;login:

SUMMER 2017 VOL. 42, NO. 2

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

UPCOMING EVENTS

USENIX ATC '17: 2017 USENIX Annual Technical Conference
July 12–14, 2017, Santa Clara, CA, USA
www.usenix.org/atc17

Co-located with USENIX ATC '17

SOUPS 2017: Thirteenth Symposium on Usable Privacy and Security
July 12–14, 2017
www.usenix.org/soups2017

HotCloud '17: 9th USENIX Workshop on Hot Topics in Cloud Computing
July 10–11, 2017
www.usenix.org/hotcloud17

HotStorage '17: 9th USENIX Workshop on Hot Topics in Storage and File Systems
July 10–11, 2017
www.usenix.org/hotstorage17

USENIX Security '17: 26th USENIX Security Symposium
August 16–18, 2017, Vancouver, BC, Canada
www.usenix.org/sec17

Co-located with USENIX Security '17

WOOT '17: 11th USENIX Workshop on Offensive Technologies
August 14–15, 2017
www.usenix.org/woot17

CSET '17: 10th USENIX Workshop on Cyber Security Experimentation and Test
August 14, 2017
www.usenix.org/cset17

FOCI '17: 7th USENIX Workshop on Free and Open Communications on the Internet
August 14, 2017
www.usenix.org/foci17

ASE '17: 2017 USENIX Workshop on Advances in Security Education
August 15, 2017
www.usenix.org/ase17

HotSec '17: 2017 USENIX Summit on Hot Topics in Security
August 15, 2017
Submissions due June 12, 2017
www.usenix.org/hotsec17

SREcon17 Europe/Middle East/Africa
August 30–September 1, 2017, Dublin, Ireland
www.usenix.org/srecon17europe

LISA17
October 29–November 3, 2017, San Francisco, CA, USA
www.usenix.org/lisa17

Enigma 2018
January 16-18, 2018, Santa Clara, CA
Submissions due August 23, 2017
www.usenix.org/enigma2018

FAST '18: 16th USENIX Conference on File and Storage Technologies
February 12–15, 2018, Oakland, CA, USA
Submissions due September 28, 2017
www.usenix.org/fast18

SREcon18 Americas
March 27–29, 2018, Santa Clara, CA

NSDI '18: 15th USENIX Symposium on Networked Systems Design and Implementation
April 9-11, 2018, Renton, WA
Paper titles and abstracts due September 18, 2017
www.usenix.org/nsdi18

USENIX Open Access Policy
USENIX is the first computing association to offer free and open access to all of our conferences proceedings and videos. We stand by our mission to foster excellence and innovation while supporting research with a practical bias. Your membership fees play a major role in making this endeavor successful.

Please help us support open access. Renew your USENIX membership and ask your colleagues to join or renew today!

www.usenix.org/membership

www.usenix.org/facebook www.usenix.org/gplus www.usenix.org/linkedin
twitter.com/usenix www.usenix.org/youtube

;login:

SUMMER 2017 VOL. 42, NO. 2

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

EDITOR
Rik Farrow
rik@usenix.org

MANAGING EDITOR
Michele Nelson
michele@usenix.org

COPY EDITORS
Steve Gilmartin
Amber Ankerholz

PRODUCTION
Arnold Gatilao
Jasmine Murcia

TYPESETTER
Star Type
startype@comcast.net

USENIX ASSOCIATION
2560 Ninth Street, Suite 215
Berkeley, California 94710
Phone: (510) 528-8649
FAX: (510) 548-5738

www.usenix.org

;login: is the official magazine of the USENIX Association. *;login:* (ISSN 1044-6397) is published quarterly by the USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710.

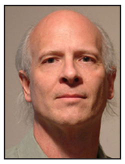
$90 of each member's annual dues is for a subscription to *;login:*. Subscriptions for nonmembers are $90 per year. Periodicals postage paid at Berkeley, CA, and additional mailing offices.

POSTMASTER: Send address changes to *;login:*, USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710.

EDITORIAL

Musings

RIK FARROW



Rik is the editor of *;login:*.
rik@usenix.org

We've got a jam-packed issue this time, so I thought I'd cut right to the chase. Instead of musing, I will just tell you why I picked a particular author, set of authors, or topic for your edification. The reason, in some cases, is to attempt to disabuse you of long-held beliefs.

We begin with Conway et al., who researched the issue of file system fragmentation in Linux [1]. The researchers used a Git workload and showed that all of the popular Linux file systems suffer performance degradation, even on SSDs. After just 100 pulls, hard drive performance was halved, while it took a bit more activity, 800 pulls, to reduce SSD performance by 25%. The authors do have an agenda: their own file system design, BetrFS, doesn't manifest this problem and is faster than other Linux file systems in many cases.

Disk drive manufacturers have been hiding information from us. For many years (decades?), they have converted the logical block address into the physical location of their firmware's choice, so file system designs that attempt to prevent fragmentation really don't have much of a chance. But there's more: hard drives with capacities greater than two terabytes use device managed shingled magnetic recording (SMR). The Aghayev et al. article describes changes they made to ext4 that improve performance not just on SMR drives, but on any hard drive.

Ganesan and company delve into another of our popular myths: that having redundant copies makes our distributed data secure. They researched a number of distributed file systems and databases, induced read or write errors, and discovered some really terrible things. That is, I think it's bad when having multiple copies means that the bad copy gets used to overwrite the good copy, or when a failed read crashes the system. You might want to read this even if you don't use any of the eight distributed systems they tested.

Shvachko and Chen take another look at HDFS. The single-point-of-failure NameNode has long been an issue, and their Giraffa system replaces the NameNode with a distributed naming and block management system. HopsFS, from the Niazi et al. paper at FAST '17 [2], also confronts this issue, although using a different approach.

Programming

Andy Rudoff, who wrote about persistent memory for *;login:* way back in 2013, has written about the PMEM libraries currently available for use with Linux and Windows systems. These libraries focus on using PMEM as memory-mapped files (`mmap()`), but Rudoff also tells us about some other useful libraries and explains how best to use these new devices. Oh, and during Rudoff's FAST '17 tutorial on this topic, he kept waving around an Intel-Micron 3D XPoint device. I actually held this device, and can tell you that **it's real**. PMEM will change the way many systems work in profound ways.

Graeme Jenkinson has written a great article explaining Rust, a programming language with a focus on type safety. Will Rust save the world from buggy code? Probably not, as most people are addicted to whatever they currently use. But Rust is still really worth looking at.

I interviewed Eric Allman, the author of both `syslog` and `sendmail`. Eric has traveled the open source road, a journey more often painful than rewarding for him.

Security

Peck et al. have written what may be the final article in the series on BeyondCorp. BeyondCorp has been a journey away from traditional, trusted, internal networks and into a Zero Trust network design [3]. This article is about the paths taken, ones that couldn't have succeeded without the long process of gaining the trust of the users of Google's networks, learning what could easily be migrated, and how to migrate the more unusual services, over several years.

Hunt et al. have written about the Ryoan sandbox, a system designed to run within Intel SGX enclaves on a distributed system. Their model provides assurance that the expected software is running in the sandbox, that the data sent through the sandbox remains private, and that the sandbox doesn't leak much information through covert channels. You can also learn a lot about how SGX enclaves work by reading their article or their OSDI '16 paper [4].

Kuppaswamy, DeLong, and Kappos challenge people to find flaws in their design, *Uptane,* for providing secure firmware updates for automobiles. Cars are loaded with computers, and many new cars are also network-connected, so having a secure method for installing updates that works both within cars and for car manufacturers is more important than ever.

Radia Perlman reprises her talk at LISA16 about Bitcoin. Radia compares the design and capabilities of Bitcoin to other systems, past and present. Bitcoin design has attracted lots of attention and investors, but is it really any better than other cryptographic systems?

Jos Wetzels spoke at Enigma 2017 about embedded system security. Wetzels researches IoT security issues, and in this article he describes some of the issues facing both researchers and developers of software for embedded systems. In short, things don't look promising, but policy and regulation could set a reasonable baseline for the IoT, just as RoHS [5] already restricts the use of certain hazardous substances in electrical and electronic equipment.

Columns

Dave Beazley explores the new `pathlib` module that appears in Python 3.4 and later. Dave had written about `pathlib` several years ago, and he demonstrates some of the things you can do with that module, as well as things you can't do. Then Dave explains both how `pathlib` improves pathname manipulation, but also problems that arise with incompatibilities between `pathlib` objects and other functions that accept pathnames.

David Blank-Edelman explores Perl modules from the air. David takes us on a tour of some of the modules that come with a stock install of Perl, a very different approach to his usual Perl examples.

Dave Josephsen gets excited about compression. Dave tells us about Gorilla, a time series database that has been open sourced by Facebook and is designed to keep the most recent data in memory. In particular, Dave explains some of the tricks used to compress datestamps in time series.

We have a new columnist this issue. Jeanne Schock, who has worked as a system administrator and now focuses on change, incident, and problem management, has written about root causes and their relations to problems. Seems obvious, right? Well, read on, because it's not that obvious.

Dan Geer and Jon Callas have written about the impact of revealing a nation-state's exploit toolkit [6]. You'll have to read their column, as it's an interesting exercise in game theory.

Robert Ferrell considers how modern backup systems *should* work, then takes pokes at software subscriptions and advertising that targets your dreams. You might think you know what that means, having watched ads with people driving fancy cars sitting next to the mate of their dreams. But that's not what Robert means.

Sometimes it seems to me that things are changing much too quickly to keep up with. Then I notice that Tim Feldman wrote about SMR drives in 2013 [7], just as Andy Rudoff was writing about PMEM the same year. Four years later, and we are just now seeing the effects of the concepts discussed back then, and while there are millions of SMR drives in use, there aren't any 3D XPoint cards available on the open market. As William Gibson quipped in 1993, "The future is already here—it's just not evenly distributed."

References
[1] A. Conway, A. Bakshi, Y. Jiao, Y. Zhan, R. Johnson, B. C. Kuszmaul, and M. Farach-Colton, "File Systems Fated for Senescence? Nonsense, Says Science!" in *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17):* https://www.usenix.org/system/files/conference/fast17/fast17-conway.pdf.

[2] S. Niazi, M. Ismail, S. Haridi, J. Dowling, S. Grohsschmiedt, and M. Ronström, "HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17):* https://www.usenix.org/system/files/conference/fast17/fast17-niazi.pdf.

[3] D. Barth and E. Gilman, "Zero Trust Networks: Building Trusted Systems in Untrusted Networks," SREcon17 Americas: https://www.usenix.org/conference/srecon17americas/program/presentation/barth.

[4] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16):* https://www.usenix.org/conference/osdi16/technical-sessions/presentation/hunt.

[5] Wikipedia, "Restriction of Hazardous Substances Directive," last modified on March 24, 2017: https://en.wikipedia.org/wiki/Restriction_of_Hazardous_Substances_Directive.

[6] I. Thomson, "That CIA Exploit List in Full: The Good, the Bad, and the Very Ugly," *The Register*, March 4, 2017: https://www.theregister.co.uk/2017/03/08/cia_exploit_list_in_full/.

[7] T. Feldman and G. Gibson, "Shingled Magnetic Recording: Areal Density Increase Requires New Data Management," *;login:*, vol. 38, no. 3 (June 2013), pp. 22–30: https://www.usenix.org/system/files/login/issues/1306_login_online.pdf.

Letter to the Editor

Hi folks,

I just read the interview with Amit in *;login:* while I'm on the road to AsiaCCS. Great interview, and it's nice to see renewed interest in security for embedded devices.

Tock sounds interesting and I'll definitively check it out.

The fun part is that at AsiaCCS I'll present our work on enforcing memory safety for TinyOS [1]. We have worked on porting a CCured-like type system to nesC and enforce memory safety for a set of embedded devices at low overhead.

Embedded devices have unique advantages such as mostly static allocation, a well known stack depth, and a bunch of other interesting features that can be used to enforce strong protections, mostly statically, only falling back to a runtime check when absolutely required. In addition, there's usually a single task and dedicated resources, so we can leverage all available slack for security mechanisms.

Coincidentally, we also have an upcoming paper at Oakland [2] on protecting embedded devices using a privilege overlay. Embedded devices often run bare-metal. Our idea was to deprivilege all instructions and then, based on a static analysis, enable privileges on only a few locations and instructions. The MPU allows a dynamic configuration of these privilege overlays and enables quick switches.

It's amazing to see the renewed interest in protecting embedded systems, and I'd love to talk as we're continuing to work in that area!

Cheers,
Mathias Payer

References
[1] https://nebelwelt.net/publications/files/17AsiaCCS2.pdf.

[2] http://nebelwelt.net/publications/files/17Oakland.pdf.

ENIGMA

A USENIX CONFERENCE

JAN 16–18, 2018

SANTA CLARA, CA, USA

CALL FOR PARTICIPATION NOW OPEN!

PROGRAM CO-CHAIRS

Bryan Payne,
Netflix

Franziska Roesner,
University of Washington

SECURITY AND PRIVACY IDEAS THAT MATTER

Enigma centers on a single track of engaging talks covering a wide range of topics in security and privacy. Our goal is to clearly explain emerging threats and defenses in the growing intersection of society and technology, and to foster an intelligent and informed conversation within the community and the world. We view diversity as a key enabler for this goal and actively work to ensure that the Enigma community encourages and welcomes participation from all employment sectors, racial and ethnic backgrounds, nationalities, and genders.

Enigma is committed to fostering an open, collaborative, and respectful environment. Enigma and USENIX are also dedicated to open science and open conversations, and will make all talk media freely available on the USENIX web site.

See the complete CFP at www.usenix.org/enigma2018/cfp

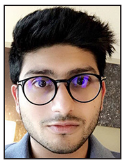
usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

FILE SYTEMS

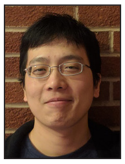
How to Fragment Your File System

ALEX CONWAY, AINESH BAKSHI, YIZHENG JIAO, YANG ZHAN, MICHAEL A. BENDER, WILLIAM JANNEN, ROB JOHNSON, BRADLEY C. KUSZMAUL, DONALD E. PORTER, JUN YUAN, AND MARTIN FARACH-COLTON

Alex Conway is a PhD student at Rutgers University, New Brunswick, New Jersey. His research interests include the theory and application of external memory algorithms and data structures, caching algorithms, and file systems. alexander.conway@rutgers.edu


Ainesh Bakshi is looking forward to starting a PhD in computer science at Carnegie Mellon University in 2017. He graduated from Rutgers University, New Brunswick, New Jersey. His research interests broadly include algorithms and theoretical machine learning, with a focus on algorithms that look to bridge the gap between theory and practice. aineshbakshi@gmail.com

Yizheng Jiao is a PhD student at Stony Brook University. His research interests focus on storage system design and implementation. Currently, he is working on a write-optimized file system for high-speed storage devices as well as efficient memory system design of big data applications. He also is interested in hacking the Linux kernel storage stack and database index engine. yizheng@cs.unc.edu


Yang Zhan is a PhD student at the University of North Carolina at Chapel Hill and is advised by Donald Porter. His research mainly focuses on write-optimized data structures. He also works on concurrent data structures. yzhan@cs.unc.edu

File systems attempt to avoid aging, or fragmentation over time, by strategically allocating space for files. System implementers and users alike treat aging as a solved problem. Here, we present a realistic workload, based on Git, that can cause these best-guess file-block-placement heuristics to fail, inducing large performance declines due to aging. This performance decline cannot be prevented with more caching or larger disks, and SSDs reduce but do not eliminate the aging effects. Our Git-based aging scheme can simulate a year of aging in under an hour. To make it easy for practitioners to incorporate aging into benchmarks, we have open-sourced our aging scripts at betrfs.org.

