, as long as writers are honest, every operation is checked even though no single user examines the whole history.

Of course in the preceding argument, if any user colludes with a malicious provider, then the chain of verifications going back to the beginning of the history is broken. To mitigate this threat, Frientegrity clients can tolerate up to f malicious users by looking back in the history until they find a point for which at least $f + 1$ different users

³An operation’s prevCommitment need not refer to the immediately preceding version. This could occur, for example, if the operation had been submitted concurrently with other operations.

⁴Specifically, an objectID is equal to the hash of its first operation, which contains a client-supplied random value, along with the provider’s name and a provider-supplied random value.

have vouched. Thus, in the example, if $f = 2$ and op_{13} , op_{14} , and op_{15} were each created by a different user, then Alice can rely on assurances from others about the history up to op_9 , but must check the following operations herself.

Frientegrity allows the application to use a different value of f for each type of object, and the appropriate f value depends on the context. For example, for an object representing a Twitter-like feed with a single trusted writer, setting $f = 0$ might be reasonable. By contrast, an object representing the wall of a large group with many writers might warrant a larger f value.

The choice of f impacts performance: as f increases, so does the number of operations that every client must verify. But when f is low relative to the number of writers, verifying an object requires logarithmic work in the history size due to the structure of history trees. We evaluate this security vs. performance trade-off empirically in §7.5.

4.2 Dependencies Between Objects

Recall that, for scalability, the provider only orders the operations submitted to an object with respect to other operations in the same object. As a result, Frientegrity only enforces fork* consistency on the history of operations within each object, but does not ordinarily provide any guarantees about the order of operations across different objects. When the order of operations spanning multiple objects is relevant, however, the objects' histories can be entangled through *dependencies*. A dependency is an assertion of the form $\langle \text{srcObj}, \text{srcVers}, \text{dstObj}, \text{dstVers}, \text{dstCommitment} \rangle$, indicating that the operation with version srcVers in srcObj happened after operation dstVers in dstObj , and that the server commitment to dstVers of dstObj was dstCommitment .

Dependencies are established by authorized clients in accordance with a policy specified by the application. When a client submits an operation to srcObj , it can create a dependency on dstObj by annotating the operation with the triple $\langle \text{dstObj}, \text{dstVers}, \text{dstCommitment} \rangle$. If another client subsequently reads the operation, the dependency serves as evidence that dstObj must have at least been at version dstVers at the time the operation was created, and the provider will be unable to trick the client into accepting an older version of dstObj .

As described in §3.4, Frientegrity uses dependencies to prevent a malicious provider from tricking clients into accepting outdated ACLs. Whenever a client submits a new operation to an object, it includes a dependency on the most recent version of the applicable ACL history that it has seen.⁵ Dependencies have other uses, however. For example, in a Twitter-like social network, every retweet could be annotated with a dependency on the original

⁵The annotation can be omitted if the prior operation in the object points to the same ACL history version.

tweet to which it refers. In that case, a provider that wished to suppress the original tweet would not only have to suppress all subsequent tweets from the original user (because Frientegrity enforces fork* consistency on the user's feed), it would also have to suppress all subsequent tweets from all the users who retweeted it.

Frientegrity uses *Merkle aggregation* [11] to implement dependencies efficiently. This feature of history trees allows the attributes of the leaf nodes to be aggregated up to the root, where they can be queried efficiently. In Frientegrity, the root of every object's history tree is annotated with a list of the other objects that the object depends on, along with those objects' most recent versions and server commitments. To prevent tampering, each node's annotations are included in its hash, so that incorrectly aggregated values will result in an incorrect root hash.

4.3 Verifiable Access Control

4.3.1 Supporting Membership Proofs

When handling a `readObject` RPC, Frientegrity ACLs must enable the provider to construct proofs that demonstrate to a client that every returned operation was created by an authorized user. But to truly demonstrate such authorization, such a proof must not only show that a user was present in the ACL at *some* point in time, it must show that the user was in the ACL at the time the operation was created (*i.e.*, in the version of the ACL on which the operation depends). As a result, an ACL must support queries not only on the current version of its state, but on previous versions as well. The abstract data type that supports both membership proofs and queries on previous versions is known as a *persistent authenticated dictionary (PAD)*. Thus, in Frientegrity, ACLs are PADs.

To realize the PAD abstract data type, an ACL is implemented as a binary search tree in which every node stores both an entry for a user and the hash of the subtree below it.⁶ To prove that an entry u exists, it suffices for the provider to return a pruned tree containing the search path from the root of the tree to u , in which unneeded subtrees in the path are replaced by stubs containing the subtrees' hashes. If the root hash of the search path matches the previously-known root hash of the full tree, a client can be convinced that u is in the ACL.

To support queries on previous versions of their states, ACLs are copy-on-write. When an administrator updates the ACL before calling `writeACL`, it does not modify any

⁶Our ACL construction expands on a PAD design from Crosby *et al.* [12] that is based on a *treap* [2]. A treap is a randomized search tree that is a cross between a tree and a heap. In addition to a key-value pair, every node has a priority, and the treap orders the nodes both according to their keys and according to the heap property. If nodes' priorities are chosen pseudorandomly, the tree will be balanced in expectation.

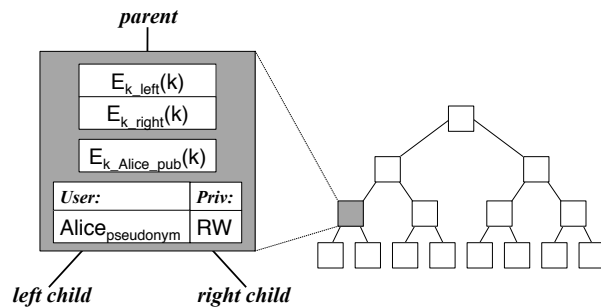


Figure 3: ACLs are organized as trees for logarithmic access time. Figure illustrates Alice's entry in Bob's ACL.

nodes directly. Instead, the administrator copies each node that needs to be changed, applies the update to the copy, and then copies all of its parents up to the root. As a result, there is a distinct root for every version of the ACL, and querying a previous version entails beginning a search at the appropriate root.

4.3.2 Preventing Equivocation about ACLs

To authenticate the ACL, it is not enough for an administrator to simply sign the root hash of every version, because a malicious provider could still equivocate about the history of ACL updates. To mitigate this threat, Frientegrity maintains a separate *ACL history* object that stores a log of updates to the ACL. An ACL history resembles an ordinary object, and clients check it for fork* consistency in the same way, but the operations that it contains are special *ModifyUserOps*. Each version of the ACL has a corresponding *ModifyUserOp* that stores the root hash as of that version and is signed by an administrator.

In summary, proving that the posts on a user Bob's wall were created by authorized users requires three steps. First, for each post, the provider must prove that the post's creator was in Bob's ACL by demonstrating a search path in the appropriate version of the ACL. Second, for each applicable version of Bob's ACL, the provider must provide a corresponding *ModifyUserOp* in Bob's ACL history that was signed by Bob. Finally, the provider must supply enough of the ACL history to allow clients to check it for fork* consistency, as described in §4.1.2.

4.3.3 Efficient Key Management and Revocation

Like many prior systems designed for untrusted servers (e.g., [5, 21, 25, 37]), Frientegrity protects the confidentiality of users' data by encrypting it under a key that is shared only among currently authorized users. When any user's access is revoked, this shared key must be changed. Unfortunately, in most of these prior systems, changing the key entails picking a new key and encrypting it under the public key of the remaining users, thereby making revocation expensive.

To make revocation more efficient, Frientegrity organizes keys into *key graphs* [59]. But rather than maintain-

ing a separate data structure for keys, Frientegrity stores keys in same ACL tree that is used for membership proofs. As shown in Figure 3, each node in Bob's ACL not only contains the pseudonym and privileges of an authorized user, such as Alice, it is also assigned a random AES key k . k is, in turn, encrypted under the keys of its left and right children, k_{left} and k_{right} , and under Alice's public key k_{Alice_pub} .⁷ This structure allows any user in Bob's ACL to follow a chain of decryptions up to the root of the tree and obtain the root key k_{Bob_root} . As a result, k_{Bob_root} is shared among all of Bob's friends and can be used to encrypt operations that only they can access. Because the ACL tree is balanced in expectation, the expected number of decryptions required to obtain k_{Bob_root} is logarithmic in the number of authorized users. More significantly, this structure makes revoking a user's access take logarithmic time as well. When a node is removed, only the keys along the path from the node to the root need to be changed and reencrypted.

4.3.4 Supporting Friends-of-Friends

Many social networking services, including Facebook, allow users to share content with an audience that includes not only their friends, but also their "friends-of-friends" (FoFs). Frientegrity could be extended naively to support sharing with FoFs by having Bob maintain a separate key tree, where each node corresponded to a FoF instead of a friend. This approach is undesirable, however, as the size of the resulting tree would be quadratic in the number of authorized users. Instead, Frientegrity stores a second FoF key k' in each node of Bob's ACL. Similar to the friend key k , k' is encrypted under the FoF keys of the node's left and right children, k'_{left} and k'_{right} . But instead of being encrypted under k_{Alice_pub} , k' is encrypted under k_{Alice_root} , the root key of Alice's ACL. Thus, any of Alice's friends can decrypt k' and ultimately obtain k'_{Bob_root} , which can be used to encrypt content for any of Bob's FoFs.

The FoF design above assumes, however, that friend relationships are symmetric: Bob must be in Alice's ACL in order to obtain k_{Alice_root} . To support asymmetric "follower-of-follower" relationships, such as Google+ "Extended Circles," Frientegrity could be extended so that a user Alice maintains a separate public-private key pair $\langle k_{Alice_FoF_pub}, k_{Alice_FoF_priv} \rangle$. Alice could then give $k_{Alice_FoF_priv}$ to her followers by encrypting it under k_{Alice_root} , and she could give $k_{Alice_FoF_pub}$ to Bob. Finally, Bob could encrypt k' under $k_{Alice_FoF_pub}$.

⁷To lower the cost of changing k , k is actually encrypted under an AES key k_{user} which is, in turn, encrypted under k_{Alice_pub} .

5. Extensions

5.1 Discovering Friends

Frientegrity identifies users by pseudonyms, such as the hashes of their public keys. But to enable users to discover new friends, the system must allow them to learn other users' real names under certain circumstances. In Frientegrity, we envision that the primary way a user would discover new friends is by searching through the ACLs of her existing friends for FoFs that she might want to "friend" directly. To make this possible, users could encrypt their real names under the keys that they use to share content with their FoFs. A user Alice's client could then periodically fetch and decrypt the real names of her FoFs and recommend them to Alice as possible new friends. Alice's client could rank each FoF according to the number of mutual friends that Alice and the FoF share by counting the number of times that the FoF appears in an ACL of one of Alice's friends.

Frientegrity's design prevents the provider from offering site-wide search that would allow any user to locate any other users by their real names. After all, if any user could search for any other user by real name, then so could Sybils acting on behalf of a malicious provider. We believe that this limitation is unavoidable, however, because there is an inherent trade-off between users' privacy and the effectiveness of site-wide search even in existing social networking systems.⁸ Thus, a pair of users who do not already share a mutual friend must discover each other, by exchanging their public keys out-of-band.

5.2 Multiple Group Administrators

As we describe in §4.3, when a user Alice reads another user Bob's wall, she verifies every wall post by consulting Bob's ACL. She, in turn, verifies Bob's ACL using Bob's ACL history, and then verifies each relevant `ModifyUserOp` by checking for Bob's signature. To support features like Facebook Groups or Pages, however, Frientegrity must be extended to enable multiple users to modify a single ACL and to allow these administrators be added and removed dynamically. But if the set of administrators can change, then, as with ordinary objects, a user verifying the ACL history must have a way to determine that every `ModifyUserOp` came from a user who was a valid administrator at the time the operation was created. One might think the solution to this problem is to have another ACL and ACL history just to keep track of which users are administrators at any given time. But this pro-

⁸For example, in 2009, Facebook chose to weaken users' privacy by forcing them to make certain information public, such as their genders, photos, and current cities. It adopted this policy, which it later reversed, so that it would be easier for someone searching for a particular user to distinguish between multiple users with the same name [46].

posal merely shifts the problem to the question of who is authorized to write to *these* data structures.

Instead, we propose the following design. Changes to the set of administrators would be represented as special `ModifyAdminOp`. Each `ModifyAdminOp` would be included in the ACL history alongside the `ModifyUserOp`, but would also have a pointer to the previous `ModifyAdminOp`. In this way, the `ModifyAdminOp` would be linked together to form a separate *admin history*, and clients would enforce fork* consistency on this history using a linear hash chain in the manner of BFT2F [33] and SPORC [21]. When a client verifies the ACL history, it would download and check the entire *admin history* thereby allowing it to determine whether a particular user was an administrator when it modified the ACL history. Although downloading an entire history is something that we have otherwise avoided in Frientegrity, the cost of doing so here likely is low: Even when the set of regular users changes frequently, the set of administrators typically does not.

5.3 Dealing with Conflicts

When multiple clients submit operations concurrently, conflicts can occur. Because servers do not have access to the operations' plaintexts, Frientegrity delegates conflict resolution to the clients, which can employ a number of strategies, such as last-writer-wins, operational transformation [21], or custom merge procedures [55]. In practice, however, many kinds of updates in social networking systems, such as individual wall posts, are *append* operations that are inherently commutative, and thus require no special conflict resolution.

5.4 Public Feeds with Many Followers

Well-known individuals and organizations often use their feeds on online social networks to disseminate information to the general public. These feeds are not confidential, but they would still benefit from a social networking system that protected their integrity. Such feeds pose scalability challenges, however, because they can have as many as tens of millions of followers.

Fortunately, Frientegrity can be readily adapted to support these feeds efficiently. Because the object corresponding to such a feed does not need to be encrypted, its ACL does not need to store encryption keys. The ACL is only needed to verify that every operation in the object came from an authorized writer. As a result, the size of the object's ACL need only be proportional to the number of users with *write* access to the object, which is often only a single user, rather than to the total number of followers.

Popular feeds would also not prevent applications from using dependencies to represent retweets in the manner described in §4.2. Suppose that Alice retweets a post from the feed of a famous individual, such as Justin Bieber.