File-system *fragmentation* occurs when a file system stores a file or directory's contents in discontiguous ranges of disk blocks. As a file system becomes more fragmented, performance can drop significantly, since reading the file requires issuing multiple I/Os to disk. The performance drop can be particularly severe on rotating disks, where each I/O may require a disk seek. Maintaining locality in a file system as files grow, shrink, and are renamed can be challenging.

For many years, file systems did not include effective measures for avoiding fragmentation. The seminal work of Smith and Seltzer [7] showed that FFS file systems age under realistic workloads, and this aging affects performance.

Users mitigated fragmentation in early file systems by running special tools to defragment their file systems. Defragmenters reorganize file contents so that each file is stored in a contiguous range of disk blocks.

Modern file systems, on the other hand, strive to avoid fragmentation by applying best effort heuristics at allocation time. For example, file systems try to place related files close together on disk, while also leaving empty space for future files [1, 4, 5, 8]. These and other heuristics attempt to stay ahead of fragmentation wrought by normal file-system usage.

Fragmentation is thus widely viewed as a solved problem. For example, the Linux System Administrator's Guide [9] says:

> Modern Linux file systems keep fragmentation at a minimum by keeping all blocks in a file close together, even if they can't be stored in consecutive sectors. Some file systems, like ext3, effectively allocate the free block that is nearest to other blocks in a file. Therefore it is not necessary to worry about fragmentation in a Linux system.

As a result, few users run defragmentation tools. Furthermore, few file-system benchmarks attempt to age the file system before measuring its performance.

In this article, we demonstrate that modern file systems can still suffer from fragmentation under representative workloads, and we describe a simple method for quickly inducing aging. Our results suggest that fragmentation can be a first-order performance concern—some file systems slow down by over 20x over the course of our experiments. We show that fragmentation causes performance declines on both hard drives and SSDs, when there is plentiful cache available, and even on large disks with ample free space.

AND STORAGE

Michael A. Bender is a Professor of Computer Science at Stony Brook University. He was Founder and Chief Scientist at Tokutek, Inc., an enterprise database company, which was acquired by Percona in 2015. Bender's research interests span the areas of data structures and algorithms, I/O-efficient computing, scheduling, and parallel computing. Bender received his BA in applied mathematics from Harvard University in 1992 and obtained a DEA in computer science from the Ecole Normale Superieure de Lyon, France, in 1993. He completed a PhD on scheduling algorithms from Harvard University in 1998. bender@cs.stonybrook.edu

William Jannen teaches at Williams College, where he attempts to design systems that accommodate the physical characteristics of their underlying media. He is also an artist and a player of games. wjannen@cs.stonybrook.edu

Rob Johnson is a Senior Researcher at VMware and Research Assistant Professor at Stony Brook University. He developed BetrFS, invented the quotient filter, founded cache-adaptive analysis, broke the High-bandwidth Digital Content Protection (HDCP) crypto-system, and co-authored CQual, a static analysis tool that has found dozens of bugs in the Linux kernel. rob@cs.stonybrook.edu

Until July 2016, Bradley C. Kuszmaul was a Research Scientist in the Computer Science and Artificial Intelligence Laboratory at the Massachusetts Institute of Technology (MIT CSAIL), where his research focused on performance engineering of multicore software as well as data structures and algorithms that optimize cache and disk I/O. He has now joined the Bare Metal Cloud group at Oracle. bradley@mit.edu

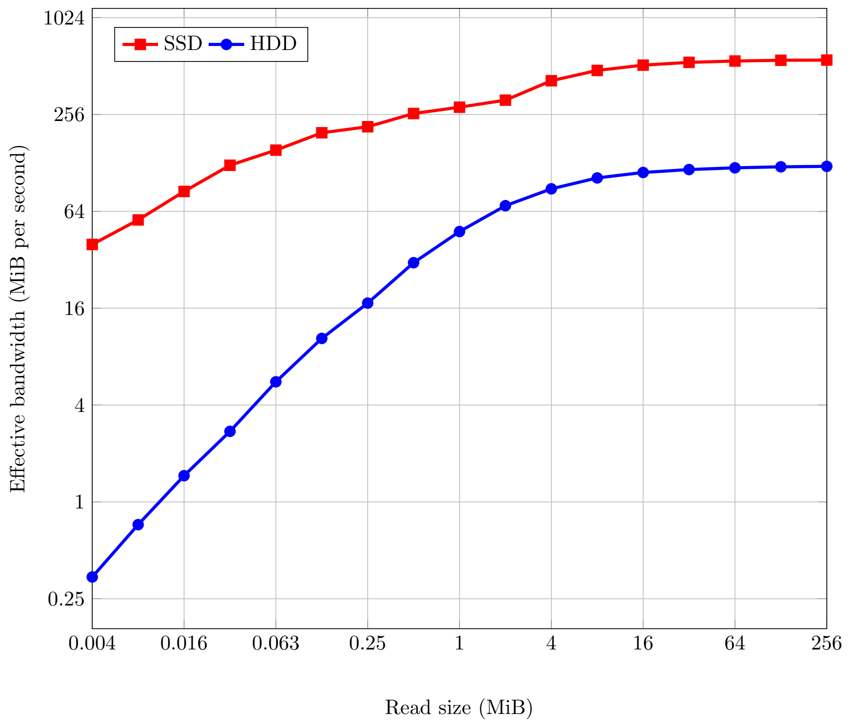


Figure 1: Effective bandwidth vs. read size (higher is better). Even on SSDs, large I/Os can yield an order of magnitude more bandwidth than small I/Os.

Fragmentation remains important because there is a large gap between sequential and random I/O performance of storage devices [2]. On rotating disks, even a few seeks can have an outsized effect on performance. For example, if a file system places a 100 MiB file in 200 disjoint pieces (i.e., 200 seeks) on a disk with 100 MiB/s bandwidth and 5 ms seek time, reading the data will take twice as long as reading it in an ideal layout.

Even on SSDs, which do not perform mechanical seeks, a decline in locality can harm performance [6]. Figure 1 shows that both HDDs and SSDs achieve substantially higher throughput when reading large blocks. On both types of hardware, we found that a surprisingly large read block of 4 MiB is necessary to achieve 75% of device bandwidth (see [2] for the specifics of our experimental setup).

Our technique for causing fragmentation makes it easy for file-system implementers and benchmarkers to incorporate aging into their evaluations. Our technique can cause years' worth of file-system aging in just a few hours and can take regular measurements as the file system ages. File systems begin aging almost immediately in our experiments, meaning that implementers and benchmarkers can use our tools to induce significant aging in under an hour.

The gold standard for realistically aging a file system is to replay a trace of file-system operations from a real system. Unfortunately, such traces are almost impossible to find. Smith and Seltzer proposed to approximate such traces by interpolating changes between successive file-system snapshots collected during a multi-year experiment [7]. Unfortunately, years-long collections of file-system snapshots have also been hard to come by.

Donald E. Porter is an Assistant Professor of Computer Science at the University of North Carolina at Chapel Hill and, by courtesy, at Stony Brook University. His research aims to improve computer system efficiency and security. In addition to recent work on write-optimization in file systems, recent projects have developed lightweight guest operating systems for virtual environments, system security abstractions, and efficient data structures for caching. porter@cs.unc.edu

Jun Yuan is an Assistant Professor of Computer Systems at Farmingdale State College of SUNY, New York. Her research interests primarily lie in systems and data structures. In addition to write-optimized file systems, she has researched programming-language-based security and access control on the Android OS. yuanj@farmingdale.edu

Martin Farach-Colton is a Professor of Computer Science at Rutgers University, New Brunswick, New Jersey. His research focuses on both the theory and practice of external memory and storage systems. He was a pioneer in the theory of cache-oblivious analysis. His current research focuses on the use of write optimization to improve performance in both read- and write-intensive big data systems. He has also worked on the algorithmics of strings and metric spaces, with applications to bioinformatics. In addition to his academic work, Professor Farach-Colton has extensive industrial experience. He was CTO and co-founder of Tokutek, a database company that was founded to commercialize his research. During 2000–2002, he was a Senior Research Scientist at Google. farach@cs.rutgers.edu

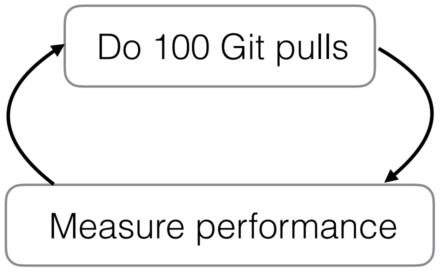


Figure 2: The Git workload

The key idea behind our aging technique is that we can view open-source Git (or any other version control system) repositories as collections of snapshots of the developers' file systems. Furthermore, replaying a repository's revision history will replay a significant portion of the developers' actual file system activity, since many developers pull changes from their collaborators multiple times per day. Thus replaying the revision history should induce fragmentation similar to that experienced by the developers when they were working on the project.

The large number of open-source projects—many of them with over a decade of history—means that we can now easily induce representative aging in file systems. Our scripts, available at betrfs.org, make it straightforward for developers and benchmarkers to integrate aging into their performance measurements.

How to Age Your File System

In the experiments in this article, we replay commits to the Linux kernel Git repository hosted on github.com. We start from the first commit and proceed in chronological order. After every 100 Git pulls, we unmount and remount the file system, clear all caches, and measure read performance (Figure 2).

We measure performance by the wall-clock time required to perform a recursive `grep` starting from the root directory of the file system. This operation descends through the directory structure, reading the content of each file. This `grep` reads a sequence of file and metadata blocks, which we call the logical order of the file-system blocks. Fragmentation occurs when two logically successive blocks are not stored in adjacent logical block addresses on the storage device. Greater fragmentation means that the average I/O size is smaller. As shown in Figure 1, this reduces the effective bandwidth, causing the `grep` to take longer.

We divide fragmentation into three categories:

- *Intrafile* is fragmentation involving blocks from the same file.
- *Interfile* is fragmentation involving blocks from two different files.
- *Metadata* is fragmentation involving at least one metadata block.

A recursive `grep` measures the impact of all these types of fragmentation on overall file-system performance.

When we run our Git aging workload, various statistics of the file system will naturally change over time as files and directories are created, modified, and deleted. For example, as a project progresses, it might include more small files, or subdirectories may include more source files. In order to make direct comparisons, we need to normalize for such changes. First, we normalize for file-system size by reporting the `grep` time in seconds per GiB. We obtain the file-system size from the output of `du`.

In order to measure potential aging, after each measurement, we copy the entire file system to a freshly formatted file system on another partition and repeat the performance

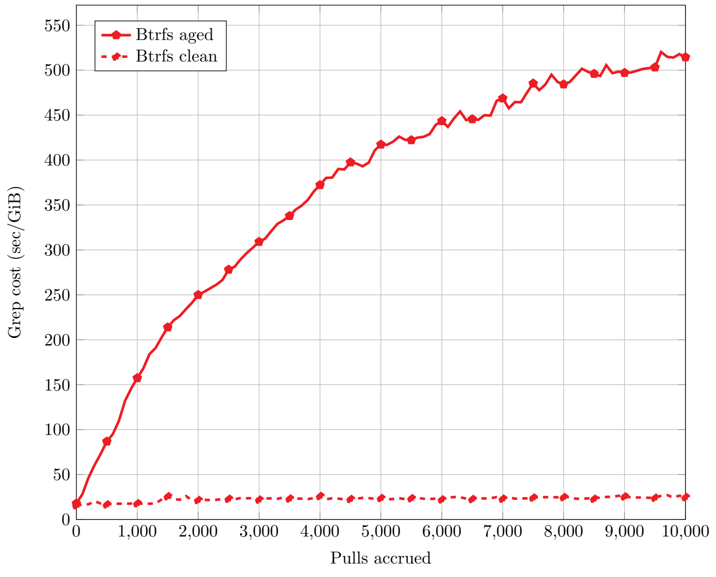


Figure 3: Git aging workload on btrfs on HDD. The overall slowdown is 20.6x. Lower is better.

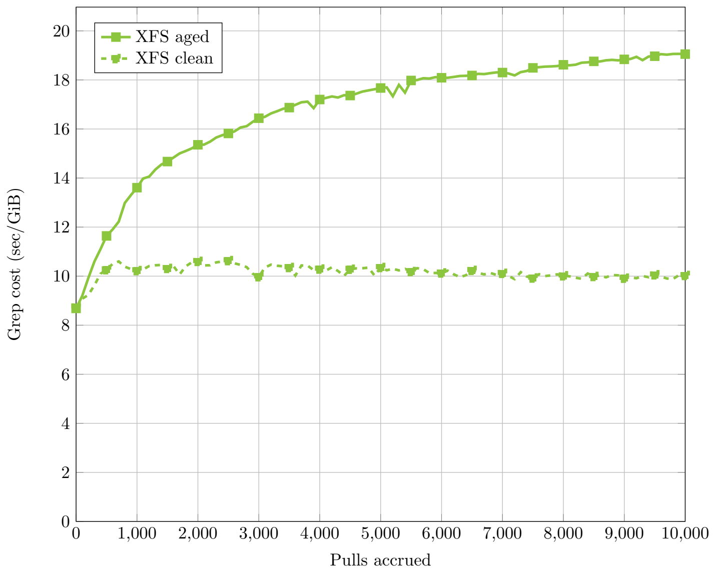


Figure 4: Git aging workload on XFS on SSD. The overall slowdown is 1.9x. Lower is better.

measurement there. We call this copy of the file system the *clean* instance, since the file system does not undergo any changes after the files are copied to it. The logical states of both file systems are the same; any performance difference between the aged and clean instances of a file system are due to the history of preceding operations.

Do modern file systems age? Figure 3 shows the results of aging btrfs with Git on a hard drive. The `grep` performance drops by a factor of 20 after 10,000 pulls. This drop in performance happens quickly; it only takes 100 pulls for a 2x slowdown and 1100 pulls for a 10x slowdown. Moreover, the `grep` ends up being very slow in absolute terms; by the end of the test it takes more than eight minutes to `grep` through 1 GiB.

In this article, we present only one file system in each experiment. Our USENIX FAST paper evaluates five popular Linux file systems under all of these experimental conditions and finds similar results [2].

Do SSDs fix aging? When we run the same workload on an SSD, we would expect to see less aging as a result of the superior random-read performance. Figure 4 shows the results of aging XFS with Git on an SSD. Although the slowdown due to aging is smaller, it is still significant. After 10,000 pulls, `greps` in the aged file-system instance are 1.9x slower than in the clean instance. After 800 pulls, the slowdown is 25%, and after 2,500 pulls, the slowdown is 50%.

Does caching fix aging? If most or all of our file system fits in cache, then the on-disk layout will not affect `grep` performance, since reads will be served from cache. We evaluated the sensitivity of the Git workloads to varying amounts of system RAM and, therefore, varying amounts of available disk cache. We use the same Git aging procedure, except that we do not flush any caches or remount the hard drive between iterations. The size of the data on disk is initially about 280 MiB and grows throughout the test to approximately 1.2 GiB.

The results for ext4 on a hard drive are summarized in Figure 5. When there is sufficient memory to keep all the data in cache, the `grep` is very fast. As soon as the size of the file system grows above a threshold, however, the warm-cache performance of `grep` quickly approaches the cold-cache performance. Furthermore, once the file system is no longer cached, the warm-cache performance is in all cases worse than the cold-cache performance of a clean copy of the file system. Unless all data fits into cache, therefore, fragmentation is a major driver of file-system performance.

Do big disks fix aging? The results shown in Figures 3 and 4 were performed on a 20 GiB partition in which the file system size never exceeded 1.2 GiB; therefore, the partition is never more than 6% full. If we run the Git workload on partitions of different sizes, as shown in Figure 6, we see that having a larger partition does not eliminate (or even mitigate) aging.

In fact, as the partition gets larger, the clean performance of ext4 gets worse. This is because ext4 spreads data across the partition in order to leave room for future files. Thus, the larger partition size actually results in longer seeks.

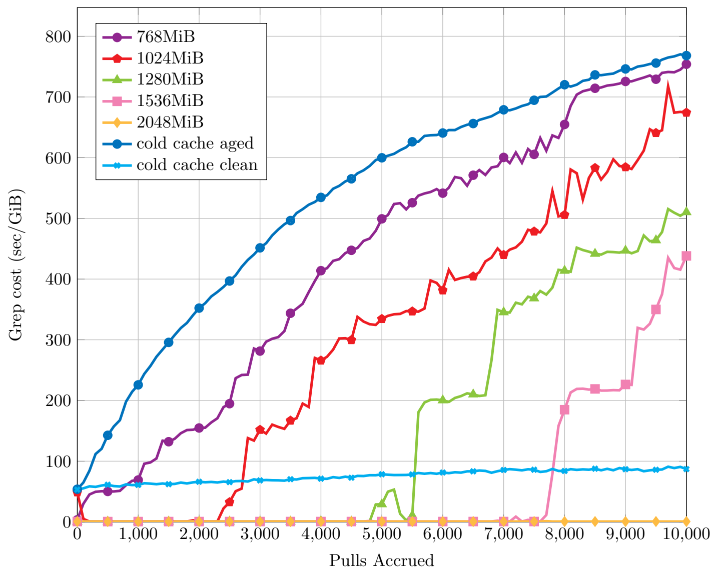


Figure 5: `grep` costs as a function of Git pulls with warm cache and varying system RAM on ext4 (top). Lower is better.

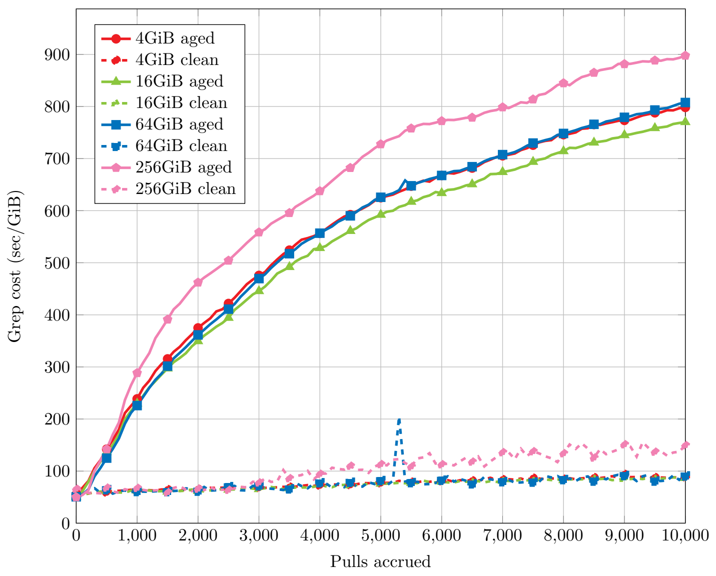


Figure 6: `grep` costs as a function of Git pulls with varying partition size on ext4. Lower is better.

Conclusion

The experiments above show that modern file systems can still age substantially under workloads representative of a typical software developer's file-system usage. They also show that SSDs, caching, and large disks do not prevent aging in today's file systems, though SSDs can help.

Furthermore, these results demonstrate that many modern file system design features, such as delayed allocation, cylinder or block groups, and extents, do not prevent aging. The file systems in these benchmarks included some or all of these features, but they aged nonetheless.

Our USENIX FAST paper delves into other file-system design tradeoffs related to aging and confirms that our research prototype file system, BetrFS [3, 10], exhibits almost no aging [2].

Our Git-based method for inducing aging makes it easy to incorporate aging into file-system benchmarks. Our scripts are available at betrfs.org.

Acknowledgments

Part of this work was done while Jiao, Porter, Yuan, and Zhan were at Stony Brook University. This research was supported in part by NSF grants CNS-1409238, CNS-1408782, CNS-1408695, CNS-1405641, CNS-1161541, IIS-1247750, CCF-1314547, a NetApp Faculty Fellowship, and VMware.

References

[1] R. Card, T. Ts'o, and S. Tweedie, "Design and Implementation of the Second Extended Filesystem," in *Proceedings of the First Dutch International Symposium on Linux,* pp. 1–6: http://e2fsprogs.sourceforge.net/ext2intro.html.

[2] A. Conway, A. Bakshi, Y. Jiao, Y. Zhan, R. Johnson, B. C. Kuszmaul, and M. Farach-Colton, "File Systems Fated for Senescence? Nonsense, Says Science!" in *Proceedings of the 15th USENIX Conference on Fi-le and Storage Technologies (FAST '17),* pp. 45–58: https://www.usenix.org/system/files/conference/fast17/fast17-conway.pdf.

[3] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter, "BetrFS: A Right-Optimized Write-Optimized File System," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15),* pp. 301–315: https://www.usenix.org/system/files/conference/fast15/fast15-paper-jannen_william.pdf.

[4] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The New ext4 Filesystem: Current Status and Future Plans," in *Proceedings of the Ottawa Linux Symposium (OLS),* vol. 2 (2007), pp. 21–34: https://www.kernel.org/doc/ols/2007/ols2007v2-pages-21-34.pdf.

[5] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems (TOCS),* vol. 2, no. 3 (August 1984), pp. 181–197: https://cs162.eecs.berkeley.edu/static/readings/FFS84.pdf.

[6] C. Min, K. Kim, H. Cho, S. Lee, and Y. I. Eom, "SFS: Random Write Considered Harmful in Solid State Drives," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12),* pp. 139–154: https://www.usenix.org/system/files/conference/fast12/min.pdf.

[7] K. A. Smith and M. Seltzer, "File System Aging—Increasing the Relevance of File System Benchmarks," in *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS),* pp. 203–213: https://www.eecs.harvard.edu/margo/papers/sigmetrics97-fs/paper.pdf.

[8] S. Tweedie, "EXT3, Journaling Filesystem," *Proceedings of the Ottawa Linux Symposium (OLS),* (July 20, 2000), pp. 24–29: http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.pdf.

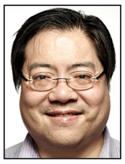
[9] L. Wirzenius, J. Oja, S. Stafford, and A. Weeks, *Linux System Administrator's Guide*, The Linux Documentation Project, Version 0.9, 2004: http://www.tldp.org/LDP/sag/sag.pdf.

[10] J. Yuan, Y. Zhan, W. Jannen, P. Pandey, A. Akshintala, K. Chandnani, P. Deo, Z. Kasheff, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter, "Optimizing Every Operation in a Write-Optimized File System," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16),* pp. 1–14: https://www.usenix.org/system/files/conference/fast16/fast16-papers-yuan.pdf.

Evolving Ext4 for Shingled Disks

ABUTALIB AGHAYEV, THEODORE TS'O, GARTH GIBSON, AND PETER DESNOYERS

Abutalib Aghayev is a PhD student in the Computer Science Department at Carnegie Mellon University. His research interests include operating systems, file and storage systems, and, recently, distributed machine learning systems. agayev@cs.cmu.edu



Theodore Ts'o started working with Linux in September 1991 and is the first North American Linux kernel developer. He also served as the tech lead for the MIT Kerberos V5 development team and as a chair of the IP Security working group at the IETF. He previously was CTO for the Linux Foundation and is currently employed at Google. Theodore is a Debian Developer, and maintains the ext4 file system in the Linux kernel. He is also the maintainer and original author of the e2fsprogs userspace utilities for the ext2, ext3, and ext4 file systems. tytso@thunk.org

Garth Gibson is a Professor of Computer Science and an Associate Dean in the School of Computer Science at Carnegie Mellon University. Garth's research is split between scalable storage systems and distributed machine learning systems, and he has had his hand in the creation of the RAID taxonomy, the Panasas PanFS parallel file system, the IETF NFS v4.1 parallel NFS extensions, and the USENIX Conference on File and Storage Technologies (FAST). garth@cs.cmu.edu

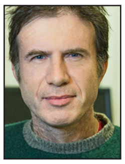
Multi-terabyte hard disks today use Shingled Magnetic Recording (SMR), a technique that increases capacity at the expense of more costly random writes. We introduce ext4-lazy, a small change to the popular Linux ext4 file system that eliminates a major source of random writes—the metadata writeback—significantly improving performance on SMR disks in general, as well as on conventional disks for metadata-heavy workloads in particular. In this article, we briefly explain why SMR disks suffer under random writes and how ext4-lazy helps.

To cope with the exponential growth of data, as well as to stay competitive with NAND flash-based solid state drives (SSDs), hard disk vendors are researching capacity-increasing technologies. Shingled Magnetic Recording (SMR) is one such technique that allows disk manufacturers to increase areal density with existing fabrication methods. So far, the industry has introduced two kinds of SMR disks: Drive-Managed (DM-SMR) and Host-Managed (HM-SMR). HM-SMR disks have a novel backward-incompatible interface that requires changes to the I/O stack and, therefore, are not widely deployed. DM-SMR disks, on the other hand, are a drop-in replacement for Conventional Magnetic Recording (CMR) disks that offer high capacity with the traditional block interface. Millions of DM-SMR disks have been shipped; in the rest of the article, therefore, we will use SMR disk as a shorthand for DM-SMR disk.

If you buy a multi-terabyte disk today, there is a good chance that it is an SMR disk in disguise, which is easy to tell: unlike CMR disks, SMR disks suffer performance degradation when subjected to continuous random write traffic, as Figure 1 shows.

One approach to adopting SMR disks is to develop a file system from scratch based on their performance characteristics. But file systems are complex and critical pieces of code that take years to mature. Therefore, we take an evolutionary approach to adopting these disks: we make a small change to the popular Linux file system, ext4, that significantly improves its performance on SMR disks by avoiding random metadata writes.

We introduce a simple technique that we call *lazy writeback journaling,* and we call a version of ext4 using our journaling technique *ext4-lazy*. Like other journaling file systems, by default ext4 writes metadata twice; as Figure 2a shows, it first writes the metadata block to a temporary location *J* in the journal and then marks the block as *dirty* in memory. Once the block has been in memory for long enough, a *writeback* thread writes the block to its static location *S*, resulting in a random write. Although metadata writeback is typically a small portion of a workload, it results in many random writes. Ext4-lazy, on the other hand, marks the block as *clean* after writing it to the journal, to prevent the writeback, and inserts a mapping (*S, J*) to an in-memory map allowing the file system to access the block in the journal, as seen in Figure 2b. Since the journal is written sequentially to a *circular* log, overwriting a metadata block is not possible. Therefore, ext4-lazy writes an updated block to the head of the log, updating the map and invalidating the old copy of the block. Ext4-lazy uses a large journal so that it can continue writing updated blocks while reclaiming the space from the



Peter Desnoyers is an Associate Professor at Northeastern University. He worked for Apple, Motorola, and a number of startups for 15 years before getting his PhD at the University of Massachusetts, Amherst, in 2007. He received BS and MS degrees from MIT. His main focuses are storage, particularly the integration of emerging storage technologies, and cloud computing. pjd@ccs.neu.edu

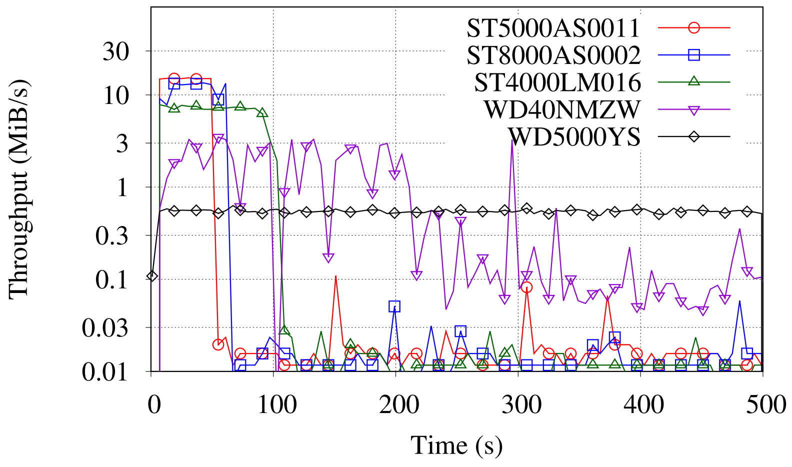


Figure 1: Throughput of CMR and SMR disks from Table 1 under 4 KiB random write traffic. The CMR disk (WD500YS) has a stable but low throughput under random writes. SMR disks, on the other hand, have a short period of high throughput followed by a continuous period of ultra-low throughput.