Then, in such a design, the application would establish a dependency from Alice’s feed to Justin Bieber’s. But because dependencies only modify the source object (in this case Alice’s feed), they would not impose any additional performance penalty on reads of Justin Bieber’s feed. Thus, even if Justin Bieber’s posts are frequently retweeted, Frientegrity could still serve his feed efficiently.

6. Implementation

To evaluate Frientegrity’s design, we implemented a prototype that simulates a simplified Facebook-like service. It consists of a server that hosts a set of user profiles and clients that fetch, verify, and update them. Each user profile is comprised of an object representing the user’s “wall,” as well as an ACL and ACL history object representing the user’s list of friends. The wall object is made up of operations, each of which contains an arbitrary byte string, that have been submitted by the user or any of her friends. The client acts on behalf of a user and can perform RPCs on the server to read from and write to the walls of the user or user’s friends, as well as to update the user’s ACL. The client can simulate the work required to build a Facebook-like “news feed” by fetching and verifying the most recent updates to the walls of each of the user’s friends in parallel.

Our prototype is implemented in approximately 4700 lines of Java code (per SLOCCount [58]) and uses the `protobuf-socket-rpc` [16] library for network communication. To support the history trees contained in the wall and ACL history objects, we use the reference implementation provided by Crosby *et al.* [13].

Because Frientegrity requires every operation to be signed by its author and every server commitment to be signed by the provider, high signature throughput is a priority. To that end, our prototype uses the Network Security Services for Java (JSS) library from Mozilla [42] to perform 2048-bit RSA signatures because, unlike Java’s default RSA implementation, it is written in native code and offers significantly better performance. In addition, rather than signing and verifying each operation or server commitment individually, our prototype signs and verifies them in batches using *spliced signatures* [10, 13]. In so doing, we improve throughput by reducing the total number of cryptographic operations at the cost of a small potential increase in the latency of processing a single message.

7. Experimental Evaluation

Social networking applications place a high load on servers, and they require reasonably low latency in the face of objects containing tens of thousands of updates and friend lists reaching into the hundreds and thousands. This section examines how our Frientegrity prototype performs and scales under these conditions.

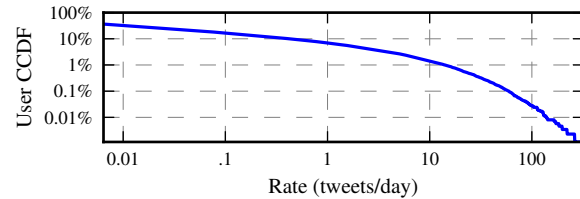


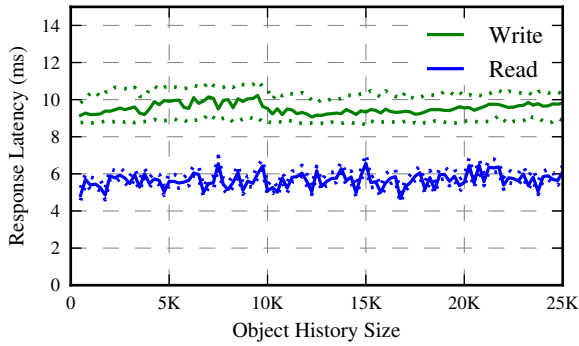
Figure 4: Distribution of post rates for Twitter users. 1% of users post at least 14 times a day, while 0.1% post at least 56 times a day.

All tests were performed on machines with dual 4-core Xeon E5620 processors clocked at 2.40 GHz, with 11 GB of RAM and gigabit network interfaces. Our evaluation ran with Oracle Java 1.6.0.24 and used Mozilla’s JSS cryptography library to perform SHA256 hashes and RSA 2048 signatures. All tests, unless otherwise stated, ran with a single client machine issuing requests to a separate server machine on the same local area network, and all data is stored in memory. A more realistic deployment over the wide-area would include higher network latencies (typically an additional tens to low hundreds of milliseconds), as well as backend storage access times (typically in low milliseconds in datacenters). These latencies are common to any Web service, however, and so we omit any such synthetic overhead in our experiments.

7.1 Single-object Read and Write Latency

To understand how large object histories may get in practice, we collected actual social network usage data from Twitter by randomly selecting over 75,000 users with public profiles. Figure 4 shows the distribution of post rates for Twitter users. While the majority of users do not tweet at all, the most active users post over 200 tweets per day, leading to tens of thousands of posts per year.

To characterize the effect of history size on read and write latency, we measured performance of these operations as the history size varies. For each read, the client fetched an object containing the five most recent operations along with any other required to verify the object. As shown in Figure 5, write latency was approximately 10 ms (as it includes both a server and client signature in addition to hashing), while read latency was approximately 6 ms (as it includes a single signature verification and hashing). The Figure’s table breaks down median request latency to its contributing components. As expected, a majority of the time was spent on public-key operations; a faster signature verification implementation or algorithm would correspondingly increase performance. While the latency here appears constant, independent of the history size, the number of hash verifications actually grows logarithmically with the history. This observed behavior arises because, at least up to histories of 25,000



Read	Server Data Fetches	0.45 ms	7.5%
	Network and Data Serialization	1.06 ms	17.5%
	Client Signature Verification	3.55 ms	58.8%
	Other (incl. Client Decrypt, Hashing)	0.98 ms	16.3%
	Total Latency	6.04 ms	
Write	Client Encryption	0.07 ms	0.7%
	Client Signature	4.45 ms	41.7%
	Network and Data Serialization	0.64 ms	6.0%
	Server Signature	4.31 ms	40.4%
	Other (incl. Hashing)	1.21 ms	11.3%
Total Latency	10.67 ms		

Figure 5: Read and write latency for Frientegrity as the object history size increases from 0 to 25000. Each data point represents the median of 1000 requests. The dots above and below the lines indicate the 90th and 10th percentiles for each trial. The table breaks down the cost of a typical median read and write request.

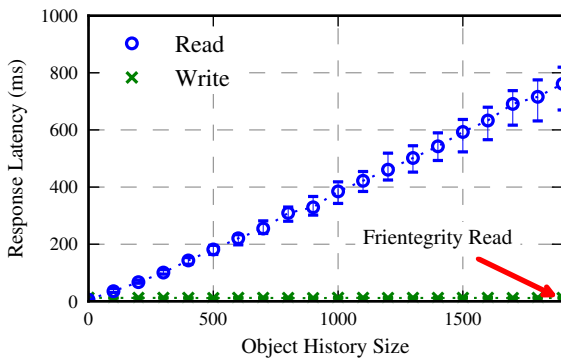


Figure 6: Latency for requests in a naive implementation using hash chains. The red arrow indicates the response time for Frientegrity read requests at an object size of 2000. Each data point is the median of 100 requests. The error bars indicate the 90th and 10th percentiles.

operations, the constant-time overhead of a public-key signature or verification continues to dominate the cost.

Next, we performed these same microbenchmarks on an implementation that verifies object history using a hash chain, rather than Frientegrity’s history trees. In this experiment, each client was stateless, and so it had to perform a complete verification when reading an object. This verification time grows linearly with the object history

Object	Signatures	7210 B
	History Tree Hashes	640 B
	Dependency Annotations	224 B
	Other Metadata	1014 B
ACL	ACL PAD	453 B
	Signatures in ACL History	1531 B
	Hashes in ACL History Tree	32 B
	Other Metadata	226 B
Total Overhead	11300 B	

Table 1: Sources of network overhead of a typical read of an object’s five most recent updates.

size, as shown in Figure 6. Given this linear growth in latency, verifying an object with history size of 25,000 operations would take approximately 10 s in the implementation based on a hash chain compared to Frientegrity’s 6 ms.

The performance of hash chains could be improved by having clients cache the results of previous verifications so they would only need to verify subsequent operations. Even if clients were stateful, however, Figure 4 shows that fetching the latest updates of the most prolific users would still require hundreds of verifications per day. Worse still, following new users or switching between client devices could require tens of thousands of verifications.

7.2 Network Overhead

When network bandwidth is limited, the size of the messages that Frientegrity sends over the network can impact latency and throughput. To understand this effect, we measure the overhead that Frientegrity’s verification and access control mechanisms add to an object that is fetched. Table 1 provides a breakdown of the sources of overhead in a read of the five most recent operations in an object. The object is comprised of 100 operations all created by a single writer. We assume that the ACL that applies to the object only contains a single user and his associated encrypted key and that the ACL history object contains only two operations (an initial operation and the operation that added the single user).

As shown in Table 1, the total overhead added by Frientegrity is 11,300 B, which would add approximately 90 ms of download time on a 1 Mbps link. Not surprisingly, the majority of the overhead comes from the signatures on individual operations and in prevCommitments. The object history tree contains 14 signatures, and the ACL history contains another four. Together, this many 2048-bit RSA bare signatures would require 4068 bytes, but because Frientegrity employs spliced signatures, they require additional overhead in exchange for faster signing and verification.

7.3 Latency of Fetching a News Feed

To present a user with a news feed, the client must perform one readObject RPC for each of the user's friends, and so we expect the latency of fetching a news feed to scale linearly with the number of friends. Because clients can hide network latency by pipelining requests to the server, we expect the cost of decryption and verification to dominate.

To evaluate the latency of fetching a news feed, we varied the number of friends from 1 to 50. We repeated each experiment 500 times and computed the median of the trials. A linear regression test on the results showed an overhead of 3.557 ms per additional friend (with a correlation coefficient of 0.99981). As expected, the value is very close to the cost of client signature verification and decryption from Figure 5.

Users in social networks may have hundreds of friends, however. In 2011, the average Facebook user had 190 friends, while the 90th percentile of users had 500 friends [19]. With Frientegrity's measured per-object overhead, fetching wall posts from all 500 friends would require approximately 1.8 s. In practice, we expect a social networking site to use modern Web programming techniques (*e.g.*, asynchronous Javascript) so that news feed items could be loaded in the background and updated incrementally while a user stays on a website. Even today, social networking sites often take several seconds to fully load.

7.4 Server Throughput with Many Clients

Social networks must scale to millions of active users. Therefore, to reduce capital and operational costs, it is important that a server be able to maximize throughput while maintaining low latency. To characterize a loaded server's behavior, we evaluated its performance as we increased the number of clients, all issuing requests to the same object. In this experiment, we ran multiple client machines, each with at most 4 clients. Each client issued 3000 requests sequentially, performing a 10 B write with a 1% probability and a read otherwise.

Figure 7 plots server throughput as the number of clients increases, as well as server latency as a function of load. We measured server latency from the time it received a request until the time that it started writing data back to its network socket. The server reached a maximal throughput of handling around 3500 requests per second, while median latency remained below 0.5 ms.

7.5 Effect of Increasing f

Frientegrity supports collaborative verification of object histories. The number of malicious clients that can be tolerated, f , has a large impact on client performance. As f increases, the client has to examine operations further back in the history until it finds $f + 1$ different writers. To

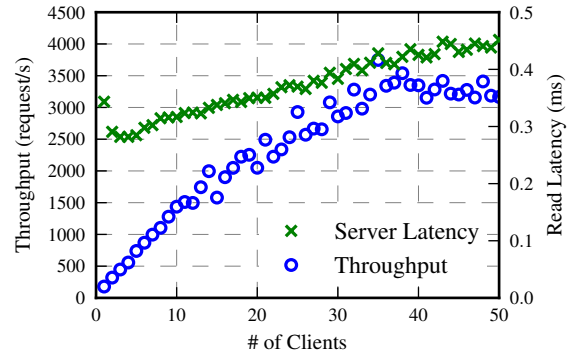


Figure 7: Server performance under increased client load. Each data point is the median of 5 runs.

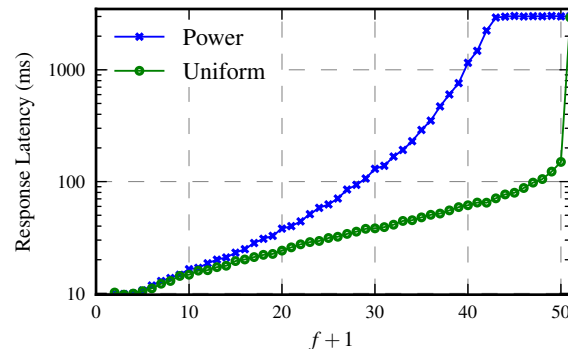


Figure 8: Performance implication of varying minimum set of trusted writers for collaborative verification.

understand this effect, we measured the read latency of a single object as f grows.

In this experiment, 50 writers first issued 5000 updates to the same object. We evaluated two different workloads for clients. In *uniform*, each writer had a uniform probability (2%) of performing the write; in *power law*, the writers were selected from a power-law distribution with $\alpha = 3.5$ (this particular α was the observed distribution of chat activity among users of Microsoft messaging [32]). A client then issued a read using increasing values of f . Read latencies plotted in Figure 8 are the median of 100 such trials.

In the uniform distribution, the number of required verifications rises slowly. But as $f + 1$ exceeds the number of writers, the client must verify the entire history. For the power law distribution, however, as f increases, the number of required verifications rises more rapidly, and at $f = 42$, the client must verify all 5000 updates. Nevertheless, this experiment shows that Frientegrity can maintain good performance in the face of a relatively large number of malicious users. Even with f at nearly 30, the verification latency was only 100 ms.

7.6 Latency of ACL Modifications

In social networking applications, operations on ACLs must perform well even when ACL sizes reach hundreds

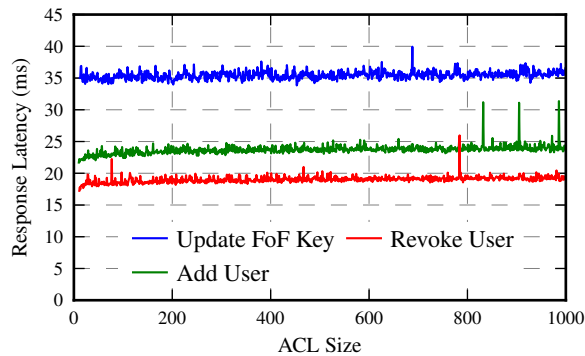


Figure 9: Latency of various ACL operations as a function of number of friends. Friend of Friend updates are measured as time to change a single user’s FoF key. Each data point is the mean of 100 runs.

of users. When a user Alice updates her ACL, she first fetches it and its corresponding ACL history and checks that they are consistent. In response, Alice’s friend Bob updates the key he shares with friends-of-friends (FoFs). To do so, he fetches and checks Alice’s ACL in order to retrieve her updated key. He then proceeds to fetch, check, and update his own ACL.

To evaluate the cost of these ACL operations, we measured Frientegrity’s performance as two users, Alice and Bob, make changes to their ACLs. While Alice added and removed users from her ACL, Bob updated the key he shares with FoFs. We performed this experiment for different ACL sizes and plotted the results in Figure 9.

As expected, updating the key shared with FoFs was the most costly operation because it requires verifying two ACLs instead of one. Furthermore, adding a new user to an ACL took longer than removing one because it requires a public key encryption. Finally, we observed that although modifying an ACL entails a logarithmic number of symmetric key operations, the cost of these operations was dominated by constant number of public key operations required to verify and update the ACL history.

8. Related Work

Decentralized approaches: To address the security concerns surrounding social networking, numerous works have proposed decentralized designs, in which the social networking service is provided by a collection of federated nodes. In Diaspora [17], perhaps the most well known of these systems, users can choose to store their data with a number of different providers called “pods.” In other systems, including Safebook [14], eXO [36], PeerSoN [6], porkut [44], and Confidant [35], users store their data on their own machines or on the machines of their trusted friends, and these nodes are federated via a distributed hash table. Still others, such as PrPI [48] and

Vis-à-Vis [49], allow users’ data to migrate between users’ own machines and trusted third-party infrastructure. We have argued, however, that decentralization is an insufficient approach. A user is left with an unenviable dilemma: either sacrifice availability, reliability, scalability, and convenience by storing her data on her own machine, or entrust her data to one of several providers that she probably does not know or trust any more than she would a centralized provider.

Cryptographic approaches: Many other works aim to protect social network users’ privacy via cryptography. Systems such as Persona [5], flyByNight [37], NOYB [25], and Contrail [53] store users’ data with untrusted providers but protect its contents with encryption. Others, such as Hummingbird [9], Lockr [56], and systems from Backes *et al.* [3], Domingo-Ferrer *et al.* [18] and Carminati *et al.* [8] attempt to hide a user’s social relationships as well, either from the provider or from other users. But, they do not offer any defenses against the sort of traffic analysis we describe in §2.3 other than decentralization. Unlike Frientegrity, in many of these systems (*e.g.*, [5, 9, 37]), “un-friending” requires work that is linear in the number of a user’s friends. The scheme of Sun *et al.* [54] is an exception, but it does not support FoFs. EASiER [28] aims to achieve efficient revocation via broadcast encryption techniques and a reencrypting proxy, but when deployed in the DECENT [29] distributed social network, it appears to perform poorly for reasons that are unclear. All of these systems, however, focus primarily on protecting users’ privacy while largely neglecting the integrity of users’ data. They either explicitly assume that third parties are “honest-but-curious” (*e.g.*, [9, 37]), or they at most employ signatures on individual messages. None deal with the prospect of provider equivocation, however.

Defending against equivocation: Several systems have addressed the threat of server equivocation in network file systems [33, 34], key-value stores [7, 38, 50], and group collaboration [21] by enforcing fork* consistency and related properties. But to enforce fork* consistency, they require clients to perform work that is linear in either the number of users or the number of updates ever submitted to the system. This overhead is impractical in social networks with large numbers of users and in which users typically are interested only in the latest updates.

FETHR [45] is a Twitter-like service that defends against server equivocation by linking a user’s posts together with a hash chain as well as optionally entangling multiple users’ histories. But besides not supporting access control, it lacks a formal consistency model. Thus, unless a client verifies a user’s entire history back to the beginning, FETHR provides no correctness guarantees.

9. Conclusion and Future Work

In designing Frienteegrity, we sought to provide a general framework for social networking applications built around an untrusted service provider. The system had to both preserve data confidentiality and integrity, yet also remain efficient, scalable, and usable. Towards these goals, we present a novel method for detecting server equivocation in which users collaborate to verify object histories, and more efficient mechanisms for ensuring fork* consistency based on history trees. Furthermore, we provide a novel mechanism for efficient access control by combining persistent authenticated dictionaries and key graphs.

In addition to introducing these new mechanisms, we evaluate a Frienteegrity prototype on synthetic workloads inspired by the scale of real social networks. Even as object histories stretch into the tens of thousands and access control lists into the hundreds, Frienteegrity provides response times satisfactory for interactive use, while maintaining strong security and integrity guarantees.

Like other social networking systems that store users' encrypted data with an untrusted provider [5, 25, 37, 53], Frienteegrity faces the problem of how such third-party infrastructure would be paid for. It has been suggested that providers would not accept a business model that would prevent them from mining the plaintext of users' data for marketing purposes. Whether this is so has not been well studied. Although there has been some work on privacy-preserving advertising systems [26, 57], the development of business models that can support privacy-preserving services hosted with third-party providers largely remains future work.

Acknowledgments We thank Andrew Appel, Matvey Arye, Wyatt Lloyd, and our anonymous reviewers for their insights and helpful comments. This research was supported by funding from NSF CAREER Award #0953197, an ONR Young Investigator Award, and a gift from Google.

References

- [1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM TOCS*, 14(1), 1996.
- [2] C. R. Aragon and R. G. Seidel. Randomized search trees. In *Proc. FOCS*, Oct. 1989.
- [3] M. Backes, M. Maffei, and K. Pecina. A security API for distributed social networks. In *Proc. NDSS*, Feb. 2011.
- [4] L. Backstrom, C. Dwork, and J. Kleinberg. Wherefore Art Thou R3579X? Anonymized social networks, hidden patterns, and structural steganography. In *Proc. WWW*, May 2007.
- [5] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin. Persona: an online social network with user-defined privacy. In *Proc. SIGCOMM*, Aug. 2009.
- [6] S. Buchegger, D. Schiöberg, L. hung Vu, and A. Datta. PeerSoN: P2P social networking early experiences and insights. In *Proc. SNS*, Mar. 2009.
- [7] C. Cachin, I. Keidar, and A. Shraer. Fail-aware untrusted storage. In *Proc. DSN*, June 2009.
- [8] B. Carminati and E. Ferrari. Privacy-aware collaborative access control in web-based social networks. In *Proc. DBSec*, July 2008.
- [9] E. D. Cristofaro, C. Soriente, G. Tsudik, and A. Williams. Hummingbird: Privacy at the time of twitter. Cryptology ePrint Archive, Report 2011/640, 2011. <http://eprint.iacr.org/>.
- [10] S. A. Crosby and D. S. Wallach. High throughput asynchronous algorithms for message authentication. Technical Report CS TR10-15, Rice University, Dec. 2010.
- [11] S. A. Crosby and D. S. Wallach. Efficient data structures for tamper-evident logging. In *Proc. USENIX Security*, Aug. 2009.
- [12] S. A. Crosby and D. S. Wallach. Super-efficient aggregating history-independent persistent authenticated dictionaries. In *Proc. ESORICS*, Sept. 2009.
- [13] S. A. Crosby and D. S. Wallach. Reference implementation of history trees and spliced signatures. <https://github.com/scrosby/fastsig>, Dec. 2010.
- [14] L. A. Cuttillo, R. Molva, T. Strufe, and T. Darmstadt. Safebook: A privacy-preserving online social network leveraging on real-life trust. *IEEE Communications Magazine*, 47(12):94–101, Dec. 2009.
- [15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. SOSP*, Oct. 2007.
- [16] S. Deo. protobuf-socket-rpc: Java and python protobuf rpc implementation using TCP/IP sockets (version 2.0). <http://code.google.com/p/protobuf-socket-rpc/>, May 2011.
- [17] Diaspora. Diaspora project. <http://diasporaproject.org/>. Retrieved April 23, 2012.
- [18] J. Domingo-Ferrer, A. Viejo, F. Sebé, and Írsula González-Nicolás. Privacy homomorphisms for social networks with private relationships. *Computer Networks*, 52:3007–3016, Oct. 2008.
- [19] Facebook, Inc. Anatomy of facebook. <http://www.facebook.com/notes/facebook-data-team/anatomy-of-facebook/10150388519243859>, Nov. 2011.
- [20] Facebook, Inc. Fact sheet. <http://newsroom.fb.com/content/default.aspx?NewsAreaId=22>. Retrieved April 23, 2012.
- [21] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. Sporc: Group collaboration using untrusted cloud resources. In *Proc. OSDI*, Oct. 2010.
- [22] Flickr. Flickr phantom photos. <http://flickr.com/help/forum/33657/>, Feb. 2007.
- [23] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. SOSP*, Oct. 2003.

- [24] Google, Inc. Transparency report. <https://www.google.com/transparencyreport/governmentrequests/userdata/>. Retrieved April 23, 2012.
- [25] S. Guha, K. Tang, and P. Francis. NOYB: Privacy in online social networks. In *Proc. WOSN*, Aug. 2008.
- [26] S. Guha, B. Cheng, and P. Francis. Privad: Practical privacy in online advertising. In *Proc. NSDI*, Mar. 2011.
- [27] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3), 1990.
- [28] S. Jahid, P. Mittal, and N. Borisov. EASiER: Encryption-based access control in social networks with efficient revocation. In *Proc. ASIACCS*, Mar. 2011.
- [29] S. Jahid, S. Nilizadeh, P. Mittal, N. Borisov, and A. Kapadia. DECENT: A decentralized architecture for enforcing privacy in online social networks. In *Proc. SESOC*, Mar. 2012.
- [30] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. STOC*, May 1997.
- [31] L. Lamport. The part-time parliament. *ACM TOCS*, 16(2): 133–169, 1998.
- [32] J. Leskovec and E. Horvitz. Planetary-scale views on a large instant-messaging network. In *Proc. WWW*, Apr. 2008.
- [33] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proc. NSDI*, Apr. 2007.
- [34] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. OSDI*, Dec. 2004.
- [35] D. Liu, A. Shakimov, R. Cáceres, A. Varshavsky, and L. P. Cox. Confidant: Protecting OSN data without locking it up. In *Proc. Middleware*, Dec. 2011.
- [36] A. Loupasakis, N. Ntarmos, and P. Triantafyllou. eXO: Decentralized autonomous scalable social networking. In *Proc. CIDR*, Jan. 2011.
- [37] M. M. Lucas and N. Borisov. flyByNight: mitigating the privacy risks of social networking. In *Proc. WPES*, Oct. 2008.
- [38] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *Proc. OSDI*, Oct. 2010.
- [39] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *Proc. PODC*, July 2002.
- [40] J. P. Mello. Facebook scrambles to fix security hole exposing private pictures. *PC World*, Dec. 2011.
- [41] R. C. Merkle. A digital signature based on a conventional encryption function. *CRYPTO*, pages 369–378, 1987.
- [42] Mozilla Project. Network security services for Java (JSS). <https://developer.mozilla.org/En/JSS>. Retrieved April 23, 2012.
- [43] A. Narayanan and V. Shmatikov. De-anonymizing social networks. In *Proc. IEEE S & P*, May 2009.
- [44] R. Narendula, T. G. Papaioannou, and K. Aberer. Privacy-aware and highly-available OSN profiles. In *Proc. WET-ICE*, June 2010.
- [45] D. R. Sandler and D. S. Wallach. Birds of a FETHR: Open, decentralized micropublishing. In *Proc. IPTPS*, Apr. 2009.
- [46] R. Sanghvi. Facebook blog: New tools to control your experience. <https://blog.facebook.com/blog.php?post=196629387130>, Dec. 2009.
- [47] E. Schonfeld. Watch out who you reply to on google buzz, you might be exposing their email address. *TechCrunch*, Feb. 2010.
- [48] S.-W. Seong, J. Seo, M. Nasielski, D. Sengupta, S. Hangal, S. K. Teh, R. Chu, B. Dodson, and M. S. Lam. PrPI: A decentralized social networking infrastructure. In *Proc. MCS*, June 2010.
- [49] A. Shakimov, H. Lim, R. Cáceres, L. P. Cox, K. Li, D. Liu, and A. Varshavsky. Vis-à-Vis: Privacy-preserving online social networking via virtual individual servers. In *Proc. COMSNETS*, Jan. 2011.
- [50] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *Proc. CCSW*, Oct. 2010.
- [51] S. Song. Why I left Sina Weibo. <http://songshinan.blog.caixin.cn/archives/22322>, July 2011.
- [52] M. Stonebraker. The case for shared nothing. *IEEE Database Engineering Bulletin*, 9(1):4–9, 1986.
- [53] P. Stuedi, I. Mohamed, M. Balakrishnan, Z. M. Mao, V. Ramasubramanian, D. Terry, and T. Wobber. Conrail: Enabling decentralized social networks on smartphones. In *Proc. Middleware*, Dec. 2011.
- [54] J. Sun, X. Zhu, and Y. Fang. A privacy-preserving scheme for online social networks with efficient revocation. In *Proc. INFOCOM*, Mar. 2010.
- [55] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. SOSP*, Dec. 1995.
- [56] A. Tootoonchian, S. Saroiu, Y. Ganjali, and A. Wolman. Lockr: Better privacy for social networks. In *Proc. CoNEXT*, Dec. 2009.
- [57] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy preserving targeted advertising. In *Proc. NDSS*, Feb. 2010.
- [58] D. Wheeler. SLOccount. <http://www.dwheeler.com/sloccount/>. Retrieved April 23, 2012.
- [59] C. K. Wong, M. Gouda, and S. S. Lam. Secure group communications using key graphs. *IEEE/ACM TON*, 8(1): 16–30, 1998.
- [60] M. Zuckerberg. Facebook S-1: Letter from Mark Zuckerberg. http://sec.gov/Archives/edgar/data/1326801/000119312512034517/d287954ds1.htm#toc287954_10, Feb. 2012.

Efficient and Scalable Socware Detection in Online Social Networks

Md Sazzadur Rahman, Ting-Kai Huang, Harsha V. Madhyastha, Michalis Faloutsos
Department of Computer Science and Engineering
University of California, Riverside

Abstract—

Online social networks (OSNs) have become the new vector for cybercrime, and hackers are finding new ways to propagate spam and malware on these platforms, which we refer to as socware. As we show here, socware cannot be identified with existing security mechanisms (e.g., URL blacklists), because it exploits different weaknesses and often has different intentions.