Type	Vendor	Model	Capacity	Form Factor
SMR	Seagate	ST8000AS0002	8 TM	3.5 inch
SMR	Seagate	ST5000AS0011	5 TB	3.5 inch
SMR	Seagate	ST4000LM016	4 TB	2.5 inch
SMR	Western Digital	WD40NMZW	4 TB	2.5 inch
CMR	Western Digital	WD5000YS	500 MB	3.5 inch

Table 1: CMR and SMR disks from two vendors used for evaluation

invalidated blocks. During mount, it reconstructs the in-memory map from the journal resulting in a modest increase in mount time. Results show that ext4-lazy significantly improves performance on SMR disks in general, as well as on CMR disks for metadata-heavy workloads in particular.

Our main contribution to ext4 includes the design, implementation, and evaluation of ext4-lazy on SMR and CMR disks. The change we make is minimally invasive—we modify 80 lines of existing code and introduce the new functionality in additional files totaling 600 lines of C code. As we show in the evaluation section, even on a metadata-light file server benchmark where the metadata writes make up less than 1% of total writes, with stock ext4 the SMR disk appears unresponsive for almost an hour with near-zero throughput. With ext4-lazy, on the other hand, the SMR disk does not suffer such a behavior and completes 1.7–5.4x faster. For directory traversal and metadata-heavy workloads, ext4-lazy achieves 2–13x improvement on both SMR and CMR disks.

Background

A high-level introduction to SMR technology has been previously presented in *;login:* [3]. Readers interested in nitty-gritty details of how an SMR disk works and why it suffers under random writes may refer to the detailed study [1] of one such disk. Here, we give just enough background on SMR disks and ext4 journaling to make the rest of the article understandable.

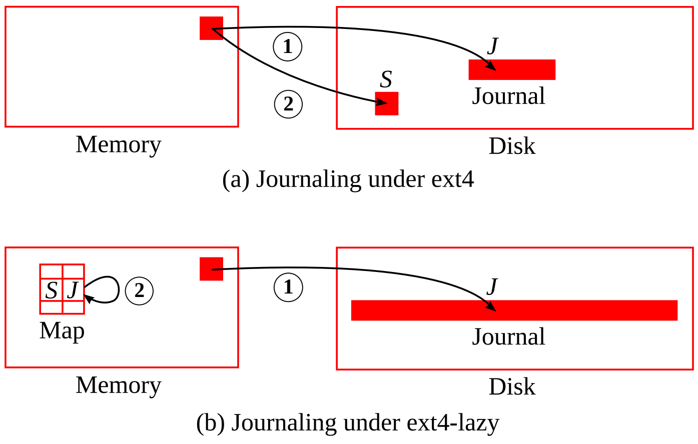


Figure 2: (a) Ext4 writes a metadata block to disk twice. It first writes the metadata block to the journal at some location *J* and marks it dirty in memory. Later, the writeback thread writes the same metadata block to its static location *S* on disk, resulting in a random write. (b) Ext4-lazy writes the metadata block approximately once to the journal and inserts a mapping (*S, J*) to an in-memory map so that the file system can find the metadata block in the journal.

SMR

As a concrete example, one SMR disk used in our evaluation consists of ≈ 30 MiB bands that are the smallest units that must be written sequentially. Overwriting a random block in a band requires read-modify-write (RMW) of the whole band. This results in reading a band, modifying it in memory, writing the updated band to a temporary band (since overwriting the original band is not atomic and could corrupt the old data if power is lost), and finally overwriting the original band, generating ≈ 90 MiB disk I/O. To hide the cost of random writes, the disk uses a *persistent cache* for handling bursts of random writes—incoming random writes are written to the persistent cache, and the bands are updated using RMW during the idle times, emptying the persistent cache. If the burst of random writes is large enough to fill the persistent cache, the throughput of the disk drops because every incoming write requires RMW of the corresponding band. Sequential writes, on the other hand, are detected and written directly to bands, bypassing the persistent cache.

Ext4 and Journaling

The ext4 file system evolved from ext2, which was influenced by Fast File System (FFS). Similar to FFS, ext2 divides the disk into *cylinder groups*—or as ext2 calls them, *block groups*—and tries to put all blocks of a file in the same block group. To further increase locality, the metadata blocks (*inode bitmap, block bitmap*, and *inode table*) representing the files in a block group are also placed within the same block group, as Figure 3a shows. In ext2 the size of a block group was limited to 128 MiB—the maximum number of 4 KiB data blocks that a 4 KiB block bitmap can represent. Ext4 introduced *flexible block groups* or *flex_bgs*—a set of contiguous block groups whose metadata is consolidated in the first 16 MiB of the first block group within the set, as shown in Figure 3b.

Ext4 ensures metadata consistency via journaling, but it does not implement journaling itself; rather, it uses a generic kernel layer called the *Journaling Block Device* that runs in a separate kernel thread called *jbd2*. In response to file system operations, ext4 reads metadata blocks from disk, updates them in memory, and exposes them to jbd2 for journaling. For increased performance, jbd2 batches metadata updates from multiple file system operations (by default, for five seconds) into a *transaction* buffer and atomically *commits* the transaction to the journal—a circular log of transactions. After a commit, jbd2 marks the in-memory copies of metadata blocks as dirty so that the writeback thread would write them to their static locations.

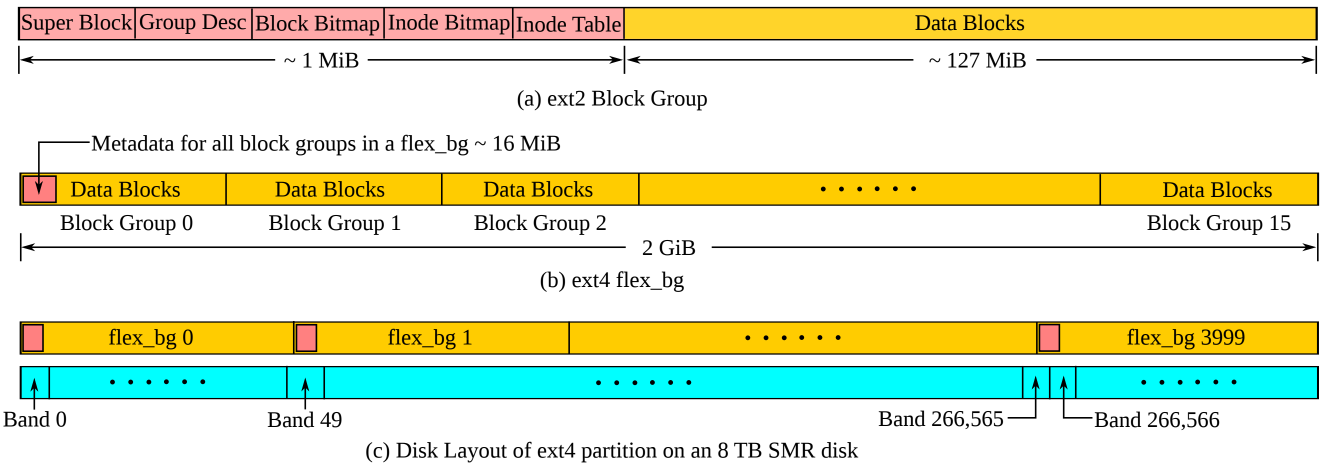


Figure 3: (a) In ext2, the first megabyte of a 128 MiB block group contains the metadata blocks describing the block group, and the rest is data blocks. (b) In ext4, a single flex bg (flexible block group) concatenates multiple (16 in this example) block groups into one giant block group and puts all of the metadata in the first block group. (c) Modifying data in a flex bg will result in a metadata write that may dirty one or two bands, seen at the boundary of bands 266,565 and 266,566.

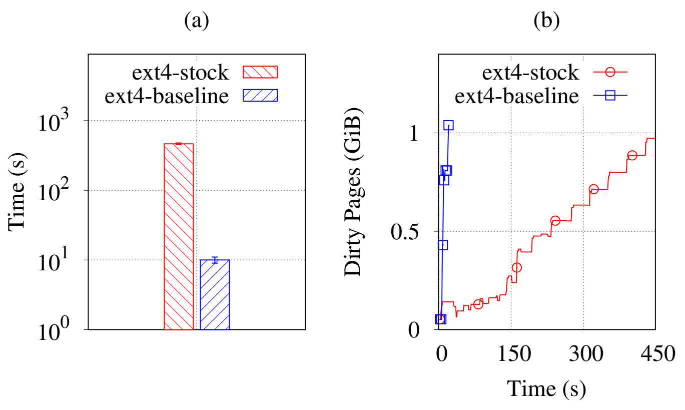


Figure 4: (a) Completion time for a benchmark creating 100,000 files on ext4-stock (ext4 with 128 MiB journal) and on ext4-baseline (ext4 with 10 GiB journal). (b) The volume of dirty pages during benchmark runs obtained by sampling `/proc/meminfo` every second.

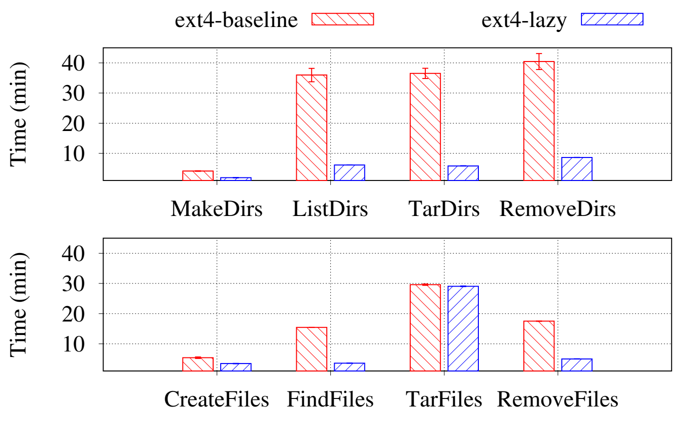


Figure 5: Microbenchmark runtimes on ext4-baseline and ext4-lazy

On SMR disks, when the metadata blocks are eventually written back, they dirty the bands that are mapped to the metadata regions in a flex_bg, as seen in Figure 3c. Since a metadata region is not aligned with a band, metadata writes to it may dirty zero, one, or two extra bands, depending on whether the metadata region spans one or two bands and whether the data around the metadata region has been written.

Design of Ext4-lazy

At a high level, ext4-lazy adds the following components to ext4 and jbd2:

Map: Ext4-lazy tracks the location of metadata blocks in the journal with an in-memory map that associates the static location *S* of a metadata block with its location *J* in the journal. The mapping is updated whenever a metadata block is written to the journal, as shown in Figure 2b.

Indirection: In ext4-lazy, all accesses to metadata blocks go through the map. If the most recent version of a block is in the journal, there will be an entry in the map pointing to it; if no entry is found, then the copy at the static location is up-to-date.

Cleaner: The cleaner in ext4-lazy reclaims space from locations in the journal that have become invalidated by the writes of new copies of the same metadata block.

Map reconstruction on mount: On every mount, ext4-lazy reads the descriptor blocks from the transactions between the tail and the head pointer of the journal and populates map.

Evaluation

We evaluate ext4-lazy on a system with a quad-core Intel i7-3820 (Sandy Bridge) 3.6 GHz CPU, 16 GB of RAM running Linux kernel 4.6, using the disks listed in Table 1. One surprising finding of our work was that the default journal size on ext4 is a bottleneck for metadata-heavy workloads. Figure 4 shows that just by increasing the journal size, a metadata-heavy workload completes over 40x faster. As a result, the recent version of e2fsprogs has increased the default journal size from 128 MiB to 1 GiB for file systems over 128 GiB. Readers interested in the details may refer to our paper [2]. Since enabling a large journal on ext4 is a command-line option to `mkfs`, we choose ext4 with a 10 GiB journal as our baseline.

Next, we first show that ext4-lazy achieves significant speedup on the CMR disk WD5000YS from Table 1 for metadata-heavy workloads, and specifically for massive directory traversal workloads. We then show that on SMR disks, ext4-lazy provides significant improvement on both metadata-heavy and metadata-light workloads.

Ext4-lazy on a CMR Disk

For metadata-heavy workloads we use the following benchmarks. *MakeDirs* creates 800,000 directories in a directory tree of depth 10. The directory tree is also used by the following benchmarks: *ListDirs* runs `ls -lR` on the directory tree, *TarDirs* creates a tarball of the directory tree, and *RemoveDirs* removes the directory tree.

CreateFiles creates 600,000 files each of size 4 KiB in a new directory tree of depth 20. The directory tree is also used by the following benchmarks: *FindFiles* runs find on the directory tree, *TarFiles* creates a tarball of the directory tree, and *RemoveFiles* removes the directory tree. All of the benchmarks start with a cold cache, set up by echoing “3” to /proc/sys/vm/drop_caches.

As Figure 5 shows, benchmarks that are in the file/directory create category (MakeDirs, CreateFiles) complete 1.5–2x faster on ext4-lazy than on ext4-baseline, while the remaining benchmarks that are in the directory-traversal category—except TarFiles—complete 3–5x faster. We choose MakeDirs and RemoveDirs as a representative of each category and analyze their performance in detail below.

	Metadata Reads (MiB)	Metadata Writes (MiB)	Journal Writes (MiB)
MakeDirs/ext4-baseline	143.7±2.8	4,631±33.8	4,735±0.1
MakeDirs/ext4-lazy	144±4	0	4,707±1.8
RemoveDirs/ext4-baseline	4,066.4±0.1	322.4±11.9	1,119±88.6
RemoveDirs/ext4-lazy	4,066.4±0.1	0	472±3.9

Table 2: Distribution of the I/O types with MakeDirs and RemoveDirs benchmarks running on ext4-baseline and ext4-lazy

MakeDirs on ext4-baseline results in ≈ 4,735 MiB of journal writes that are transaction commits containing metadata blocks, as seen in the first row of Table 2 and at the center in Figure 6a; as the dirty timer on the metadata blocks expires, they are written to their static locations, resulting in a similar amount of metadata writeback. The block allocator is able to allocate large contiguous blocks for the directories, because the file system is fresh. Therefore, in addition to journal writes, metadata writeback is sequential as well. The write time dominates the runtime in this workload: hence, by avoiding metadata writeback and writing only to the journal, ext4-lazy halves the writes as well as the runtime, as seen in the second row of Table 2 and Figure 6b. On an aged file system, the metadata writeback is more likely to be random, resulting in even higher improvement on ext4-lazy.

An interesting observation about Figure 6b is that although the total volume of metadata reads—shown as periodic vertical spreads—is ≈ 140 MiB (3% of total I/O in the second row of Table 2), they consume over 30% of runtime due to long seeks across the disk. In this benchmark, the metadata blocks are read from their static locations because we run the benchmark on a fresh file system, and the metadata blocks are still at their static locations. As we show next, once the metadata blocks migrate to the journal, reading them is much faster since no long seeks are involved.