In this paper, we present MyPageKeeper, a Facebook application that we have developed to protect Facebook users from socware. Here, we present results from the perspective of over 12K users who have installed MyPageKeeper and their roughly 2.4 million friends. Our work makes three main contributions. First, to enable protection of users at scale, we design an efficient socware detection method which takes advantage of the *social context* of posts. We find that our classifier is both accurate (97% of posts flagged by it are indeed socware and it incorrectly flags only 0.005% of benign posts) and efficient (it requires 46 ms on average to classify a post). Second, we show that socware significantly differs from traditional email spam or web-based malware. For example, website blacklists identify only 3% of the posts flagged by MyPageKeeper, while 26% of flagged posts point to malicious apps and pages hosted on Facebook (which no current antivirus or blacklist is designed to detect). Third, we quantify the prevalence of socware by analyzing roughly 40 million posts over four months; 49% of our users were exposed to at least one socware post in this period. Finally, we identify a new type of parasitic behavior, which we refer to as “Like-as-a-Service”, whose goal is to artificially boost the number of “Likes” of a Facebook page.

1 Introduction

As online social networks (OSNs) are becoming the new epicenter of the web, hackers are expanding their territory to these services [8]. Anyone using Facebook or Twitter is likely to be familiar with what we call here **socware**¹: fake, annoying, possibly damaging posts from friends of the potential victim. The propagation of socware takes the form of postings and communications between

¹ We find the introduction of the term socware necessary because, as we elaborate later in Section 2, the types of intent associated with socware encompasses more than traditional phishing and malware.

friends on OSNs. Users are enticed into visiting suspicious websites or installing apps with the lure of false rewards (e.g., free iPads in memory of Steve Jobs [30]), and they unwittingly send the post to their friends, thus enabling a viral spreading. This is exactly where the power of socware lies: posts come with the implicit endorsement of the sending friend. Beyond this being a nuisance, socware also enables cyber-crime, with several Facebook scams resulting in loss of real money for users [11, 12].

Defenses against email spam are insufficient for identifying socware since reputation-based filtering [29, 28, 51] is insufficient to detect socware received from friends and, as we show later, the keywords used in email spam significantly differ from those used in socware. We also find that URL blacklists designed to detect phishing and malware on the web do not suffice, e.g., because a large fraction of socware (26% in our dataset) points to suspicious applications hosted on Facebook. Finally, though Facebook has its own mechanisms for detecting and removing malware [52], they seem to be less aggressive either due to what they define as malware or due to computational limitations.

In this paper, we present the design and implementation of a Facebook application, MyPageKeeper [24], that we develop specifically for the purpose of protecting Facebook users from socware. For any subscribing user of MyPageKeeper, whenever socware appears in that user’s wall or news feed, we seek to detect the socware soon thereafter and alert the user (hopefully before she views the post). Until October 2011, MyPageKeeper had been installed by more than 12K Facebook users (since its launch in June 2011). By monitoring the news feeds of these users, we also observe posts on the walls of the 2.4 million friends of these users. In this paper, we evaluate MyPageKeeper using a dataset of over 40 million posts that it inspected during the four month period from June to October 2011.

The key contributions of our work can be grouped into three main thrusts.

a. Designing an accurate, efficient, and scalable detection method. In order to operate MyPageKeeper at

scale, but at low cost, the distinguishing characteristic of our approach is our strident focus on efficiency. Prior solutions for detecting spam and malware on OSNs (which we describe in detail later) rely on information obtained either by crawling the URLs included in posts or by performing DNS resolution on these URLs. In contrast, our socware classifier relies solely on the *social context* associated with each post (e.g., the number of walls and news feeds in which posts with the same embedded URL are observed, and the similarity of text descriptions across these posts). Note that this approach means that we do not even resolve shortened URLs (e.g., using services like bit.ly) into the full URLs that they represent. This approach maximizes the rate at which we can classify posts, thus reducing the cost of resources required to support a given population of users.

We employ a Machine Learning classification module using Support Vector Machines on a carefully selected set of such features that are readily available from the observed posts. 97% of posts flagged by our classifier are indeed socware and it incorrectly flags only 0.005% of benign posts. Furthermore, it requires an average of only 46 ms to classify a post.

b. Socware is a new kind of malware. We show that socware is significantly different than traditional email spam or web-based malware. First, URL blacklists cannot detect socware effectively. These blacklists identify only 3% of the malicious posts that MyPageKeeper flags. The inability of website blacklists to identify socware is partly due to the fact that a significant fraction of socware is hosted on popular blogging domains such as `blogspot.com` and on Facebook itself. Specifically, 26% of the flagged posts point to Facebook apps or pages. Moreover, we also observe a low overlap between the keywords associated with email based spam and those we find in socware.

c. Quantifying socware: prevalence and intention. We find that 49% of our users were exposed to at least one socware post in four months. We also identify a new type of parasitic behavior, which we refer to as “Like-as-a-Service”. Its goal is to artificially boost the number of “Likes” of a Facebook page. With the lure of games and rewards, several Facebook apps push users to *Like* the Facebook pages of say a store or a product, thus artificially inflating their reputation on Facebook. This further confirms the difference between socware and other forms of malware propagation.

2 Socware on Facebook

In this section, we provide relevant background about Facebook, and we describe typical characteristics of socware found on Facebook.

2.1 The Facebook terminology

Facebook is the largest online social network today with over 900 million registered users, roughly half of whom visit the site daily. Here, we discuss some standard Facebook terminology relevant to our work.

- *Post*: A post represents the basic unit of information shared on Facebook. Typical posts either contain only text (status updates), a URL with an associated text description, or a photo/album shared by a user. In our work, we focus on posts that contain URLs.
- *Wall*: A Facebook user’s wall is a page where friends of the user can post messages to the user. Such messages are called wall posts. Other than to the user herself, posts on a user’s wall are visible to other users on Facebook determined by the user’s privacy settings. Typically a user’s wall is made visible to the user’s friends, and in some cases to friends of friends.
- *News feed*: A Facebook user’s news feed page is a summary of the social activity of the user’s friends on Facebook. For example, a user’s news feed contains posts that one of the user’s friends may have shared with all of her friends. Facebook continually updates the news feed of every user and the content of a user’s news feed depends on when it is queried.
- *Like*: Like is a Facebook widget that is associated with an object such as a post, a page, or an app. If a user clicks the Like widget associated with an object, the object will appear in the news feed of the user’s friends and thus allow information about the object to spread across Facebook. Moreover, the number of Likes (i.e., the number of users who have clicked the Like widget) received by an object also represents the reputation or popularity of the object.
- *Application*: Facebook allows third-party developers to create their own applications that Facebook users can add. Every time a user visits an application’s page on Facebook, Facebook dynamically loads the content of the application from a URL, called the canvas URL, pointing to the application server provided by the application’s developer. Since content of an application is dynamically loaded every time a user visits the application’s page on Facebook, the application developer enjoys great control over content shown in the application page. The Facebook platform uses OAuth 2.0 [2] for user authentication, application authorization and application authentication. Here, application authorization ensures that the users grant precise data (e.g., email address) and capabilities (e.g., ability to post on the user’s wall) to the applications they choose to add, and application authentication ensures that a user grants access to her data to the correct application.

2.2 Socware

We start by defining the meaning of *socware*. We describe typical characteristics of socware and elaborate on how socware distinguishes itself from traditional email spam and web malware.

What is socware? Our intention is to use the term socware to encompass all criminal and parasitic behavior in an OSN, including anything that annoys, hurts, or makes money off of the user. In the context of this paper, we consider a Facebook post as malicious, if it satisfies one of the following conditions: (1) the post spreads malware and compromises the device of the user, (2) the web page pointed to by the post requires the user to give away personal information, (3) the post promises false rewards (e.g., free products), (4) the post is made on a user's behalf without the user's knowledge (typically by having previously lured the user into providing relevant permissions to a rogue Facebook app), (5) the web page pointed to by the post requires the user to carry out tasks (e.g., fill out surveys) that help profit the owner of that website, or (6) the post causes the user to artificially inflate the reputation of the page (e.g., by forcing the user to 'Like' the page). While the first two criteria are typical malware and phishing, the latter four are distinctive of socware.

Disclaimer. As with email spam, there can be some ambiguity in the definition of socware: a post considered as annoying by one user may be considered useful by another user. In practice, our ultimate litmus test is the opinion of MyPageKeeper's users: if most of them report a post as annoying, we will flag it as such.

How does socware work? Socware appears in a Facebook user's wall or news feed typically in the form of a post which contains two parts. First, the post contains a URL², usually obfuscated with a URL shortening service, to a webpage that hosts either malicious or spam content. Second, socware posts typically contain a catchy text message (e.g., "*two free Southwest tickets*") that entice users to click on the URL included in the post. Optionally, socware posts also contain a thumbnail screenshot of the landing page of the URL, also used to entice the user to click on the link. For example, a purported image of Osama's corpse is included in a post that claims to point to a video of his death.

The operation of most socware epidemics can be associated with two distinct mechanisms.

a. Propagation mechanism. Once a user follows the embedded URL to the target website, the post tries to propagate itself through that user. For this, the user is often asked to complete several steps in order to obtain

²We leave the identification of socware posts that do not contain an URL for future work. The propagation of socware is harder in such cases, since the user needs to perform a more laborious operation (e.g., enter an URL into the browser's address bar) than simply clicking on the embedded URL.

App Name	Application Message	Monthly Active Users
Free Phone Calls	I'm making a Free Call with the Free Phone Call Facebook App! ... I'll never pay for a phone call again. Make your free call at URL	435,392
The App	Check if a friend has deleted you URL	35,216
The App	Check if a friend has deleted you URL	25,778

Table 1: Three rogue Facebook applications identified by MyPageKeeper.

Page Name	Message to persuade 'Like'	No. of Likes
Clif Bar	Hey there! Looking for a cliff builder's coupon? Just like us by clicking the button above. thanks!	79919
FarmVille Bonus	You can't claim you you haven't clicked on the like button	94907
Courtesy Chevrolet	Like our page to play and have a chance to win!	86287
Greggs The Bakers	Like us to claim your voucher	288039
Mobilink Infinity	Like us for big infinite fun	26105

Table 2: Top five pages identified by MyPageKeeper that persuade users to 'Like' them.

the fake reward (e.g., "Free Facebook T-shirt"). These steps involve "Liking" or "sharing" the post, or posting the socware on the user's wall. Thus, the cycle continues with the friends of that user, who see the post in their news feed. In contrast, users seldom forward email spam to their friends.

b. Exploitation mechanism. The exploitation often starts after the propagation phase. The hacker attempts either to learn the user's private information via a phishing attack [9], to spread malware to user devices, or to make money by "forcing" a particular user action or response, such as completing a survey for which the hacker gets paid [19].

Where is socware hosted? Socware can be broadly classified into two categories based on the infrastructure that hosts them.

a. Socware hosted outside Facebook: In this category, URLs point to a domain hosted outside Facebook. Since the URL points to a landing page outside the OSN, hackers can directly launch the different kinds of attacks mentioned above once a user visits the URL in a socware post. Though several URL blacklists should be able to flag such URLs, the process of updating these blacklists is too slow to keep up with the viral propagation of socware on OSNs [44].

b. Socware hosted on Facebook: A significant fraction of socware is hosted on Facebook itself: the embedded URL points to a Facebook page or application. Naturally, current blacklists and reputation-based schemes fail to flag such URLs. Such URLs typically point to the following types of Facebook objects:

- **Malicious Facebook applications:** Rogue applica-

tions post catchy messages (e.g., “*Check who deleted you from your profile*”) on the walls of users with a link pointing to the installation page of the application. Table 1 lists three such socware-spreading applications in our data. Users are conned into installing the application to their profile and granting several permissions to it. The application then not only gets access to that user’s personal information (such as email address, home town, and high school) but also gains the ability to post on the victim’s wall. As before, posts on a user’s wall typically appear on the news feeds of the user’s friends, and the propagation cycle repeats. Creating such applications has become easy with ready to use toolkits starting at \$25 [18].

- **Malicious Facebook events:** Sometimes hackers create Facebook events that contain a malicious link. One such event is the ‘*Get a free Facebook T-Shirt (Sponsored by Reebok)*’ scam. This event page states that 500,000 users will get a free T-shirt from Facebook. To be one among those 500,000 users, a user must attend the event, invite her friends to join, and enter her shipping address.
- **Malicious Facebook pages:** Another approach taken by hackers to spread socware is to create a Facebook page and post spam links on the page [27]. We also identified a trend in aggressive marketing by companies that force users to click “Like” on their Facebook pages to spread their pages as well as increase the reputation of the page. Table 2 lists the top five such Facebook pages, along with the message on the page and the number of Likes received by these pages.

3 MyPageKeeper Architecture

To identify socware and protect Facebook users from it, we develop MyPageKeeper. MyPageKeeper is a Facebook application that continually checks the walls and news feeds of subscribed users, identifies socware posts, and alerts the users. We present our goals in designing MyPageKeeper, and then describe the system architecture and implementation details.

3.1 Goals

We design MyPageKeeper with the following three primary goals in mind.

1. Accuracy. Our foremost goal is to ensure accurate identification of socware. We are faced with the obvious trade-off between missing malware posts (false negatives), and “crying wolf” too often (false positives). Although one could argue that minimizing false negatives is more important, users would abandon overly sensitive detection methods, as recognized by the developers of Facebook’s Immune System [52].

2. Scalability. Our end goal is to have MyPageKeeper provide protection from socware for all users on Face-

book, not just for a select few. Therefore, the system must be scalable to easily handle increased load imposed by a growth in the number of subscribed users.

3. Efficiency. Finally, we seek to minimize our costs in operating MyPageKeeper. The period between when a post first becomes visible to a user and the time it is checked by MyPageKeeper represents the window of vulnerability when the user is exposed to potential socware. To minimize the resources necessary to keep this window of vulnerability short, MyPageKeeper’s techniques for classification of posts must be efficient.