In RemoveDirs benchmark, on both ext4-baseline and ext4-lazy, the disk reads ≈ 4,066 MiB of metadata, as seen in the last two rows of Table 2. However, on ext4-baseline the metadata blocks are scattered all over the disk, resulting in long seeks as indicated by the vertical spread in Figure 6c, while on ext4-lazy they are within the 10 GiB region in the journal, resulting in only short seeks, as Figure 6d shows. Ext4-lazy also benefits from skipping metadata writeback, but most of the improvement comes from eliminating long seeks for metadata reads. The significant difference in the volume of journal writes between ext4-baseline and ext4-lazy seen in Table 2 is caused by metadata write coalescing: Since ext4-lazy completes faster, there are more operations in each transaction, with many modifying the same metadata blocks, each of which is only written once to the journal.

The improvement in the remaining benchmarks is also due to reducing seeks to a small region and avoiding metadata writeback. We do not observe a dramatic improvement in TarFiles, because unlike the rest of the benchmarks that read only metadata from the journal, TarFiles also reads data blocks of files that are scattered across the disk. Massive directory traversal workloads are a constant source of frustration for users of most file systems. One of the biggest benefits of consolidating metadata in a small region is an order-of-magnitude improvement in such workloads.

Ext4-lazy on SMR Disks

An additional critical factor for file systems when running on SMR disks is the cleaning time after a workload. A file system resulting in a short cleaning time gives the disk a better chance of emptying the persistent cache during idle times of a bursty I/O workload, and has a higher chance of continuously performing at the persistent cache speed, whereas a file system resulting in a long cleaning time is more likely to force the disk to interleave cleaning with file system user work.

In the next section we show microbenchmark results on just one SMR disk—ST8000AS0002 from Table 1. At the end of every benchmark, we run a vendor-provided script that polls the disk until it has completed background cleaning and reports the total cleaning time, which we report in addition to the benchmark runtime. We achieve similar normalized results for the remaining disks, which we skip to save space.

Microbenchmarks

Figure 7 shows results of the microbenchmarks (see section "Ext4-lazy on a CMR Disk") repeated on ST8000AS0002 with a 2 TB partition, on ext4-baseline and ext4-lazy. MakeDirs and CreateFiles do not fill the persistent cache, and, therefore, they typically complete 2–3x faster than on CMR disk. Similar to CMR disk, MakeDirs and CreateFiles are 1.5–2.5x faster on ext4-lazy. On the other hand, ListDir, for example, one of the remaining directory traversal benchmarks, completes 13x faster on ext4-lazy, as compared to 5x faster on CMR disk.

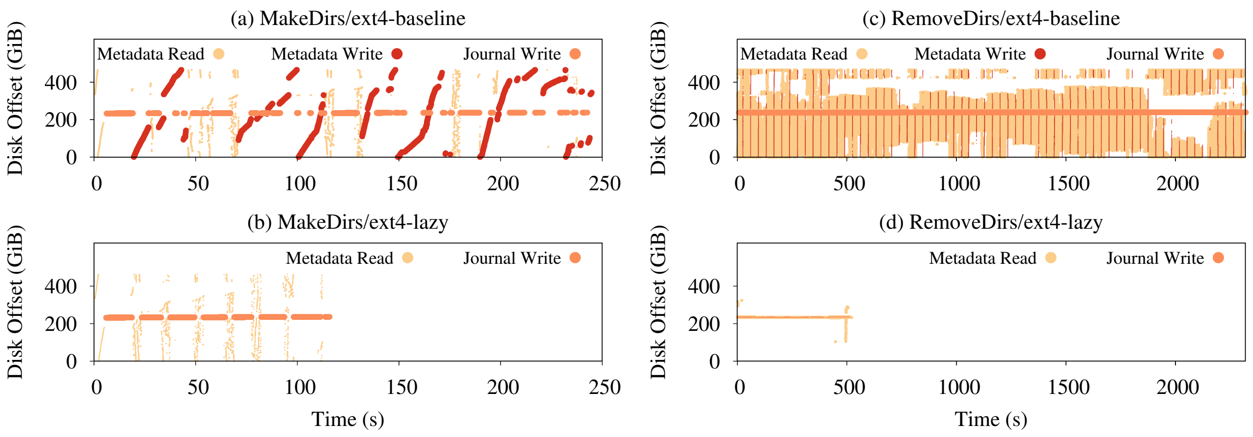


Figure 6: Disk offsets of I/O operations during MakeDirs and RemoveDirs microbenchmarks on ext4-baseline and ext4-lazy. Metadata reads and writes are spread out while journal writes are at the center. The dots have been scaled based on the I/O size. In part (d), journal writes are not visible due to low resolution. These are pure metadata workloads with no data writes.

The cleaning times for ListDirs, FindFiles, TarDirs, and TarFiles are zero because they do not write to disk—TarDirs and TarFiles write their output to a different disk. However, cleaning time for MakeDirs on ext4-lazy is zero as well, compared to ext4-baseline's 846 seconds, despite having written over 4 GB of metadata, as Table 2 shows. Being a pure metadata workload, MakeDirs on ext4-lazy consists of journal writes only, as Figure 6b shows, all of which are streamed, bypassing the persistent cache and resulting in zero cleaning time. Similarly, cleaning time for RemoveDirs and RemoveFiles are 10 and 20 seconds, respectively, on ext4-lazy compared to 590 and 366 seconds on ext4-baseline, because these too are pure metadata workloads resulting in only journal writes for ext4-lazy. During deletion, however, some journal writes are small and end up in persistent cache, resulting in short cleaning times.

File Server Macrobenchmark

Our file server benchmark creates a working set of 10,000 files spread sparsely across 25,000 directories, with file sizes ranging from 512 bytes to 1 MiB, and then executes 100,000 transactions with the I/O size of 1 MiB. In total, the benchmark writes 37.89 GiB and reads 31.54 GiB of data from user space.

Table 3 shows the distribution of write types completed by a ST8000AS0002 SMR disk with a 400 GB partition during the benchmark. On ext4-baseline, metadata writes make up 1.6% of total writes. Although the unique amount of metadata is only ≈120 MiB, as the storage slows down, metadata writeback increases slightly, because each operation takes a long time to complete, and the writeback of a metadata block occurs before the dirty timer is reset.

The benchmark completes more than 2x faster on ext4-lazy, in 461 seconds, as seen in Figure 8. On ext4-lazy, the disk sustains 140 MiB/s throughput and fills the persistent cache in 250 seconds, and then drops to a steady 20 MiB/s until the end of the run. On ext4-baseline, however, the large number of small metadata writes reduces throughput to 50 MiB/s, taking the disk 450 seconds to fill the persistent cache. Once the persistent cache fills, the disk interleaves cleaning and file system user work, and small metadata writes become prohibitively expensive, as seen, for example, between seconds 450 and 530. During this period we do not see any data writes, because the writeback thread alternates between page cache and buffer cache when writing dirty blocks, and it is the buffer cache's turn. We do, however, see journal writes because jbd2 runs as a separate thread and continues to commit transactions.

The benchmark completes even more slowly on a full 8 TB ext4 partition, as seen in Figure 9, because ext4 spreads the same workload over more bands. With a small partition, updates to different files are likely to update the same metadata region. Therefore, cleaning a single band frees more space in the persistent cache, allowing it to accept more random writes. With a full

	Data Writes (MiB)	Metadata Writes (MiB)	Journal Writes (MiB)
ext4-baseline	32,917±9.7	563±0.9	1,212±12.6
ext4-lazy	32,847±9.3	0	1,069±11.4

Table 3: Distribution of write types completed by a ST8000AS0002 SMR disk during a Postmark run on ext4-baseline and ext4-lazy. Metadata writes make up 1.6% of total writes in ext4-baseline, only 20% of which is unique.

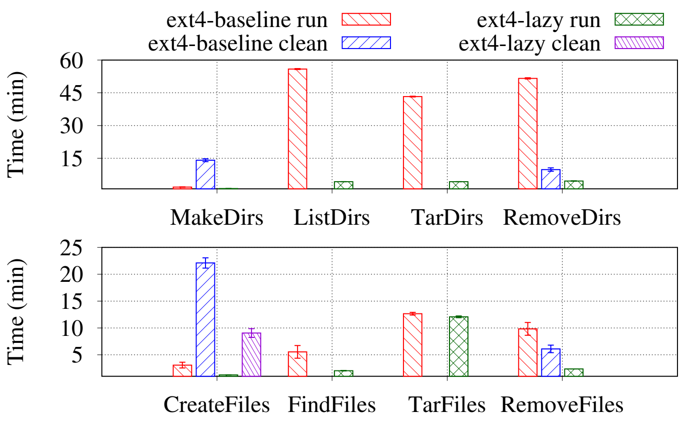


Figure 7: Microbenchmark runtimes and cleaning times on ext4-baseline and ext4-lazy running on an SMR disk. Cleaning time is the additional time after the benchmark run that the SMR disk was busy cleaning.

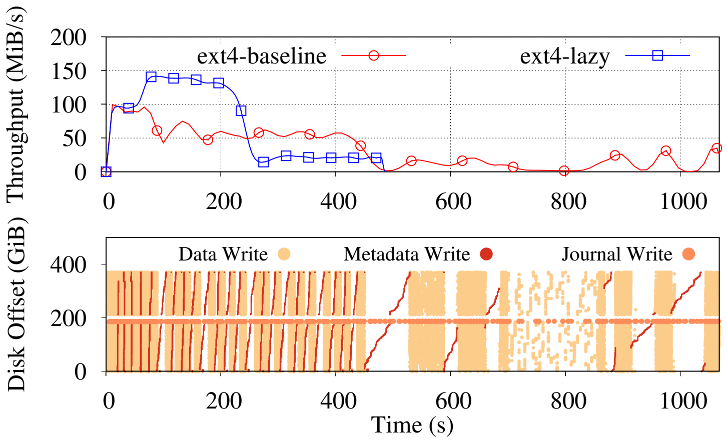


Figure 8: The top graph shows the throughput of a ST8000AS0002 SMR disk with a 400 GB partition during a file server benchmark run on ext4-baseline and ext4-lazy. The bottom graph shows the offsets of write types during the run on ext4-baseline. The graph does not reflect sizes of the writes, only their offsets.

partition, however, updates to different files are likely to update different metadata regions: now the cleaner has to clean a whole band to free a space for a single block in the persistent cache. Hence, after an hour of ultra-low throughput due to cleaning, it recovers slightly towards the end, and the benchmark completes 5.4x slower on ext4-baseline. Interested readers may refer to our paper [2] for the evaluations of all disks from Table 1.

Conclusion

Previous work has explored separating metadata from data and managing it as a log by designing a file system from scratch [4–6]. Our work, however, is the first that leverages the metadata separation idea for adapting a legacy file system to SMR disks. It shows how effective a well-chosen small change can be. It also suggests that while three decades ago it was wise for file systems depending on the block interface to scatter the metadata across the disk, today, with large memory sizes that cache metadata and with changing recording technology, putting metadata at the center of the disk and managing it as a log looks like a better choice.

We conclude with the following general takeaways:

- We think modern disks are going to practice more extensive "lying" about their geometry and perform deferred cleaning when exposed to random writes; therefore, file systems should work to eliminate structures that induce small isolated writes, especially if the user workload is not forcing them.
- With modern disks, operation costs are asymmetric: random writes have a higher ultimate cost than random reads, and, furthermore, not all random writes are equally costly. When random writes are unavoidable, file systems can reduce their cost by confining them to the smallest perimeter possible.

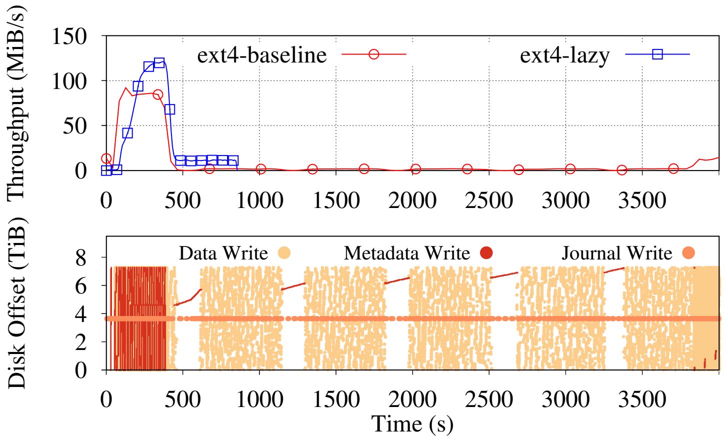


Figure 9: The top graph shows the throughput of a ST8000AS0002 SMR disk with a full 8 TB partition during a file server benchmark run on ext4-baseline and ext4-lazy. The bottom graph shows the offsets of write types during the run on ext4-baseline. The graph does not reflect sizes of the writes, only their offsets.

References

[1] A. Aghayev and P. Desnoyers, "Skylight—A Window on Shingled Disk Operation," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, pp. 135–149: https://www.usenix.org/system/files/conference/fast15/fast15-paper-aghayev.pdf.

[2] A. Aghayev, T. Ts'o, G. Gibson, and P. Desnoyers, "Evolving Ext4 for Shingled Disks," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, pp. 105–120: https://www.usenix.org/system/files/conference/fast17/fast17-aghayev.pdf.

[3] T. Feldman and G. Gibson, "Shingled Magnetic Recording: Areal Density Increase Requires New Data Management," *;login:*, vol. 38, no. 2 (June 2013), pp. 22–30: http://www.pdl.cmu.edu/PDL-FTP/HECStorage/05_feldman_022-030.pdf.

[4] J. Piernas, T. Cortes, and J. Garcia, "DualFS: A New Journaling File System without Meta-Data Duplication," in *Proceedings of the 16th International Conference on Supercomputing*, 2002, pp. 137–146: http://ditec.um.es/web-ditec/ficheros/publicaciones/publicacion95.pdf.

[5] K. Ren and G. Gibson, "TableFS: Enhancing Metadata Efficiency in the Local File System," in *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*, pp. 145–156: http://www.pdl.cmu.edu/PDL-FTP/FS/CMU-PDL-13-102.pdf.

[6] Z. Zhang and K. Ghose, "hFS: A Hybrid File System Prototype for Improving Small File and Metadata Performance," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '07)*, pp. 175–187: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.160.507&rep=rep1&type=pdf.

Redundancy Does Not Imply Fault Tolerance

Analysis of Distributed Storage Reactions to Single Errors and Corruptions

AISHWARYA GANESAN, RAMNATTHAN ALAGAPPAN,
ANDREA C. ARPACI-DUSSEAU, AND REMZI H. ARPACI-DUSSEAU

Aishwarya Ganesan is a PhD student in Computer Sciences at the University of Wisconsin-Madison. Her advisors are Professors Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau. She is interested in distributed systems, file systems, and storage. ag@cs.wisc.edu

Ramnatthan Alagappan is a PhD student in computer sciences at the University of Wisconsin-Madison, advised by Professor Andrea Arpaci-Dusseau and Professor Remzi Arpaci-Dusseau. His research interests include file systems, storage, operating systems, and distributed systems. ra@cs.wisc.edu

Andrea Arpaci-Dusseau is a Professor of Computer Sciences at the University of Wisconsin-Madison. She is an expert in file and storage systems, having published more than 80 papers in this area, co-advised 20 PhD students, and received more than 10 Best Paper awards. dusseau@cs.wisc.edu

Remzi Arpaci-Dusseau is a Professor of Computer Sciences at the University of Wisconsin-Madison. His research focus is on file and storage systems, and his teaching interests lie in creating free online materials for all (e.g., http://www.ostep.org). remzi@cs.wisc.edu

We analyze how modern distributed storage systems behave in the presence of file-system faults such as data corruption and read and write errors. We characterize the behaviors of eight popular distributed storage systems, including Cassandra, Redis, and ZooKeeper. The major result of our study is that a single file-system fault introduced in one node of the cluster can induce catastrophic outcomes such as data loss, corruption, and unavailability. We find that most systems do not consistently use redundancy to recover from file-system faults. We also find that the above outcomes arise due to fundamental problems in file-system fault handling that are common across many systems. Our results have implications for the design of next generation fault-tolerant distributed storage systems.

Redundancy is a well-known technique for providing fault tolerance. Using redundancy, a system can tolerate failures of one or more of its components. For example, in a distributed storage system, data and functionality are replicated across many servers for fault tolerance. In most cases, replication can mask various failures such as system crashes, power failures, or nodes becoming inaccessible due to network failures. Modern distributed storage systems typically depend on local file systems to store and manage their data. Although replication can mask whole machine failures, local file systems exhibit a more complex failure model. For instance, certain blocks of data can become inaccessible due to an underlying latent sector error or, worse, the local file system may silently return corrupted data on reads if the underlying device block is corrupted. We call these failures file-system faults.

Several studies have shown the prevalence of errors and corruptions in disks and SSDs [1, 2, 5] that lead to these file-system faults. However, little is known about how modern distributed storage systems react to such file-system faults. Therefore, in this study, we answer the following questions: *How do distributed storage systems behave in the presence of local file-system faults? Do they use redundancy to recover from local file-system faults?*

To answer these questions, we systematically inject file-system faults into distributed storage systems and observe the effects of the injected fault. We picked a broad spectrum of distributed storage systems, implementing a variety of replication protocols such as replicated state machines, primary backup, and dynamo-style quorums.

Our fault model is very simple—we inject exactly one file-system fault into one file-system block in one node in the system at a time. We inject corruptions on reads, errors on reads, and errors on writes. Moreover, our fault model only includes data corruptions that are detectable by applications (e.g., using application-level checksums) and does not include undetectable memory corruptions.

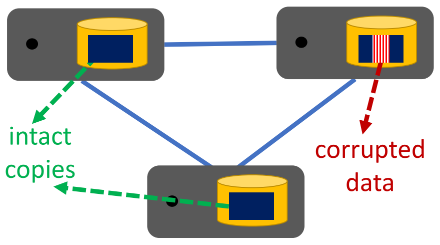


Figure 1: User expectations. The figure shows a data item replicated on three servers in a distributed storage system. When one copy is corrupted, users typically expect that redundant copies will help recover from the single corruption.

Catastrophic Outcomes	Redis	ZooKeeper	Cassandra	Kafka	RethinkDB	MongoDB	LogCabin	CockroachDB
Silent Corruption	■		■		■			
Unavailability	■	■	■	■				■
Data Loss			■	■	■			■
Query Failures			■			■		■
Reduced Redundancy	■	■	■	■	■	■	■	■

Table 1: Catastrophic outcomes: summary. The table shows the summary of catastrophic outcomes resulting from a single file-system fault. A shaded box for a system indicates that we discovered at least one instance of the outcome mentioned on the left.

A common and widespread expectation is that redundancy in higher layers (i.e., across replicas) enables recovery from local file-system faults. For instance, consider a data item that is replicated across three machines in a system as shown in Figure 1. What would a user expect if one of the copies of the data item in the system gets corrupted? Similarly, what if one of the blocks in one of the copies becomes inaccessible? It is completely reasonable for a user to expect that the corrupted data will be recoverable from the intact copies on other replicas and that the user never sees the corrupted data.

Unfortunately, from our study, we find that redundancy does not provide fault tolerance in many distributed storage systems. We find several pieces of evidence where a single file-system fault in only one node leads to catastrophic outcomes such as data loss, silent user-visible corruption, unavailability, or sometimes even the spread of corrupted data to other intact replicas. Table 1 shows the prevalence of various undesirable behaviors across multiple systems. Note that since the system has redundant copies of data and we inject only one fault at a time, these behaviors are surprising and undesirable.

Why does redundancy not imply fault tolerance? One might wonder whether the discovered outcomes arise simply due to some implementation-level bugs that could be fixed by moderate developer effort. Unfortunately, from our study, we find that the above outcomes arise due to some alarming and fundamental root causes in file-system fault tolerance that are common to many distributed storage systems.

The first fundamental problem we observe is that *faults are often undetected locally* by the nodes in a distributed storage system, leading to harmful effects such as corrupted data being returned to the users. Second, even when systems reliably detect faults, in most cases, they *simply crash* instead of using redundancy to recover from the fault. Third, many systems *do not discern corruptions caused due to crashes from other corruptions*, resulting in many data loss cases. Finally, we find that local fault-handling behaviors and global distributed protocols *interact in an unsafe manner,* leading to propagation of corruption or data loss.

As distributed storage systems are emerging as the primary choice for storing critical user data, carefully building them to tolerate file-system faults is important. Our study is a step in this direction, and we hope that our results will lead to discussions and future research to improve the resiliency of next generation cloud storage systems. The full version of our work was published in FAST '17 [3]. Our testing framework is publicly available at http://research.cs.wisc.edu/adsl/Software/cords.

Methodology

In this section, we first discuss the fault model and then describe our methodology to study how distributed storage systems react to local file-system faults.

Fault Model

Our fault model is very simple—we inject a single fault into a single file-system block exactly one node at a time. We inject these faults into file-system user data and not the file-system metadata. The reason for this is simple: the file system is responsible for maintaining the integrity of its metadata, while applications should take care of their on-disk data.

Our fault model captures the behavior of different real file systems. Consider that the nodes of a distributed storage system run on an ext4 file system. If the underlying device block is corrupted, ext4 returns corrupted data as-is to applications since it does not have checksums for user data. On the other hand, consider a file system such as btrfs that maintains checksums for user data; such a file system transforms an underlying block corruption into a read error.

To capture these different file system behaviors, our fault model injects three types of faults: corruption on reads, error on reads, and error on writes. Our fault model assumes detectable corruptions (e.g., corruptions detectable using application-level check-

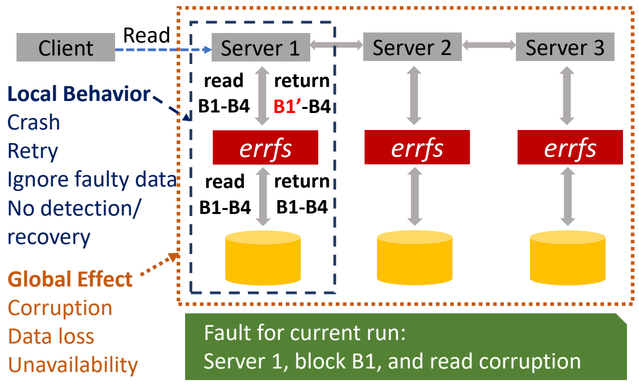


Figure 2: Fault injection methodology. errfs injects faults into one file-system block one node at a time. For each fault, we need to observe the local behavior and the global effect.

sums) and does not include arbitrary memory corruptions that are not detectable by applications (e.g., corruptions introduced before checksum computation or corruptions introduced after checksum verification).

Fault Injection

To study how distributed storage systems react to local file-system faults, we build a framework called CORDS, which includes the following key pieces: *errfs*, a user-level FUSE file system that systematically injects file-system faults, and *errbench*, a suite of system-specific workloads which drives systems to interact with their local storage.

To understand how our fault-injection methodology works, consider a distributed storage system with three nodes, as shown in Figure 2. We configure the system to run atop errfs and run a system-specific workload multiple times, each time injecting a single fault for a single file-system block in a single node. Assume that for a particular run we would like to inject a read corruption for block *B1* on server 1. After reading the blocks from the disk, errfs corrupts *B1* before returning to the server. To emulate errors, errfs does not perform the operation but simply returns an appropriate error code.

Behavior Inference

In a distributed system, multiple nodes work with their local file system to store user data. When a fault is injected in a node, we need to observe two things: first, the *local behavior* of the node where the fault is injected. Locally, the faulty node could *crash*, *retry* the operation, detect and *ignore the faulty data*, or perform *no detection or recovery*, etc.

Second, we need to observe the *global effect* of the injected fault. The global effect of a fault is the result that is externally visible. Ideally, we should not observe any harmful effect since the data

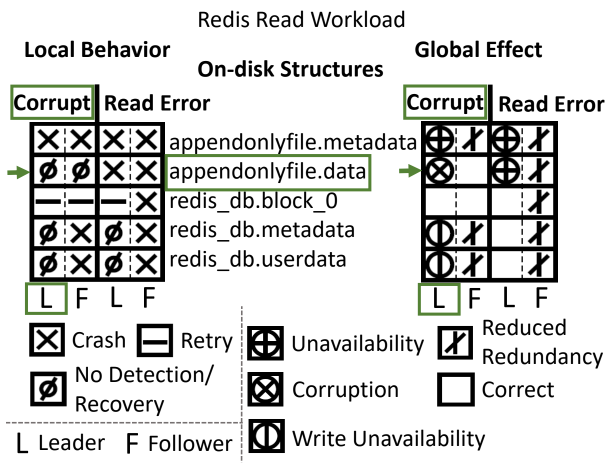


Figure 3: Behavior analysis of Redis read. The figure shows local behaviors and global effects when corruptions and read errors are injected in various on-disk logical structures during read workload in Redis. The grid on the left shows the local behavior of the node where the fault is injected, and the one on the right shows the cluster-wide global effect of the injected fault. The annotation on the top of a grid shows the type of fault: for example, "Corrupt" means that we inject data corruption using errfs. The annotation between the grids shows the on-disk logical structure in which the fault is injected. Annotations on the bottom show where a particular fault is injected (L - leader, F - follower).

is replicated and we inject only one fault at a time. Some adverse global effects that could occur include data loss, user-visible corruption, read-unavailability, write-unavailability, unavailability, or query failure. These local behaviors and global effects for a given workload and a fault might vary depending on the role played (leader or follower) by the node where the fault is injected.

Behavior Analysis

We studied the following eight distributed storage systems using CORDS, our framework for injecting faults: Redis (v3.0.4), ZooKeeper (v3.4.8), Cassandra (v3.7), Kafka (v0.9), RethinkDB (v2.3.4), MongoDB (v3.2.0), LogCabin (v1.0), and CockroachDB (beta-20160714).

An Example: Redis

To illustrate our behavior analysis, we use Redis as an example. Redis is a data structure store with a leader and set of followers. On a write request, data is appended to the *append-only file* and also replicated on to the followers. The append-only file is periodically snapshotted into the Redis *database_file*.

Figure 3 shows the behaviors of Redis when faults are injected during a read workload. We represent our results in grids like the ones shown in the figure. We inject different faults such as corruption and read or write errors into either a leader or a follower one at a time and for different on-disk structures. The on-disk structures take the form: *file_name.logical_entity*. We derive

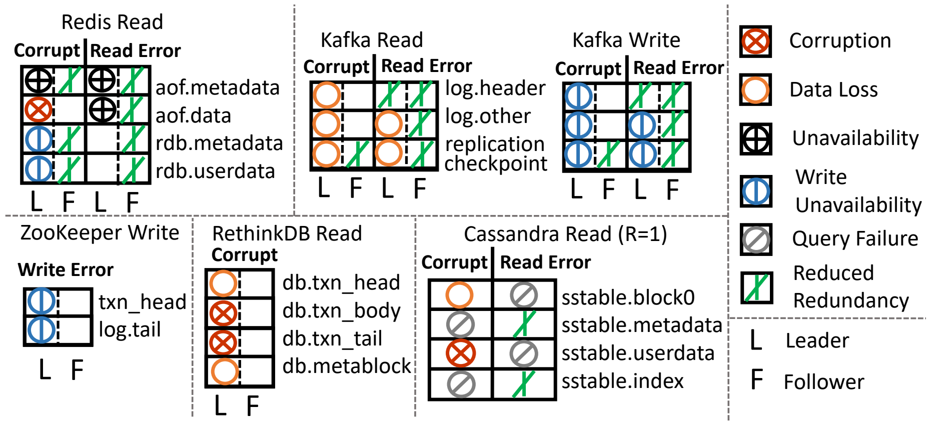


Figure 4: Redundancy does not provide fault tolerance. The figure shows a sample of catastrophic outcomes such as corruption, data loss, unavailability, query failures, and reduced redundancy that occur across many systems. These outcomes (global effects) occur when corruptions, read errors, and write errors are injected in various on-disk logical structures during read and write workloads in different distributed storage systems.

the logical entity name from our understanding of the on-disk format of the file. For each injected fault, we observe how the system behaves.

For example, when there are corruptions in the data in the append-only file on the leader (highlighted with outlining in the figure), the corruption is undetected (local behavior), and the corrupted data is silently returned (global effect). Redis does not use checksums for append-only file user data; thus, it does not detect corruptions. Moreover, the resynchronization protocol in Redis propagates corrupted user data from the leader to the followers leading to a global user-visible corruption. We repeat this analysis by running the read workload multiple times, each time injecting a different fault into a different on-disk structure.

We also repeat the analysis for other systems for read and write workloads. These results and analyses are presented in detail in our FAST '17 paper [3]. We will use the results from this behavior analysis of various systems to draw observations in the rest of this article.

Major Results

The most important overarching lesson from our study is this: a single file-system fault can induce catastrophic outcomes in most modern distributed storage systems. Despite the presence of checksums, redundancy, and other resiliency methods prevalent in distributed storage, a single file-system fault can lead to data loss, corruption, unavailability, and, in some cases, the spread of corruption to other intact replicas. Figure 4 shows a sample of results that illustrate the prevalence of catastrophic problems across multiple systems.

In most cases, the problems shown in Figure 4 are not caused by simple implementation bugs. Rather, they are caused due to some fundamental problems in file-system fault tolerance that are common to many distributed storage systems.