3.2 MyPageKeeper components

MyPageKeeper consists of six functional modules.

a. User authorization module. We obtain a user’s authorization to check her wall and news feed through a Facebook application, which we have developed. Once a user installs the MyPageKeeper application, we obtain the necessary credentials to access the posts of that user. For alerting the user, we also request permission to access the user’s email address and to post on the user’s wall and news feed. Figure 1(a) shows how a Facebook user authorizes an application.

b. Crawling module. MyPageKeeper periodically collects the posts in every user’s wall and news feed. As mentioned previously, we currently focus only on posts that contain a URL. Apart from the URL, each post comprises several other pieces of information, such as a text message associated with the post, the user who made the post, number of comments and Likes on the post, and the time when the post was created.

c. Feature extraction module. To classify a post, MyPageKeeper evaluates every embedded URL in the post. Our key novelty lies in considering only the social context (e.g., the text message in the post, and the number of Likes on it) for the classification of the URL and the related post. Furthermore, we use the fact that we are observing more than one user, which can help us detect an epidemic spread. We discuss these features in more detail later in Section 3.3.

d. Classification module. The classification module uses a Machine Learning classifier based on Support Vector Machines, but also utilizes several local and external whitelists and blacklists that help speed up the process and increase the overall accuracy. The classification module receives a URL and the related social context features extracted in the previous step. Since the classification is our key contribution, we discuss this in more detail in Section 3.3. If a URL is classified as socware, all posts containing the URL are labeled as such.

e. Notification module. The notification module notifies all users who have socware posts in their wall or news feed. The user can currently specify the notification mechanism, which can be a combination of emailing the

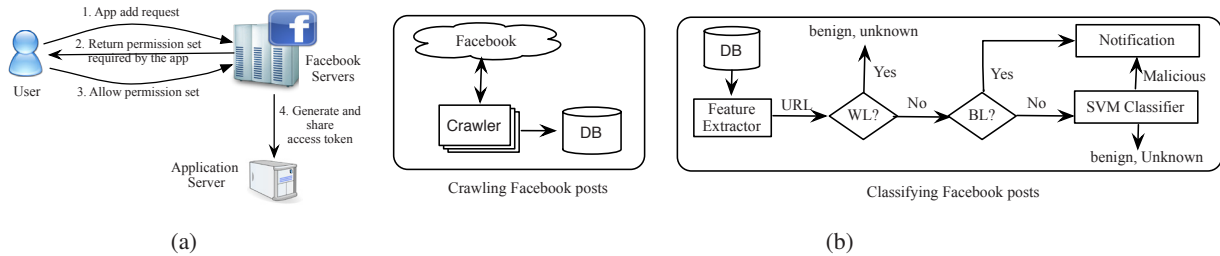


Figure 1: (a) Application installation process on Facebook, and (b) architecture of MyPageKeeper.

user or posting a comment on the suspect posts. In the future, we will consider allowing our system to remove the malicious post automatically, but this can create liabilities in the case of false positives.

f. User feedback module. Finally, to improve MyPageKeeper’s ability to detect socware, we leverage our user community. We allow users to report suspicious posts through a convenient user-interface. In such a report, the user can optionally describe the reason why she considers the post as socware.

3.3 Identification of socware

The key novelty of MyPageKeeper lies in the classification module (summarized in Figure 1(b)). As described earlier, the input to the classification module is a URL and the related social context features extracted from the posts that contain the URL. Our classification algorithm operates in two phases, with the expectation that URLs and related posts that make it through either phase without a match are likely benign and are treated as such.

Using whitelists and blacklists. To improve the efficiency and accuracy of our classifier, we use lists of URLs and domains in the following two steps. First, MyPageKeeper matches every URL against a whitelist of popular reputable domains. We currently use a whitelist comprising the top 70 domains listed by Quantcast, excluding domains that host user-contributed content (e.g., OSNs and blogging sites). Any URL that matches this whitelist is deemed safe, and it is not processed further.

Second, all the URLs that remain are then matched with several URL blacklists that list domains and URLs that have been identified as responsible for spam, phishing, or malware. Again, the need to minimize classification latency forces us to only use blacklists that we can download and match against locally. Such blacklists include those from Google’s Safe Browsing API [17], Malware Patrol [23], PhishTank [26], APWG [1], SpamCop [28], joewein [20], and Escrow Fraud [7]. Querying blacklists that are hosted externally, such as SURBL [31], URIBL [33] and WOT [34], will introduce significant latency and increase MyPageKeeper’s latency in detecting socware, thus inflating the window of vulnerability. Any URL that matches any of the blacklists that we use is classified as socware.

Using machine learning with social context features. All URLs that do not match the whitelist or any of the blacklists are evaluated using a Support Vector Machines (SVM) based classifier. SVM is widely and successfully used for binary classification in security and other disciplines [49, 46] [32]. We train our system with a batch of manually labeled data, that we gathered over several months prior to the launch of MyPageKeeper. For every input URL and post, the classifier outputs a binary decision to indicate whether it is malicious or not. Our SVM classifier uses the following features.

Spam keyword score. Presence of spam keywords in a post provides a strong indication that the post is spam. Some examples of such spam keywords are ‘FREE’, ‘Hurry’, ‘Deal’, and ‘Shocked’. To compile a list of such keywords that are distinctive to socware, our intuition is to identify those keywords that 1) occur frequently in socware posts, and 2) appear with a greater frequency in socware as compared to their frequency in benign posts.

We compile such a list of keywords by comparing a dataset of manually identified socware posts with a dataset of posts that contain URLs that match our whitelist (we discuss how to maintain this list of keywords in Section 7). We transform posts in either dataset to a bag of words with their frequency of occurrence. We then compute the likelihood ratio p_1/p_2 for each keyword where $p_1 = p(\text{word}|\text{socware post})$ and $p_2 = p(\text{word}|\text{benign post})$. The likelihood ratio of a keyword indicates the bias of the keyword appearing more in socware than in benign posts. In our current implementation of MyPageKeeper, we have found that the use of the 6 keywords with the highest likelihood ratio values among the 100 most frequently occurring keywords in socware is sufficient to accurately detect socware.

Thereafter, to classify a URL, MyPageKeeper searches all posts that contain the URL for the presence of these spam keywords and computes a spam keyword score as the ratio of the number of occurrences of spam keywords across these posts to the number of posts.

Message similarity. If a post is part of a spam campaign, it usually contains a text message that is similar to the text in other posts containing the same URL (e.g., because users propagate the post by simply sharing it). On the other hand, when different users share the same

popular URL, they are likely to include different text descriptions in their posts. Therefore, greater similarity in the text messages across all posts containing a URL portends a higher probability that the URL leads to spam. To capture this intuition, for each URL, we compute a message similarity score that captures the variance in the text messages across all posts that contain the URL. For each post, MyPageKeeper sums the ASCII values of the characters in the text message in the post, and then computes the standard deviation of this sum across all the posts that contain the URL. If the text descriptions in all posts are similar, the standard deviation will be low.

News feed post and wall post count. The more successful a spam campaign, the greater the number of walls and news feeds in which posts corresponding to the campaign will be seen. Therefore, for each URL, MyPageKeeper computes counts of the number of wall posts and the number of news feed posts which contained the URL.

Like and comment count. Facebook users can ‘Like’ any post to indicate their interest or approval. Users can also post comments to follow up on the post, again indicating their interest. Users are unlikely to ‘Like’ posts pointing to socware or comment on such posts, since they add little value. Therefore, for every URL, MyPageKeeper computes counts of the number of Likes and number of comments seen across all posts that contain the URL.

URL obfuscation. Hackers often try to spread malicious links in an obfuscated form, e.g., by shortening it with a URL shortening service such as *bit.ly* or *goo.gl*. We store a binary feature with every URL that indicates whether the URL has been shortened or not; we maintain a list of URL shorteners.

Note that none of the above features by themselves are conclusive evidence of socware, and other features could potentially further enhance the classifier (e.g., we can account for spam keywords such as ‘free’ included in URLs such as <http://nfljerseyfree.com>). However, as we show later in our evaluation, the features that we currently consider yield high classification accuracy in combination.

3.4 Implementing MyPageKeeper

We provide some details on MyPageKeeper’s implementation.

Facebook application. First, we implement the MyPageKeeper Facebook application using FBML [14]. We implement our application server using Apache (web server), Django (web framework), and Postgres (database). Once a user installs the MyPageKeeper app in her profile, Facebook generates a secret access token and forwards the token to our application server, which we then save in a database. This token is used by the crawler to crawl the walls and news feeds of subscribed users using the Facebook open-graph API. If any user deactivates

Data	Total	# distinct URLs
MyPageKeeper users	12,456	-
Friends of MyPageKeeper users	2,370,272	-
News feed posts	38,764,575	29,522,732
Wall posts	1,760,737	1,532,055
User reports	679	333

Table 3: Summary of MyPageKeeper data.

MyPageKeeper from their profile, Facebook disables this token and notifies our application server, whereupon we stop crawling that user’s wall and news feed.

Crawler instances and frequency. We run a set of crawlers in Amazon EC2 instances to periodically crawl the walls and news feeds of MyPageKeeper’s users. The set of users are partitioned across the crawlers. In our current instantiation, we run one crawler process for every 1,000 users. Thus, as more users subscribe to MyPageKeeper, we can easily scale the task of crawling their walls and news feeds by instantiating more EC2 instances for the task. Our Python-based crawlers use the open-graph API, incorporating users’ secret access tokens, to crawl posts from Facebook. Once the data is received in JSON format, the crawlers parse the data and save it in a local Postgres database.

Currently, as a tradeoff between timeliness of detection and resource costs on EC2, we instantiate MyPageKeeper to crawl every user’s wall and news feed once every two hours. Every couple of hours, all of our crawlers start up and each crawler fetches new posts that were previously not seen for the users assigned to it. Once all crawlers complete execution, the data from their local databases is migrated to a central database.

Checker instances. Checker modules are used to classify every post as socware or benign. Every two hours, the central scheduler forks an appropriate number of checker modules determined by the number of new URLs crawled since the last round of checking. Thus, the identification of socware is also scalable since each checker module runs on a subset of the pool of URLs. Each checker evaluates the URLs it receives as input—using a combination of whitelists, blacklists, and a classifier—and saves the results in a database. We use the *libsvm* [41] library for SVM based classification. Once all checker modules complete execution, notifiers are invoked to notify all users who have posts either on their wall or in their news feed that contain URLs that have been flagged as socware.

4 Evaluation

Next, we evaluate MyPageKeeper from three perspectives. First, we evaluate the accuracy with which it classifies socware. Second, we determine the contribution of MyPageKeeper’s social context based classifier in identifying socware compared to the URL blacklists that it uses. Lastly, we compare MyPageKeeper’s efficiency

Feature	F-Score
URL obfuscated?	0.300378
Spam keyword score	0.262220
# of news feed posts	0.173836
Message similarity score	0.131733
# of Likes	0.039895
# of wall posts	0.019857
# of comments	0.006367

Table 4: Feature scores used by MyPageKeeper’s classifier.

Alternative source	# of posts
Flagged by blacklist	18,923
Flagged by on.fb.me	2,102
Content deleted by Facebook	3,918
Blacklisted app	1,290
Blacklisted IP	5,827
Domain is deleted	247
Points to app install	4,658
Spamming app	6,547
Manually verified	14,876
True positives	58,388 (97%)
Unknown	1,803 (3%)
Total	60,191

Table 5: Validation of socware flagged by MyPageKeeper classifier.

with alternative approaches that would either crawl every URL or at least resolve short URLs in order to identify socware.

Table 3 summarizes the dataset of Facebook posts on which we conduct our evaluation. This data is obtained during MyPageKeeper’s operation over a four month period from 20th June to 19th October, 2011. MyPageKeeper had over 12K users during this period, who had around 2.37M friends in union. Our data comprises 38.7 million and 1.7 million posts that contain URLs from the news feeds and walls of these 12K users. We consider only those posts that contain URLs since MyPageKeeper currently checks only such posts. Overall, these 40 million posts contained around 30 million unique URLs. In addition, we received 679 reports of socware from 533 distinct MyPageKeeper users during the four month period, with 333 distinct URLs across these reports. Though it is hard to make any general claims with regard to representativeness of our data, we find that several user metrics (e.g., the male-to-female ratio and the distribution of users across age groups) closely match that of the Facebook user base at-large.

4.1 Accuracy

As previously mentioned, MyPageKeeper first matches every URL to be checked against a whitelist. If no match is found, it checks the URL with a set of locally queryable URL blacklists. Finally, MyPageKeeper applies its social context based classifier learned using the SVM model. In this process, we assume URL information provided by whitelists and blacklists to be ground truth, i.e., classification provided by them need not be independently validated. Therefore, we focus here on validating the

App name	Description	# of posts
Sendible	Social Media Management	6,687
iRazoo	Search & win!	1,853
4Loot	4Loot lets you win all sorts of Loot while searching the web	1,891

Table 6: Top three spamming applications in our dataset.

socware flagged by MyPageKeeper’s classifier based on social context features.

We trained MyPageKeeper’s classifier using a manually verified dataset of URLs that contain 2,500 positive samples and 5,000 negative samples of socware posts; we gathered these samples over several months while developing MyPageKeeper. Table 4 shows the importance of the various features in the SVM classifier learned. During the course of MyPageKeeper’s operation over four months, we applied the classifier to check 753,516 unique URLs; these are URLs that do not match the whitelist or any of the blacklists. Of these URLs, the classifier identified 4,972 URLs, seen across 60,191 posts, as instances of socware.

It is important to note that when MyPageKeeper sees a URL in multiple posts over time, the values of the features associated with the URL may change every time it appears, e.g., the message similarity score associated with the URL can change. However, once MyPageKeeper classifies a URL as socware during any of its occurrences, it flags all previously seen posts that contain the URL and notifies the corresponding users. Therefore, in evaluating MyPageKeeper’s classifier, URL blacklists, or MyPageKeeper as a whole, we consider here that a technique classified a particular URL as socware if that URL was flagged by that technique upon any of the URL’s occurrences. Correspondingly, we consider a URL to have not been classified as socware if it was not identified as such during any of its occurrences.

Checking the validity of socware identified by MyPageKeeper’s classifier is not straightforward, since there is no ground truth for what represents socware and what does not. However, here we attempt to evaluate the positive samples of socware identified by MyPageKeeper’s classifier using a combination of a host of complementary techniques (we later discuss in Section 7 the validation of posts that are deemed safe by MyPageKeeper). To do so, we use an instrumented Firefox browser to crawl the 4,972 URLs flagged by MyPageKeeper at the end of the four month period of MyPageKeeper operation. For every URL that we crawl, we record the landing URL, the IP address and other whois information of the landing domain, and contents of the landing page. To verify the reputation of every URL, we then apply several techniques in the order summarized in Table 5.

- *Blacklisted URLs:* First, we check if any of the URLs or the corresponding landing URLs are found in any

URL blacklists. Note that, though we use blacklists in the operation of MyPageKeeper itself, we use only those that can be stored and queried locally. Therefore, here we use for validation other external blacklists for which we have to issue remote queries. Further, even for blacklists used in MyPageKeeper, they may not identify some instances of socware when they initially appear because blacklists have been found to lag in keeping up with the viral propagation of spam on OSNs [44]. Hence, we check if a URL identified as socware by MyPageKeeper’s classifier appeared in any of the blacklists used by MyPageKeeper at a later point in time, even though it did not appear in any of those blacklists initially when MyPageKeeper spotted posts containing that URL.

- *Flagged by fb.me URL shortener:* Many URLs posted on Facebook are shortened using Facebook’s URL shortener `fb.me`. When Facebook determines any link shortened using their service to be unsafe, the corresponding shortened URL thereafter redirects to Facebook’s home page—`facebook.com/home.php`—instead of the actual landing page. Of the URLs flagged by MyPageKeeper’s classifier, we check if those shortened using Facebook’s URL shortening service redirect to Facebook’s home page.
- *Content deleted from Facebook:* If Facebook determines any URL hosted under the `facebook.com` domain to be unsafe (e.g., the page for a spamming Facebook application), it thereafter redirects that URL to `facebook.com/4oh4.php`. We use this as another source of information to validate URLs flagged by MyPageKeeper’s classifier.
- *Blacklisted apps:* If the URLs in posts made by a Facebook app are flagged due to any of the above reasons, we consider that app to be malicious and declare all other URLs posted by it as unsafe, thus helping validate some of the URLs declared as socware by MyPageKeeper’s classifier.
- *Blacklisted IPs:* For every URL flagged by any of the above techniques, we record the IP address when that URL is crawled and blacklist that IP. Of the URLs flagged by MyPageKeeper’s classifier, we then consider those that lead to one of these blacklisted IP addresses as correctly classified.
- *Domain deleted:* Malicious domains are often deleted once they are caught serving malicious content. Therefore, we deem MyPageKeeper’s positive classification of a URL to be correct if the domain for that URL no longer exists when we attempt to crawl it.
- *Obfuscation of app installation page:* Posts made by Facebook applications to attract users to install them typically include an un-shortened URL pointing to a Facebook page that contains information about the ap-

Source	# (%) of URLs	# (%) of posts	Overlap with classifier (# of URLs)
Google SBA2	221 (6.8%)	378 (0.4%)	0
Phishtank	12 (0.4%)	435 (0.5%)	1
Malware Norm	69 (2.1%)	154 (0.2%)	0
Joewein	240 (7.4%)	652 (0.7%)	11
APWG	56 (1.7%)	569 (0.6%)	0
Spamcop	232 (7.1%)	921 (1.0%)	0
All blacklists	830 (25.6%)	3104 (3.4%)	12
MyPageKeeper classifier	2405 (74.4%)	89389 (96.6%)	

Table 7: Comparison of contribution made by blacklists and classifier to MyPageKeeper’s identification of socware during the four month period of operation.

plication. Once a user visits this page, she can read the application’s description and then click on a link on this page if she decides to install it. However, posts from some surreptitious applications contained shortened URLs that directly take the user to a page where they request the user to grant permissions (e.g., to post on the user’s wall) and install the application. We have found all instances of such applications to be spamming applications. Therefore, if any of the URLs flagged by MyPageKeeper’s classifier is a shortened URL that directly points to the installation page for a Facebook app, we declare that classification correct.

- *Spamming app:* From our dataset, we manually identified several Facebook applications that try to spread on Facebook by promising free money to users and make posts that point to the application page. Once installed by a user, such applications periodically post on the user’s wall (without requesting the user’s authorization for each post) in an attempt to further propagate by attracting that user’s friends; Table 6 shows some such applications that frequently appear in our dataset. Any URLs classified as socware by MyPageKeeper’s classifier that happen to be posted by one of these manually identified spamming apps are deemed correct.
- *Manual analysis:* Finally, over the operation of MyPageKeeper during the four months, we periodically verified a subset of URLs flagged by the classifier. These provide an additional source of validation.

In all, the union of the above techniques validates that 58,388 out of 60,191 posts declared as socware by the MyPageKeeper classifier are indeed so. Therefore, 97% of the socware identified by MyPageKeeper’s classifier are true positives. On the other hand, the 1,803 posts incorrectly classified as socware constitute less than 0.005% of the over 40 million posts in our dataset. Note that, though all of the above techniques could be folded into MyPageKeeper itself to help identify socware, we do not do so because all of these techniques require us to crawl a URL in order to evaluate it; we cannot afford the latency of crawling.

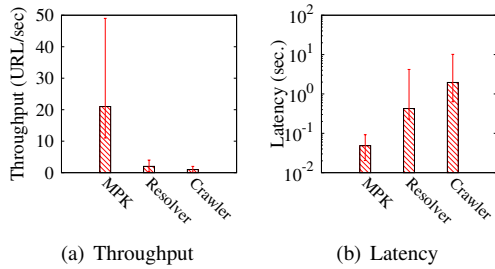


Figure 2: Comparison of MyPageKeeper’s throughput and latency in classifying URLs with a short URL resolver and a crawler-based approach. The height of the box shows the median, with the whiskers representing 5th and 95th percentiles.

4.2 Comparison with blacklists

Though we see that the identification of socware by MyPageKeeper’s classifier is accurate, the next logical question is: what is the classifier’s contribution to MyPageKeeper in comparison with URL blacklists? Table 7 provides a breakdown of the URLs and posts classified as socware by MyPageKeeper during the four month period under consideration. There are two main takeaways from this table. First, we see that the classifier finds 74.4% of socware URLs and 96.6% of socware posts identified by MyPageKeeper. Thus, the classifier accounts for a large majority of socware identified by MyPageKeeper and is thus critical to the system’s operation. Second, there is very little overlap between the URLs flagged by blacklists and those flagged by the classifier. The typically low frequency of occurrence of URLs that match blacklists is another reason that the classifier’s share of identified socware posts is significantly greater than its corresponding share of flagged URLs.

4.3 Efficiency

Beyond accuracy, it is critical that MyPageKeeper’s identification of socware be efficient, so as to minimize the costs that we need to bear in order to keep the delay in identifying socware and alerting users low. The matching of a URL against a whitelist or a local set of blacklists incurs minimal computational overhead. In addition, we find that execution of the classifier also imposes minimal delay per URL verified.

To demonstrate the efficiency of MyPageKeeper, we compare the rate at which it classifies URLs with the classification throughput that two other alternative classes of approaches would be able to sustain. Our first point of comparison is an approach that relies only on locally queryable URL whitelists and blacklists but resolves all shortened URLs into the corresponding complete URL. Our second alternative crawls URLs to evaluate them, e.g., using the content on the page or the IP address of the target website. Figure 2(a) compares the throughput of classifying URLs with the three approaches, using data from

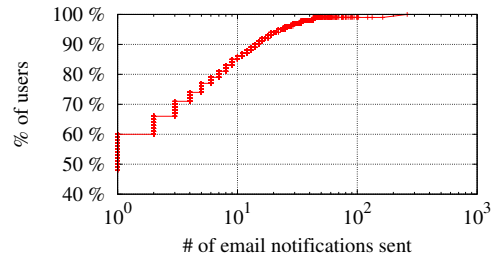


Figure 3: 49% of MyPageKeeper’s 12,456 users were notified of socware at least once in four months of MyPageKeeper’s operation.

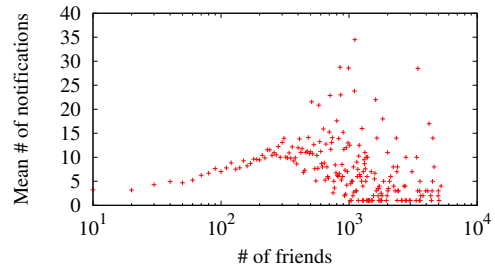


Figure 4: Correlation between vulnerability and social degree of exposed users.

two weeks of MyPageKeeper’s execution. We see that the throughput with MyPageKeeper is almost an order of magnitude greater than the alternatives, with all three approaches using the same set of resources on EC2. As we see in Figure 2(b), MyPageKeeper’s better performance stems from its lower execution latency to check an URL; the median classification latency with MyPageKeeper is 48 ms compared to a median of 426 ms when resolving short URLs and 1.9 seconds when crawling URLs. Thus, we are able to significantly reduce MyPageKeeper’s classification latency, compared to approaches that need to resolve short URLs or crawl target web pages, by keeping all of its computation local.

Furthermore, a crawler-based approach will be significantly more expensive than MyPageKeeper. Thomas et al. [54] found that crawler-based classification of 15 million URLs per day using cloud infrastructure results in an expense of \$800/day. Therefore, we estimate that it would cost approximately \$1.5 million/year to handle Facebook’s workload; 1 million URLs are shared every 20 minutes on Facebook [35]. Since MyPageKeeper’s classification latency is 40 times less than a crawler-based approach, we estimate that the expense incurred with MyPageKeeper would be at least 40 times lower than a system that classifies URLs by crawling them.

5 Analysis of Socware

Thus far we described how MyPageKeeper detects socware efficiently at scale. In this section, we analyze the socware that we have found during MyPageKeeper’s operation to throw light on characteristics of socware on Facebook.

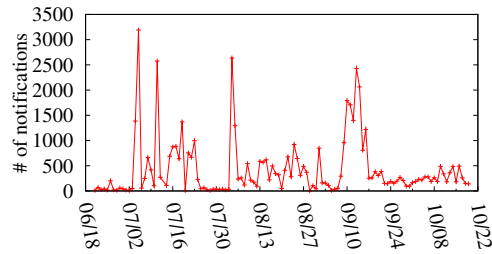


Figure 5: No. of socware notifications per day. On 11th July, 19th Sep, and 3rd Oct, socware was observed in large scale.

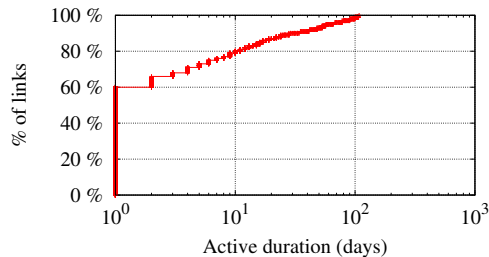


Figure 6: Active-time of socware links. 20% of socware links were observed more than 10 days apart.

5.1 Prevalence of socware

49% of MyPageKeeper’s users were exposed to socware within four months. First, we analyze the prevalence of socware on Facebook. To do so, we define that a user was exposed to a particular socware post if that post appeared in her wall or news feed. As shown in Figure 3, 49% of MyPageKeeper’s users were exposed to at least one socware post during the four month period we consider here. Though this already indicates the wide reach of socware on Facebook, we stress that 49% is only a lower-bound due to a couple of reasons. First, many of MyPageKeeper’s users subscribed to our application at some time in the midst of the four month period and therefore, we miss socware that they were potentially exposed to prior to them subscribing to MyPageKeeper. Second, Facebook itself detects and removes posts that it considers as spam or pointing to malware [36, 38, 52]. All the socware detected by MyPageKeeper is after such filtering by Facebook.

Given that some users are exposed to more socware than others, we analyze if the social degree of a user has any impact on the probability of a user being exposed to socware. Figure 4 shows the number of socware notifications received by MyPageKeeper users as a function of the number of friends they have on Facebook. We bin users with the number of friends within 10 of each other and plot the average number of notifications per bin; we consider here only those users who were subscribed to MyPageKeeper for at least three months. We see that the probability of users being exposed to socware is largely independent of their social degree. This indicates that whether a user is more likely to be exposed to socware is not simply a function of how many friends she has, but

Shortening service	% of socware URLs
bit.ly	21.9%
tinypurl.com	18.8%
goo.gl	5.1%
t.co	3.16%
tiny.cc	1.6%
ow.ly	1.1%
on.fb.me	1.0%
is.gd	0.7%
j.mp	0.4%
0rz.com	0.3%
All shortened URLs	54%

Table 8: Top URL shortening services in our socware dataset.

Domain Name	% of URLs	% of posts
facebook.com	20.7%	26.3%
blogspot.com	6.3%	8.7%
miessass.info	1.9%	3.2%
shurulburul.tk	1.8%	1.2%
tomoday.info	0.8%	0.13%

Table 9: Top two-level domains in our socware dataset.

likely depends on the susceptibility of those friends to becoming victims of scams and helping propagate them.

We also find that socware on Facebook is prevalent over time. Figure 5 shows the number of socware notifications sent per day by MyPageKeeper to its users. We see a consistently large number of notifications going out daily, with noticeable spikes on a few days. On 11th July 2011, a scam that conned users to complete surveys with the pretext of fake free products went viral and posts pointing to the scam appeared 4,056 times on the walls and news feeds of MyPageKeeper’s users. Two other scams, that promised ‘Free Facebook shoes’ and conned users to fill out surveys, also caused MyPageKeeper to send out a large number of notifications on that day. On 19th Sep. 2011, different variants of the ‘Facebook Free T-Shirt’ scam [9] were spreading on Facebook and was spotted 2,040 times by MyPageKeeper. On 3rd Oct. 2011, a video scam was spreading on Facebook and MyPageKeeper observed it in 1,739 posts.

We next analyze the prevalence and impact of socware from the perspective of individual socware links. For each link, we define its “active-time” as the difference between the first and last times of its occurrence in our dataset. Figure 6 shows that we did not see 60% of socware links beyond one day. Subsequent posts containing these links may have been filtered by Facebook once it recognized their spammy or malicious nature, or our dataset may miss those posts due to MyPageKeeper’s limited view into Facebook’s 850 million users. Further, we do not attempt any clustering of links into campaigns here. However, even with these caveats, 20% of socware links were seen in multiple posts separated by at least 10 days, suggesting that a significant fraction of socware eludes Facebook’s detection mechanisms and lasts on Facebook for significant durations.