Fundamental Problems

We now discuss some of the fundamental root causes that are responsible for the catastrophic problems that we discover in all systems.

Faults Are Often Undetected Locally

The first fundamental problem we observe is that faults are often undetected locally. These locally undetected faults might lead to harmful global effects. For example, a locally undetected corruption could result in a global silent corruption.

Figure 5 shows how a locally undetected fault leads to harmful global effects in Cassandra. The figure shows the case where the user data in the *sstable* on one node is corrupted. Cassandra does not detect this corruption using checksums when compression is not enabled. Thus, any read request for this data item to the corrupted replica will silently receive corrupted data. Further, the

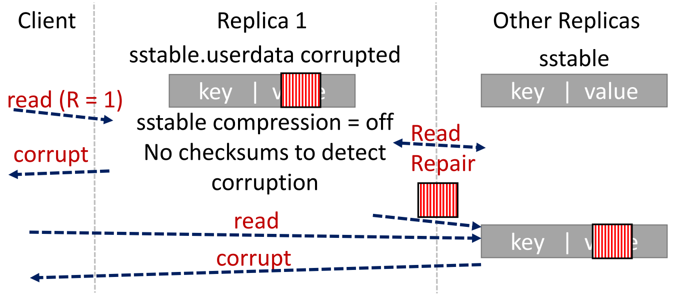


Figure 5: Faults are often undetected locally. The figure shows how a locally undetected fault can lead to harmful global effects in Cassandra.

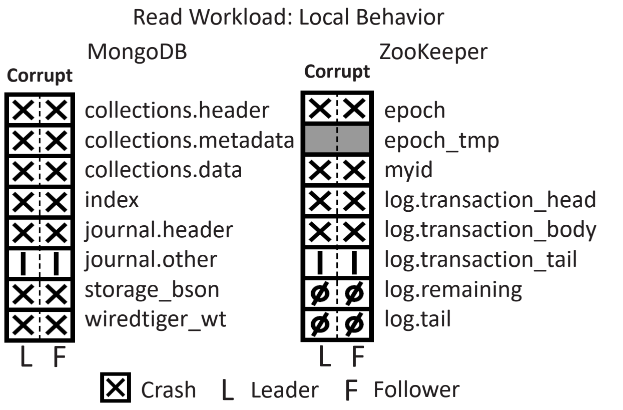


Figure 6: Crashing is the most common local reaction. The figure shows that crashing is the most common local reaction when corruptions are introduced into various on-disk structures during the read workload in MongoDB and ZooKeeper.

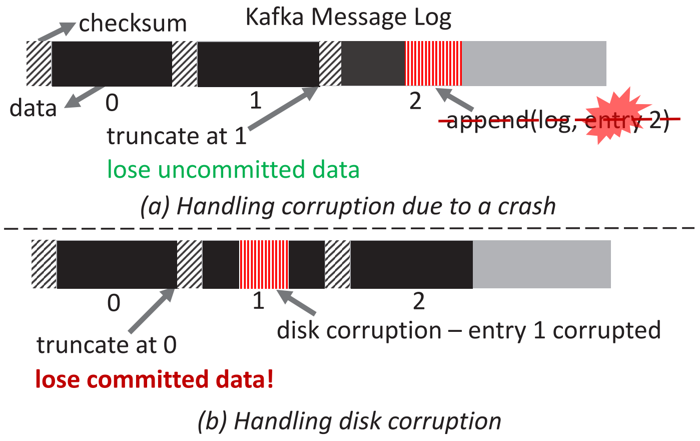


Figure 7: Crash and corruption handling are entangled. The figure shows how entanglement in crash and corruption handling could lead to a local data loss of committed data in Kafka.

read repair protocol that fixes stale versions of data propagates the corruption to other replicas. Many other systems exhibit similar problems (e.g., RethinkDB and Redis); these systems completely trust and rely upon the lower layers in the storage stack to handle data integrity problems.

Crashing Is the Most Common Reaction

The next fundamental problem is that crashing is the most common local reaction. Many systems do reliably detect faults, but in most cases they simply crash on detecting a fault instead of using redundancy to recover from the fault. For example, MongoDB and ZooKeeper have checksums for most of their on-disk data structures to detect corruptions. Figure 6 shows the local behavior of these systems when corruptions are introduced into various on-disk structures during the read workload. As shown in the figure, nodes in MongoDB and ZooKeeper simply crash on detecting a corruption. We observe the same behavior in many other systems.

Although crashing does not result in a harmful effect immediately, it introduces the possibility of an imminent unavailability. Moreover, since storage faults could be persistent, simply restarting the faulty node does not help; the node would encounter the same fault and crash again. Solving such cases requires some manual intervention, which is often error-prone and cumbersome. Although crashing may seem like a good strategy to employ, in a distributed system there are opportunities to recover from local faults using copies on other intact replicas.

Crashing and Corruption Handling Are Entangled

The next observation we make is that crash and corruption handling are entangled. We illustrate this using Kafka. Kafka is a persistent distributed message queue in which the messages are stored in a log. Incoming messages are appended to the log, and each message is checksummed. Consider that a Kafka node crashes during an append of message 2 as shown in Figure 7. When the node recovers from the crash, it detects a checksum mismatch because of the partially appended entry. As a recovery action, the node truncates the log at message 1. Note that message 2 is uncommitted as the node crashed while appending it. Hence, it is safe to truncate the uncommitted message in this case.

On the other hand, consider the case where all messages 0, 1, and 2 are persisted safely on disk, but the block holding message 1 is corrupted. Kafka detects this corruption using checksums, but it truncates the log at message 0 since it treats this disk corruption as a corruption that occurred due to a crash. Note that messages 1 and 2 were committed and it is not safe to lose them. Since Kafka conflates the handling of a disk corruption and a corruption due to a crash, it loses committed data.

Developers of RethinkDB and LogCabin agree that entanglement is a problem. Thus, there is a need to disentangle corruptions due to crashes from other types of corruptions.

Unsafe Interaction between Local and Global Protocols

Next, we observe that the local behavior of a faulty node and the global protocols interact in unsafe ways. We illustrate this again using Kafka. Recall that the Kafka node treats a disk corruption the same way it treats a corruption due to a crash, resulting in a data loss. However, this data loss is the local behavior of the corrupted node. Assume that this data loss occurred on node 1. Other nodes still have the data as shown in Figure 8.

Kafka maintains a piece of metadata that contains information about replicas that are in-sync; any node in this set has all the committed data and is eligible to become a leader. In this case,

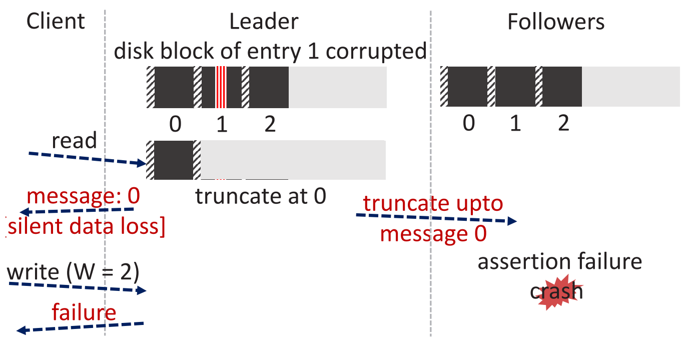


Figure 8: Unsafe interaction between local behaviors and global protocols. The figure shows how local fault-handling behaviors in Kafka interact with the global leader election protocol in an unsafe manner. Node 1, which lost committed data due to entanglement in crash and corruption handling, is elected as the leader, resulting in data loss and write unavailability.

Problem	Redis	ZooKeeper	Cassandra	Kafka	RethinkDB	MongoDB	LogCabin	CockroachDB
Locally Undetected Faults	■	■	■		■			■
Crashing on Faults	■	■	■	■	■	■	■	■
Crash Corruption Entangled		■		■	■	■	■	
Unsafe Protocol Interaction	■		■	■				
Redundancy Underutilized	■	■	■	■	■	■	■	■

Table 2: Fundamental problems summary. The table shows the summary of the fundamental problems across all the systems we studied. A shaded box for a system indicates that we observed at least one instance of the problem mentioned on the left.

node 1, which lost committed data, is not removed from the set of in-sync replicas and is elected as the leader. Thus, any further reads return only message 0, resulting in a silent data loss. Moreover, the leader also instructs the followers to truncate the log at message 0 which triggers an assertion at followers, resulting in their crash. Thus, all future writes become unavailable. The unsafe interaction between local behavior (i.e., to truncate the log) and the global protocol (leader election) in Kafka leads to a data loss and write unavailability. Thus, there is a need for synergy between local behaviors and global protocols to avoid such problems.

Fundamental Problems: Summary

Table 2 shows how the fundamental problems are common across many systems. We observe that all systems we studied simply crash on detecting a fault in many cases. In some cases, systems take incorrect recovery action on detecting a fault, leading to undesirable behaviors. We also observe that all systems miss opportunities to recover from local file-system faults using redundancy.

Conclusion

Most popular distributed systems we studied are not yet resilient to local file-system faults. Although a body of research work and enterprise storage systems provide software guidelines to tackle partial file-system faults, such wisdom has not filtered down to commodity distributed storage systems. Our findings provide motivation for distributed systems to build on existing research work to tolerate practical faults other than crashes.

Our study provides four important lessons for future distributed storage system design. First, in the world of layered storage stacks that run on commodity hardware, faults are common; thus, distributed storage systems need to detect such faults carefully. Second, in a distributed system, several unavoidable cases such as power faults and network failures can cause nodes to be unavailable. In cases where automatic recovery is possible, simply crashing is not the optimal behavior. Next, by disentangling corruptions caused by a crash from other types of corruptions and by handling them differently, storage systems can avoid many problems. Finally, local fault-handling behavior has global implications for distributed systems. Distributed storage system developers need to fully understand this interaction in order to improve reliability.

We hope that our study and results will provide direction for the design of more robust distributed storage systems. Our fault-injection framework is available at http://research.cs.wisc.edu/adsl/Software/cords.

Acknowledgments

We thank the anonymous FAST reviewers, Hakim Weatherspoon (our shepherd), and Rik Farrow for their insightful comments. We thank the members of the ADSL and the developers of CockroachDB, LogCabin, Redis, RethinkDB, and ZooKeeper for their valuable discussions.

This material was supported by funding from NSF grants CNS-1419199, CNS-1421033, CNS-1319405, and CNS1218405, DOE grant DE-SC0014935, as well as donations from EMC, Facebook, Google, Huawei, Microsoft, NetApp, Samsung, Seagate, Veritas, and VMware. Finally, we thank CloudLab [4] for providing a great environment for running our experiments. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE, or other institutions.

References

[1] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder, "An Analysis of Data Corruption in the Storage Stack," in *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, February 2008, pp. 223–238: https://www.usenix.org/legacy/event/fast08/tech/full_papers/bairavasundaram/bairavasundaram.pdf .

[2] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler, "An Analysis of Latent Sector Errors in Disk Drives," in *Proceedings of the 2007 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, June 2007: http://research.cs.wisc.edu/adsl/Publications/latent-sigmetrics07.pdf.

[3] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, February 2017, pp. 149–166: https://www.usenix.org/system/files/conference/fast17/fast17-ganesan.pdf.

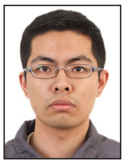
[4] R. Ricci, E. Eide, and CloudLab Team, "Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications," *;login:*, vol. 39, no. 6 (December 2014): https://www.usenix.org/publications/login/dec14.

[5] B. Schroeder, R. Lagisetty, and A. Merchant, "Flash Reliability in Production: The Expected and the Unexpected," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, February 2016, pp. 67–80: https://www.usenix.org/system/files/conference/fast16/fast16-papers-schroeder.pdf.

Scaling Namespace Operations with Giraffa File System

KONSTANTIN V. SHVACHKO AND YUXIANG (CHRIS) CHEN

Konstantin V. Shvachko is an expert in big data technologies, file systems, and storage solutions. He specializes in efficient data structures and algorithms for large-scale distributed storage systems. Konstantin is known as an open source software developer, author, inventor, and entrepreneur. He is currently a part of the Hadoop team at LinkedIn. kshvachko@linkedin.com



Yuxiang Chen is a graduate student in the School of Computer Science, Carnegie Mellon University. In summer 2016 he worked as an intern with the Hadoop Development team at LinkedIn. His research interests include cloud computing and distributed systems. yuxiang1@andrew.cmu.edu

HDFS clusters rely on a single NameNode, the master, as its metadata service. Single master design of HDFS is known to be a limiting factor for potential growth of the file system in its size and performance. Project Giraffa replaces the single master of HDFS with a dynamically distributed namespace service, thus overcoming scalability limits of HDFS while remaining fully compatible with it. We focus on the performance of namespace operations and present a benchmark that demonstrates that Giraffa can linearly scale the throughput of metadata operation by simply adding more servers to store the file-system namespace.

Apache Hadoop is a system for distributed storage and computation for big data problems. As members of the Hadoop Development team at LinkedIn, it is our daily job to monitor the condition of our clusters, fix problems, and optimize their performance. The most troubling problems are those that result in a cluster-wide crisis.

One day, a user complained that his job was running unusually slowly and not progressing. We thought it could be a problem of the particular job. But with more similar reports coming in, we realized that the cluster became stagnant for most of the jobs assigned to it. Eventually we noticed that the NameNode was unresponsive, running at 100% CPU. Further drilling into HDFS audit logs, we identified one job that was producing hundreds of thousands of namespace operations per second, saturating the NameNode and degrading its performance. The majority of these operations were read requests such as `listStatus`, `getFileInfo`, `getBlockLocations`.

We call the above scenario the "bad client" problem, which means a single "bad" job can make the whole cluster unavailable for everybody. The root cause of this problem is the single master architecture of HDFS, where the performance of a single NameNode, the single master, can constrain the performance of the entire cluster.

Scaling file system metadata along with its data is our primary motivation for building the Giraffa file system. We show that Giraffa metadata operations scale linearly and thus can prevent the bad client problem. See [4] for different aspects of scalability limitations of HDFS architecture [6].