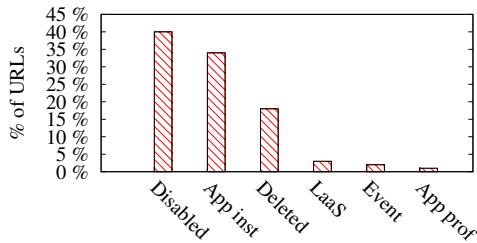


Figure 7: Breakdown of socware links, when crawled in Nov. 2011, that originally point to web pages in the `facebook.com` domain.

5.2 Domain name characteristics

20% of socware links are hosted inside Facebook.

In the next section of our analysis, we focus on the domain-level characteristics of socware links. First, Table 8 shows the top ten URL shortening services used in socware links observed by MyPageKeeper. In all, shortened URLs account for 54% of socware links in our dataset. Our design of MyPageKeeper’s classifier to rely solely on social context, and to not resolve short URLs, hence makes a significant difference (as previously seen in the comparison of classification latency).

Further, we find it surprising that a large fraction of socware links (46%) are not shortened, given that shortening of URLs enables spammers to obfuscate them. On further investigation, we find that many Facebook scams such as ‘free iPhone’ and ‘free NFL jersey’ use domain names that clearly state the message of the scam, e.g., `http://iphonefree5.com/` and `http://nfljerseymfree.com/`. These URLs are more likely to elicit higher click-through rates compared to shortened URLs. On the other hand, most of the shortened URLs were used by malicious or spam applications (e.g., ‘The App’, ‘Profile Stalker’) that generate shortened URLs pointing to their application’s installation page. We find that 89% of shortened URLs in our dataset of socware links were posted by Facebook applications.

Next, based on our crawl of the socware links in our dataset, we inspect the top two-level domains found on the landing pages pointed to by these links. First, as shown in Table 9, we find that a large fraction of socware (over 20% of URLs and 26% of posts) is hosted on Facebook itself. Second, a sizeable fraction of socware uses sites such as `blogspot.com` and `wordpress.com` that enable the spammers to easily create a large number of URLs without going through the hassle of registering new domains. Further, all of these domains are of good reputation and are unlikely to be flagged by traditional website blacklists.

5.3 Analysis of socware hosted in Facebook

Hackers use numerous channels in Facebook to spread socware. Given the large fraction of socware

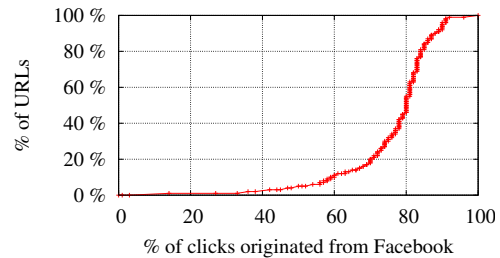


Figure 8: For most socware links shortened with `bit.ly` or `goo.gl`, a large majority of the clicks came from Facebook.

hosted on Facebook itself, we next analyze this subset of socware. First, in early November 2011, we crawled every socware link in our dataset that had pointed to a landing page in the `facebook.com` domain at the time when MyPageKeeper had initially classified that link as socware. Figure 7 presents a breakdown of the results of this crawl. If Facebook disables a URL, it redirects us to `facebook.com/home.php`. Similarly, if crawling a URL points us to `facebook.com/4oh4`, it implies that Facebook has deleted the content at that URL. Therefore, as seen in Figure 7, a large fraction of socware links that were originally pointing to Facebook have now been deactivated. However, we also see that a significant fraction of these links—over 40%—were still live. Further, the figure shows that spammers use several different channels, such as applications, events, and pages to propagate their scams on Facebook. In the figure, ‘App inst’ and ‘App prof’ refer to the installation and profile pages of Facebook applications, and ‘LaaS’ refers to campaigns intended to increase the number of Likes on a Facebook page (described in detail in Section 6).

In our dataset, we see 257 distinct socware links shortened with the `bit.ly` and `goo.gl` URL shorteners that point to landing pages in the `facebook.com` domain. Using the APIs [5, 16] offered by these URL shortening services, we computed the number of clicks recorded for these 257 links in two cases—1) where the Referrer was Facebook, and 2) where the Referrer was any other domain. Figure 8 shows that Facebook is the dominant platform from which most of these links received most of their clicks; 80% of links received over 70% of their clicks from Facebook. This seems to indicate that most socware hosted on Facebook is propagated solely on Facebook and tailored for that platform.

5.4 Comparison of socware to email spam

Socware keywords exhibit little (10%) overlap with spam email keywords. As we saw earlier in Section 4, spam keyword score is a key feature in MyPageKeeper’s classifier. Therefore, in the final section of our analysis, we investigate the overlap in ‘spam keywords’ that we observe in socware on Facebook with those seen in another medium targeted by spammers, specifically email.

Socware word	Likelihood ratio	Spam email word	Likelihood ratio
free	12.1	money	11.5
< 3	∞	price	26.6
iphone	∞	free	0.08
awesome	31.3	account	9.6
win	24.3	stock	9.7
wow	90.8	address	5.2
hurry	36.8	bank	56.4
omg	332.3	pills	∞
amazing	4.9	viagra	∞
deal	1.9	watch	1.9

Table 10: Top keywords from socware posts and spam emails.

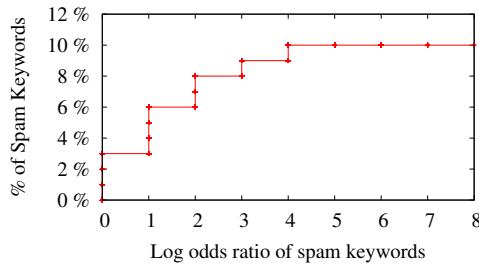


Figure 9: Overlap of keywords between email and Facebook.

We investigate whether spammers use similar keywords on Facebook as they use in email spam.

To perform this analysis, we collected over 17,000 spam emails from [50]. For Facebook spam, we use 92,493 socware posts collected by MyPageKeeper. We transform posts in either dataset to a bag of words with their frequency of occurrence. Similar to [54], we then compute the log odds ratio for each keyword to determine its overlap in Facebook socware and spam email. Here, the log odds ratio for a keyword is defined by $ratio = |\log(p_1 q_2 / p_2 q_1)|$ where p_i is the likelihood of that keyword appearing in set i and $q_i = 1 - p_i$. A value of 0 for the log odds ratio indicates that the keyword is equally likely to appear in both datasets, whereas an infinite ratio indicates that the keyword appears in only one of the datasets. In Figure 9 (infinite values are omitted), we see only a 10% overlap in spam keywords between email and Facebook. This indicates that Facebook spam significantly differs from traditional email spam.

Further, Table 10 shows the likelihood ratio (defined earlier in Section 3.3) for the top keywords in either dataset. The higher the likelihood ratio of a socware keyword, the stronger the bias of the keyword appearing more in Facebook socware than in email spam; an infinite ratio implies the keyword exclusively appears in Facebook socware. The word ‘omg’ is 332 times more likely to be used in Facebook socware than in email spam. On the other hand, words such as ‘pills’ and ‘viagra’ are restricted solely to email spam.

6 Like-as-a-Service

Facebook has now become the premier online destination on the Internet. Over 900 million users, half of whom visit the site daily, spend over 4 hours on the site every

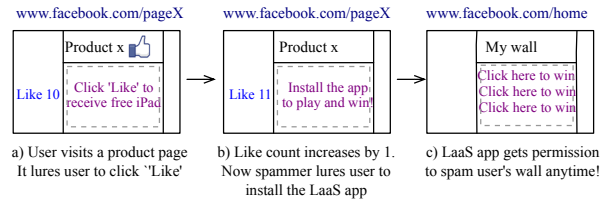


Figure 10: A representation of how a Like-as-a-service Facebook application collects Likes for its client’s page and gains access to the user’s wall for spamming. Dotted region of the page is controlled by the spammer.

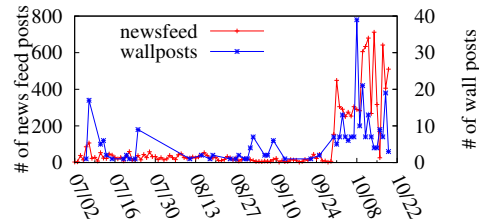


Figure 11: Timeline of posts made by the Games LaaS Facebook application seen on users’ walls and news feeds.

month [10]. To leverage user activity on Facebook, an increasingly large number of businesses have Facebook pages associated with their products. However, attracting users to their page is a challenge for any business. One way of doing so is to make users who visit a Facebook page click the ‘Like’ button on the page. A large number of Likes has two significant implications. First, the number of Likes associated with a page has begun to represent the reputation associated with a page, e.g., a higher number of Likes improves the page’s rank in Bing [3]. Second, a link to the product page appears in the news feed of the friends of the user who clicked Like on the page, thus enabling the link to the page to spread on Facebook.

Based on our view of Facebook socware through the MyPageKeeper lens, we see an emerging Like-as-a-Service³ market to help businesses attract users to their pages. We identify several Facebook apps (e.g., ‘Games’ [15], ‘FanOffer’ [13], and ‘Latest Promotions’ [21]) which are hired by the owners of Facebook pages to help increase the number of Likes on their pages. These applications, which offer Likes as a service, presumably get paid on a ‘Pay-per-Like’ model by the owners of Facebook pages that make use of their services.

Figure 10 shows how a Like-as-a-Service (LaaS) application typically works. First, a customer of the LaaS application integrates the application into their Facebook page. When users visit the page, the LaaS application entices the user to click Like on page. Typically, the re-

³ Note that ‘Like-as-a-Service’ differs from ‘Likejacking’ [22], where users are tricked into clicking the Like button without them realizing they are doing so, e.g., by enticing the user to click on a Flash video, within which the Like button is hidden.

Page Name	Application Message	No. of Likes
Raging Bid	Just got a better score on Raging Bid's Bouncing Balls contest and I am now in 12297th place. I am getting closer to winning a Sony Bravia 3D HDTV. Who thinks they can beat my score? Click here to try: URL	168,815
www.WalkerToyota.com	DAILY CONTEST UPDATE: I am currently in 7573rd place in Walker Toyota's Tetris contest. There is still plenty of time to try and win a 16GB iPad2. Who thinks they can get a better score than me? Click here to try: URL	136,212
Chip Banks Chevrolet Buick	DAILY CONTEST UPDATE: I am currently in 310th place in Chip Banks Chevrolet Buick's Gem Swap II contest. There is still plenty of time to try and win a 16GB iPad2. Who thinks they can get a better score than me? Click here to try: URL	2,190
Casey Jamerson	DAILY CONTEST UPDATE: I am currently in 6234th place in Casey Jamerson Music's Gem Swap II contest. There is still plenty of time to try and win a 16GB iPad2. Who thinks they can get a better score than me? Click here to try: URL	47,496
Tara Gray	DAILY CONTEST UPDATE: I am currently in 10213th place in Tara Gray's Gem Swap II contest. There is still plenty of time to try and win a Burma Ruby Ring. Who thinks they can get a better score than me? Click here to try: URL	231,035

Table 11: Five example Facebook pages integrated with the Games LaaS application to spam users' walls for propagation.

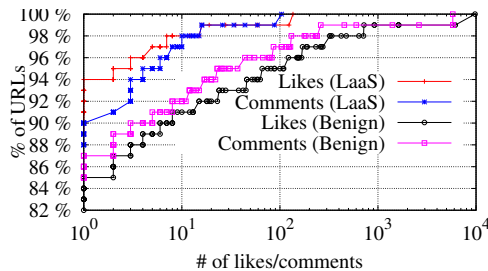


Figure 12: # of Likes and comments associated with URLs posted by the Games Facebook app.

ward promised to the user in return for his Like is that the user can play some games on the page or have a chance of winning free products. However, once the user clicks Like on the page to access the promised reward, the LaaS application then demands that the user add the application to his profile in order to proceed further. In the process of getting the user to add the LaaS application, the application requests the user to grant permission for it to post on the user's wall. Once the application obtains such permissions, it periodically spams the user's wall with posts that contain links to the Facebook page of the customer who enrolled the LaaS application for its services. These posts will appear in the news feeds of the unsuspecting user's friends, who in turn may visit the Facebook page and go through the same cycle again. The LaaS application thus enables the Facebook pages of its customers to accumulate Likes and increase their reputation, even though users are clicking Like on these pages with the promise of false rewards rather than because they like the products advertised on the page.

Here, we analyze the activity of one such LaaS application—Games [15]. Figure 11 shows that posts made by this application appear regularly in the walls and news feeds of MyPageKeeper's users. Even with our small sample of roughly 12K users from Facebook's total population of over 850 million users, we see that 40 users have posts made by Games on their walls, which implies that these users have installed the application and granted it permission to make posts on their wall at any time. We also see that the number of users who installed Games

significantly rose around mid-September 2011. Further, from the news feeds of MyPageKeeper users, we see that Games posted links to as many as 700 Facebook pages on a single day; each link points to the Facebook page of a different customer of this LaaS application. Table 11 shows the posts made by Games for some of its customers, the variation in text messages across these posts, and the large number of Likes garnered by the Facebook pages of these customers.

We next analyze the Likes and comments received by 721 URLs posted by the Games app. As shown in Figure 12, we see that over 95% of these URLs have less than 100 Likes and less than 100 comments; this fraction is significantly lesser on a dataset of randomly chosen 721 URLs from benign posts. However, over 20% of the URLs posted by the Games app do receive Likes and comments, thus enabling them to propagate on Facebook. Real users may be unknowingly helping to spreading spam in these cases; such users have been previously referred to as creepers [52].

7 Discussion

Client-based solution. An alternative to MyPageKeeper's server-side detection of socware would be to identify socware on client machines. In such an approach, a client-side tool can classify a post at the instant when the user accesses the post. However, we choose not to use such an approach for multiple reasons. First, a server-side solution is more amenable to adoption; it is easier to convince users to add an app to their Facebook profile than to convince them to download and install an application or browser extension on their machines. Second, users can access Facebook from a range of browsers and even from different device types (e.g., mobile phones). Developing and maintaining client-side tools for all of these platforms is onerous. Finally, and most importantly, many of the features used by our socware classifier (e.g., message similarity score) fundamentally depend on aggregating information across users. Therefore, a view of Facebook from the perspective of a

single client may be insufficient to identify socware accurately.

Estimating false negatives. While we evaluated the accuracy of socware identified by MyPageKeeper by cross-validating with other techniques, evaluating the accuracy of MyPageKeeper's classifier in cases where it declares a URL safe is much harder. Not only do we lack ground truth, but since the highly common case is that a Facebook post is benign, manual verification of a randomly chosen subset of the classifier's negative outputs is insufficient.

We therefore evaluate whether MyPageKeeper's classifier misses any socware by using data from user-reported samples of socware. As shown in Table 3, 533 distinct MyPageKeeper users have submitted 679 such reports and we have received 333 unique URLs across these reports. Based on manual verification, we find that 296 of these 333 URLs indeed point to spam or malware. The remaining 37 URLs point to sites like `surveymonkey.com` (fill out surveys) and `clixsense.com` (get paid to view advertisements), which though abused by spammers have legitimate uses as well. We suspect that our users did come across socware, but reported the URL of the landing page, rather than the URL that they originally found in a socware post.

Of the 296 instances of true socware reported by users, MyPageKeeper's classifier flagged all but 17 of them, independently of users reporting them to us. This translates into a false negative rate of 5% for the classifier. However, 16 of these 17 URLs had been found to match against one of the URL blacklists used by MyPageKeeper. Thus, the false negative rate for the whole MyPageKeeper system, which combines blacklists and the classifier to detect socware, is 0.3%.

Arms race with spammers. Though our current techniques seem to suffice to accurately identify socware on Facebook, we speculate here on how spammers may evolve socware, given the knowledge of how MyPageKeeper works. One option for spammers to evade MyPageKeeper is to use different shortened URLs for a single malicious landing URL. In such cases, MyPageKeeper would consider every posted shortened URL separately even though they are all part of the same campaign. Thus, if any of these shortened URLs does not appear on the walls/news feeds of several users, MyPageKeeper may fail to flag it. Another option for socware to evade MyPageKeeper is for spammers to slow down its rate of propagation; as we found in Section 4.2, MyPageKeeper sometimes misses socware which is observed only a few times in our dataset. However, slowing down a socware epidemic makes it likely that it will be flagged by other techniques, such as URL blacklists. Moreover, spammers may often be unable to control how fast a socware epidemic spreads. In the case where an epidemic

spreads by luring users into installing a Facebook app, the spammer can control how often the app posts spam on the user's wall. However, in cases where users are asked to 'Like' or 'Share' a post to access a fake reward, the socware is self-propagating and its viral spread cannot be controlled by spammers.

Another option is for spammers to change the keywords that they use in socware posts, thus affecting the spam keyword score used by MyPageKeeper's classifier. Though spammers are constrained in their choice of keywords by the need to attract users, some of the keywords may evolve over time as popular colloquial expressions (e.g., 'OMG') change. To evaluate MyPageKeeper's ability to cope with such change, we identified the top keywords (those with high likelihood ratio compared to benign posts among frequently occurring keywords) distinctive to user-reported socware posts. We find that the spam keywords that we use in MyPageKeeper's classifier (identified from manually identified samples of socware) match those computed here. Though this captures data only across four months, MyPageKeeper can similarly recompute the set of spam keywords over time.

8 Related Work

Motivated by the increasing presence of spam and malware on OSNs, there have been several recent related efforts. Here, we contrast our work with these prior efforts.

Studies of spam on OSNs. Gao et al. [43] analyzed posts on the walls of 3.5 million Facebook users and showed that 10% of links posted on Facebook walls are spam, with a large majority pointing to phishing sites. They also presented techniques to identify compromised accounts and spam campaigns. In a similar study on Twitter, Grier et al. [44] showed that at least 8% of links posted on Twitter are spam while 86% of the involved accounts are compromised. In contrast to this study, Thomas et al. [55] show that the majority of suspended accounts in Twitter are created by spammers as opposed to compromised users. All of these efforts however focus on post-mortem analysis of historical OSN data and are not applicable to MyPageKeeper's goal of identifying socware soon after it appears on a user's wall or news feed.

Detecting spam accounts. Benevenuto et al. [39] and Yang et al. [57] developed techniques to identify accounts of spammers on Twitter. Others have proposed a honeypot based approach [53, 47] to detect spam accounts on OSNs. Yardi et al. [58] analyzed behavioral patterns among spam accounts in Twitter. Instead of focusing on accounts created by spammers, MyPageKeeper enables socware detection on the walls and news feeds of legitimate Facebook users.

Real-time spam detection in OSNs. Thomas et al. [54] developed Monarch, a real-time system that

crawls URLs submitted from services such as Twitter to determine whether a URL directs to spam. Monarch relies on the network and domain level properties of URLs as well as the content of the web pages obtained when URLs are crawled. Interestingly, Monarch's classification accuracy is shown to be independent of the social context on Twitter. MyPageKeeper distinguishes itself from Monarch in several ways—1) we study socware on Facebook, which we see significantly differs in its characteristics from traditional spam messages, 2) to make MyPageKeeper efficient, our socware classifier operates without crawling of links found in posts, and 3) we find that the use of social context based features is crucial to efficient detection of socware. In another study, Gao et al. [42] perform online spam filtering on OSNs using incremental clustering. Their technique however relies on having the whole social graph as input, and so, is usable only by the OSN provider. MyPageKeeper instead relies only on the view of the OSN as seen by MyPageKeeper's users. Lee et al. [48] built Warningbird, a system to detect suspicious URLs in Twitter; their system however relies on following the HTTP redirection chains of URLs, thus making their approach less efficient than MyPageKeeper.

Wang et al. [56] propose a unified spam detection framework that works across all OSNs, but they do not have an implementation of such a system in practice. Stein et al. [52] describe Facebook's Immune System (FIS), a scalable real-time adversarial learning system deployed in Facebook to protect users from malicious activities. However, Stein et al. provide only a high-level overview about threats to the Facebook graph and do not provide any analysis of the system. Similarly, other Facebook applications [6, 25, 4] that defend users against spam and malware are proprietary with no details available about how they work. Abu-Nimeh et al. [37] analyze the URLs flagged by one of these applications, Defenseio, but they do not discuss Defenseio's classification techniques and their analysis is restricted to that of the hosting infrastructure (country and ASN) underlying Facebook spam. To the best of our knowledge, we are the first to provide classification of socware on Facebook that relies solely on social context based features, thus enabling MyPageKeeper to efficiently detect socware at scale.

Social context based email spam. Jagatic et al. [45] discuss how email phishing attacks can be launched by using publicly available personal information (e.g., birthday) from social networks, and Brown et al. [40] analyzed such email spam seen in practice. However, due to revisions in Facebook's privacy policy over the last couple of years, only a user's friends have access to such information from the user's profile, thus making such email spam no longer possible. Further, MyPageKeeper focuses on spam propagated on Facebook rather than via email.

9 Conclusions

Facebook is becoming the new epicenter of the web, and we showed that hackers are adapting to this change by designing new types of malware suited to this platform, which we call socware. In this paper, we presented the design and implementation of MyPageKeeper, a Facebook application that can accurately and efficiently identify socware at scale. Using data from over 12K Facebook users, we found that the reach of socware is widespread and that a significant fraction of socware is hosted on Facebook itself. We also showed that existing defenses, such as URL blacklists, are ill-suited for identifying socware, and that socware significantly differs from email spam. Finally, we identified a new trend in aggressive marketing of Facebook pages using "Like-as-a-Service" applications that spam users to make money based on a "Pay-per-Like" model.

References

- [1] Anti-phishing working group. <http://www.antiphishing.org/>.
- [2] Application authentication flow using oauth 2.0. <http://developers.facebook.com/docs/authentication/>.
- [3] Bing gets friendlier with Facebook. <http://www.technologyreview.com/web/37585/>.
- [4] Bitdefender Safego. <http://www.facebook.com/bitdefender.safego>.
- [5] bit.ly API. <http://code.google.com/p/bitly-api/wiki/ApiDocumentation>.
- [6] Defenseio Social Web Security. <http://www.facebook.com/apps/application.php?id=177000755670>.
- [7] Escrow-fraud. <http://escrow-fraud.com/>.
- [8] Experts: Facebook crime is on the rise. <http://www.zdnet.com/blog/facebook/experts-facebook-crime-is-on-the-rise/2632>.
- [9] Facebook birthday T-shirt scam steals secret mobile email addresses. <http://bit.ly/Kvax0t>.
- [10] Facebook is the web's ultimate timesink. <http://mashable.com/2010/02/16/facebook-nielsen-stats/>.
- [11] Facebook Phishing Scam Costs Victims Thousands of Dollars. <http://www.hyphenet.com/blog/2011/10/04/facebook-phishing-scam-costs-victims-thousands-of-dollars/>.
- [12] Facebook scam involves money transfers to the Philippines. <http://profitscam.com/facebook-scam-involves-money-transfers-to-the-philippines-post/>.
- [13] Fan Offer. <https://www.facebook.com/apps/application.php?id=107611949261673>.
- [14] FBML- Facebook Markup Language. <https://developers.facebook.com/docs/reference/fbml/>.

- [15] Games. <https://www.facebook.com/apps/application.php?id=121297667915814>.
- [16] goo.gl API. http://code.google.com/apis/urlshortener/v1/getting_started.html.
- [17] Google Safe Browsing API. <http://code.google.com/apis/safebrowsing/>.
- [18] Hackers selling \$25 toolkit to create malicious Facebook apps. <http://zd.net/M2WNe1>.
- [19] How to spot a Facebook Survey Scam. <http://facecrooks.com/Safety-Center/Scam-Watch/How-to-spot-a-Facebook-Survey-Scam.html>.
- [20] Joewein: Fighting spam and scams on the Internet. <http://www.joewein.net/>.
- [21] Latest Promotions. <https://www.facebook.com/apps/application.php?id=174789949246851>.
- [22] Likejacking takes off on Facebook. http://www.readwriteweb.com/archives/likejacking_takes_off_on_facebook.php.
- [23] MalwarePatrol- Malware is everywhere! . <http://www.malware.com.br/>.
- [24] MyPageKeeper. <https://www.facebook.com/apps/application.php?id=167087893342260>.
- [25] Norton Safe Web. <http://www.facebook.com/apps/application.php?id=310877173418>.
- [26] Phishtank. <http://www.phishtank.com/>.
- [27] Rihanna video scam. "http://www.virteacon.com/2011/11/sick-i-just-hate-rihanna-after-watching.html".
- [28] Spamcop. <http://www.spamcop.net/>.
- [29] Spamhaus. <http://www.spamhaus.org/sbl/index.lasso>.
- [30] Steve Jobs death scams are just the greedy exploiting the gullible. <http://bit.ly/M67Zme>.
- [31] SURBL. <http://www.surbl.org/>.
- [32] SVM Tutorials. <http://svms.org/tutorials/>.
- [33] URIBL. <http://www.uribl.com/>.
- [34] Web-of-trust. <http://www.mywot.com/>.
- [35] What 20 Minutes On Facebook Looks Like. <http://tcn.ch/KytqzB>.
- [36] Facebook becomes partner with Web of Trust (WOT). <https://www.facebook.com/notes/facebook-security/keeping-you-safe-from-scams-and-spam/10150174826745766>, May 2011.
- [37] S. Abu-Nimeh, T. M. Chen, and O. Alzubi. Malicious and spam posts in online social networks. In *IEEE Computer Society*, 2011.
- [38] F. becomes partner with WebSense. <http://www.thetechherald.com/article.php/201139/7675/Facebook-implements-malicious-link-scanning-service>, Oct 2011.
- [39] F. Benevenuto, G. Magno, T. Rodrigues, and V. Almeida. Detecting spammers on Twitter. In *CEAS*, 2010.
- [40] G. Brown, T. Howe, M. Ihbe, A. Prakash, and K. Borders. Social networks and context-aware spam. In *ACM CSCW*, 2008.
- [41] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2, 2011.
- [42] H. Gao, Y. Chen, K. Lee, D. Palsetia, and A. Choudhary. Towards online spam filtering in social networks. 2012.
- [43] H. Gao, J. Hu, C. Wilson, Z. Li, Y. Chen, and B. Y. Zhao. Detecting and characterizing social spam campaigns. In *IMC*, 2010.
- [44] C. Grier, K. Thomas, V. Paxson, and M. Zhang. @spam: The underground on 140 characters or less. In *CCS*, 2010.
- [45] T. N. Jagatic, N. A. Johnson, M. Jakobsson, and F. Menczer. Social phishing. *Commun. ACM*, 2007.
- [46] A. Le, A. Markopoulou, and M. Faloutsos. Phishdef: Url names say it all. In *Infocom*, 2010.
- [47] K. Lee, J. Caverlee, and S. Webb. Uncovering social spammers: social honeypots + machine learning. In *SIGIR*, 2010.
- [48] S. Lee and J. Kim. Warningbird: Detecting suspicious urls in twitter stream. 2012.
- [49] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker. Beyond blacklists: learning to detect malicious web sites from suspicious urls. In *KDD*, 2009.
- [50] V. Metsis, I. Androutsopoulos, and G. Paliouras. Spam filtering with naive bayes – which naive bayes? In *CEAS*, 2006.
- [51] Z. Qian, Z. M. Mao, Y. Xie, and F. Yu. On network-level clusters for spam detection. In *NDSS*, 2010.
- [52] T. Stein, E. Chen, and K. Mangla. Facebook immune system. In *SNS*, 2011.
- [53] G. Stringhini, C. Kruegel, and G. Vigna. Detecting spammers on social networks. In *ACSAC*, 2010.
- [54] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song. Design and Evaluation of a Real-Time URL Spam Filtering Service. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.
- [55] K. Thomas, C. Grier, V. Paxson, and D. Song. Suspended accounts in retrospect: An analysis of twitter spam. In *IMC*, 2011.
- [56] D. Wang, D. Irani, and C. Pu. A social-spam detection framework. In *CEAS*, 2011.
- [57] C. Yang, R. Harkreader, and G. Gu. Die free or live hard? empirical evaluation and new design for fighting evolving twitter spammers. In *RAID*, 2011.
- [58] S. Yardi, D. Romero, G. Schoenebeck, et al. Detecting spam in a twitter network. *First Monday*, 2009.