Giraffa Overview

Giraffa [5] is a distributed, highly scalable file system that aims to:

1. Support millions of concurrent clients
2. Store trillions of objects
3. Maintain exabyte total storage capacity

Giraffa is intended to scale both the data storage and its metadata. Giraffa keeps its metadata—directories, files, and blocks—in a distributed key-value store, currently Apache HBase, as a single table distributed across multiple servers, while file data are stored in block

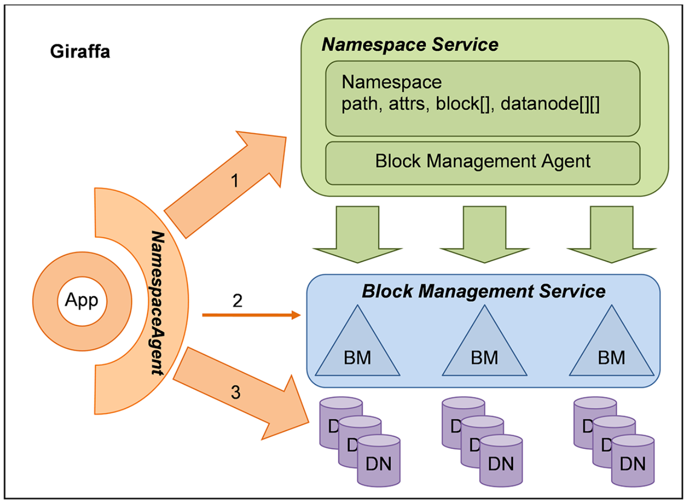


Figure 1: Giraffa Namespace Agent obtains metadata from Giraffa Namespace Service and streams data to or from HDFS DataNodes, while Giraffa Block Manager maintains all blocks.

files located on HDFS DataNodes. In other words, we still store all the data in DataNodes as Hadoop does. However, we save all the information that is stored in the NameNode in Hadoop to an HBase table in Giraffa. This architecture makes Giraffa a drop-in (no data copy) replacement for HDFS. Figure 1 shows the high-level architecture of Giraffa.

In Giraffa the file system metadata is served by the Namespace Service, which is composed of a single HBase table called Namespace. The Namespace table stores records corresponding to files and directories. Each record has a unique key, identifying the file or the directory, and contains the following attributes: local name, owner, group, permissions, access time, modification time, block size, replication, length, and a directory flag. When you need to read a file, you get the file's list of blocks and their locations, so your application can read the data from the respective DataNodes. When you write to a file, Giraffa allocates a block using its BlockManager. The client then writes data to the designated DataNodes.

BlockManager is another service that is used to maintain the flat namespace of blocks. The BlockManager is responsible for:

1. New block allocation
2. Scheduling block replication and deletion
3. DataNode management: process DataNode block reports, heartbeats, detect lost nodes

HBase automatically partitions its tables, and this allows Giraffa to dynamically partition its Namespace. That is, file and directory metadata—table rows—can automatically migrate between nodes based on nodes' utilization and load-balancing requirements. Since metadata is distributed across multiple nodes, this allows the number of files in the file system to increase and ensures that Giraffa is able to deal with trillions of files representing as much as 1000 PB of data on a single cluster.

Row keys identify files and directories as rows in the Namespace table, and they also define the sorting of the rows in the table. Thus, keys play an important role in Namespace partitioning. Row-key definition is based on the locality requirement and is chosen during file-system formatting.

Currently the row key is implemented as a byte array representing the full path to a file in the namespace tree. For example, file `/user/jsmith/job.xml` is identified by the row key, which is a byte representation of the string "`/user/jsmith/job.xml`". Lexicographic ordering of such keys guarantees locality of reference—that is, the children of the same directory fall into the same table partition, a region, most of the time. In the future we plan to define the row keys based on unique immutable INode IDs, which include selfID and two nearest parent IDs. This way, we still guarantee the locality of reference but also allow in-place renames—that is, if a file name changes, it remains in the same region because name changes do not affect row key values.

Giraffa is still in an experimental phase. The problems remaining to be addressed include:

1. Full set of HDFS functionality
2. INode ID-based keys to allow in-place atomic rename
3. Distributed block management
4. Short-circuit HBase metadata into itself
5. HBase scalability: single HMaster, region redundancy

Setting Up a Giraffa Cluster

We've used Giraffa on Java 8 without issues, but it also works with Java 7. We need Gradle 2.5 to build Giraffa sources. Similar to Hadoop, Giraffa uses Google Protocol Buffers version 2.5.0. Giraffa currently depends on hbase-1.0.1 and hadoop-2.5.1.

Although the Giraffa Wiki page on GitHub has instructions for setting up Giraffa in standalone mode, we will show you how to install Giraffa on a real cluster. Our cluster consisted of 11 physical servers (node-001 to node-011). Below are the step-by-step instructions on how to set up the cluster. One may consider writing a batch of scripts to automate the installation process.

Hadoop 2.5.1 Setup

Set up Hadoop normally if you haven't already, following Cluster Setup instructions [1]. HDFS cluster status can be checked via the NameNode Web UI at http://node-001:50070. In our case, node-001 runs the NameNode process, while the other 10 servers node-002–node-011 run DataNodes.

HBase 1.0.1 Setup

1. Follow the official Apache HBase Reference guide [2] to configure and set up HBase cluster.
2. Start HBase. In our cluster, node-001 hosts HMaster and HQuorumPeer processes, and the remaining machines host HRegionServer process. The status of the HBase cluster can be checked on the HMaster Web UI at http://node-001:16010.
3. Stop HDFS and HBase after testing.

Giraffa Setup

1. Download and build Giraffa according to [3].
2. Copy giraffa-standalone-0.4-SNAPSHOT.tgz to all nodes, and change the configuration according to [3].
3. Start and format Giraffa using `giraffa format` command. The script that starts Giraffa will also bring up Hadoop and HBase.

After completing these steps, you should be able to run file system operations on Giraffa. Here are some examples of Giraffa CLI commands.

Get listing of the Giraffa root directory:

```
bin/giraffa fs –ls /
```

Create a new directory:

```
bin/giraffa fs -mkdir testdir
```

YARN Setup

1. Configure YARN according to the official Apache Hadoop tutorial [1].
2. Use Giraffa commands to start YARN daemons: the ResourceManager on node-001, and NodeManager processes on the rest of the nodes:
   ```
   bin/yarn-giraffa-daemon.sh start resourcemanager
   bin/yarn-giraffa-daemon.sh start nodemanager
   ```

The cluster setup is now complete.

TeraSort is an example of a YARN application. By default it starts small MapReduce jobs, which will test the entire setup. Note that in this case all data is stored and processed on the Giraffa file system rather than on HDFS.

1. Run TeraGen:
   ```
   bin/yarn-giraffa jar $HADOOP_HOME/share/hadoop
   /mapreduce/hadoop-mapreduce-examples-2.5.1.jar
   teragen 10000000 /teragen
   ```
2. Run TeraSort:
   ```
   bin/yarn-giraffa jar $HADOOP_HOME/share/hadoop
   /mapreduce/hadoop-mapreduce-examples-2.5.1.jar
   terasort /teragen /terasort
   ```
3. Run TeraValidate:
   ```
   bin/yarn-giraffa jar $HADOOP_HOME/share/hadoop
   /mapreduce/hadoop-mapreduce-examples-2.5.1.jar
   teravalidate /terasort /teravalidate
   ```

The Benchmarks

In order to show that Giraffa scales linearly with the number of region servers, we built a benchmark. In this benchmark, we first create a number of files, and then run a MapReduce job, where each mapper calls `listStatus` for those files.

Suppose we have *m* map tasks running in parallel, and each map task performs `listStatus` for *n* files. Then the result we want to output is `(m * n / t)`, where `t` is the time of the mapping phase. YARN does not guarantee that all tasks start at the same time. In order to synchronize our *m* map tasks running in parallel, we set a start time `t1`. All map tasks will wait until time point `t1` before running the `listStatus` operations. That way we can guarantee that the mappers hit the Namespace Service all at once, providing maximum workload on the service. Finally, we record time `t2` when the last map task stops, and measure the running time for all mappers as `t = t2 – t1`.

This benchmark gives us the number of read operations that Giraffa can handle per second, which is an important metric of the cluster performance.

The configuration of the experiment is as follows:

We set up a cluster with 11 nodes. node-001 hosts master processes: NameNode, HMaster, ResourceManager. node-002–node-011 host the slave processes: DataNode, HRegion, NodeManager. We managed to run 220 map tasks simultaneously on our cluster, and required each of them to perform listStatus for 10,000 files. We collected the running time and repeated this experiment several times to get rid of the soft bias.

We chose the number of map tasks to run (220) based on the capacity of the cluster. YARN as a resource manager allocates containers, each of which runs a single task and defines how much of execution resources, RAM and CPU (vCores), to be dedicated to a specific task. Thus, the cluster capacity is determined by the total amount of RAM and the total number of vCores. Our goal was to fully utilize the cluster without overutilizing it, so that all mappers ran simultaneously rather than in "waves."

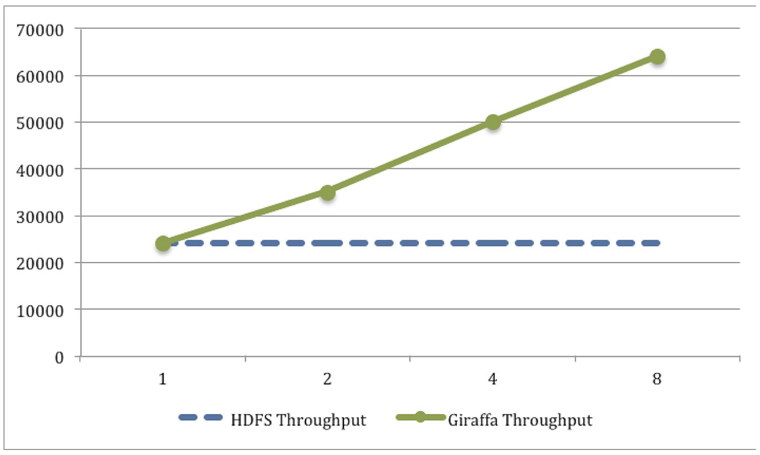


Figure 2: Giraffa read performance scales linearly with number of servers compared to the single NameNode.

In our cluster, we had a total of 220 GB of RAM and 320 vCores available for containers. Each task requires at least 1 GB of memory and 1 vCore. We therefore decided to set the number of map tasks to be 220, which satisfies the single wave requirement without affecting the performance of the cluster.

We started the Giraffa benchmark with a single region server serving the entire Namespace table. Then we used the HBase `split` command to dynamically partition the table into two regions served by two different region servers. *Dynamically* here means that we did not need to copy file data or restart the cluster for repartitioning. Then we similarly split the table into four and eight regions and made sure that each of them was assigned to a different region server.

In order to compare the performance of Giraffa and HDFS, we ran the same benchmark on an HDFS cluster using the same hardware. The main difference is that the Hadoop cluster does not need HMaster and HRegion processes. We stopped the Giraffa cluster, set up HDFS, and configured and started YARN with HDFS according to [1].

For Hadoop we also ran 220 parallel mappers with each of them performing `listStatus` for 10,000 files. Figure 2 shows the benchmark results.

The x-axis represents the number of region servers serving Giraffa namespace, and the y-axis represents the number of read operations per second that the file system processed. Since in our HDFS cluster we had only one NameNode, the number of read operations per second does not change, and the dashed line serves as the baseline. The solid line represents the throughput of Giraffa. It shows linear growth of read operations per second with the number of region servers. The benchmark is limited to eight region servers because of the cluster size limitations.

From these tests, we can see that the read performance of Giraffa scales linearly with the number of region servers. The write performance was partly addressed in [7]. It shows that the `mkdir` operation scales linearly. We expect that some operations like file `create` or `delete` will scale linearly as well, but some like `addBlock` will not due to limitations of the current Giraffa implementation, something yet to be fixed.

Conclusion

We showed that the Giraffa file system could linearly scale metadata operation for read requests by simply adding more servers to store the file-system namespace.

Authors of [7] came to the same conclusion as they benchmarked Giraffa along with two other systems, ShardFS and IndexFS, on a variety of metadata workloads. It shows that Giraffa scales linearly in throughput as more servers are dynamically added to the system for most of the workloads.

References

[1] Apache Hadoop Cluster Setup: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/ClusterSetup.html.

[2] Apache HBase Configuration: https://hbase.apache.org/0.94/book/configuration.html.

[3] How to Set up, Build, and Use Giraffa: https://github.com/GiraffaFS/giraffa/wiki/How-to-Setup,-Build,-and-Use-Giraffa.

[4] K. V. Shvachko, "HDFS Scalability: The Limits to Growth," *;login:*, vol. 35, no. 2 (April 2010): https://www.usenix.org/legacy/publications/login/2010-04/openpdfs/shvachko.pdf.

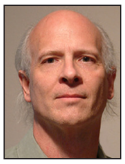
[5] K. V. Shvachko, P. Jeliazkov, "Dynamic Namespace Partitioning with Giraffa File System," Hadoop Summit 2012: http://lanyrd.com/2012/hadoop-summit/stttw/.

[6] K. Shvachko, H. Kuang, S. Radia, R. Chansler, "The Hadoop Distributed File System," 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 2010.

[7] L. Xiao, K. Ren, Q. Zheng, G. A. Gibson, "ShardFS vs. IndexFS: Replication vs. Caching Strategies for Distributed Metadata Management in Cloud Storage Systems," Sixth ACM Symposium on Cloud Computing, 2015.

2017 USENIX Research in Linux File and Storage Technologies Summit (Linux FAST Summit '17)

RIK FARROW



Rik is the editor of *;login:*.
rik@usenix.org

Editor's Note: This report includes some summaries from Mai Zheng (zheng@nmsu.edu) and Om Rameshwar Gatla (omram@nmsu.edu)

Ric Wheeler (Red Hat) chaired the Linux FAST Summit '17. There were 50 attendees, the most yet, with 60% from large companies, 20% from universities, and the rest consultants or from smaller companies. According to Ric, 33% of the Linux FAST attendees did not attend FAST '17.

After introducing ourselves and briefly explaining why we were attending, discussion of issues with block I/O began. Someone mentioned that the latest Linux kernels can handle as many as 40 million IOPS. Ted Ts'o (Google) suggested that it's time to start considering techniques used in high-speed networking to further improve performance.

Erez Zadok (Stony Brook University) wondered how multiple write queues to the same device affected order handling. Christoph Hellwig (consultant and Linux file system hacker) said ordering isn't handled; it's an unsolved problem. Most devices behave as if they are non-volatile, returning completion codes while data is still buffered in on-device RAM. And devices perform out-of-order writes as they see fit. That pretty much guarantees that anything done by an OS, such as write barriers, can't work.

Andrew Morton (Google) then began the "how to work with the Linux kernel" section, a tradition at Linux FAST. Andrew suggested sending him your first patch (for file system patches) rather than just posting your patch to the Linux-kernel list. Andrew pointed out that the kernel developers had gotten a bad reputation for being harsh, but now "we're pretty professional."

Ted Ts'o put this another way. Suppose someone unknown to the developers sends an email, which is like cold calling. You want to work through introductions if at all possible, just as you would in any social situation, and it's also important to use the most recent kernel possible. You can get the most recent build at kernel.org, but if you are working with a specialist in some area, ask that person which build to work with. In general, choosing a stable release means you will be working with a kernel that will be supported for some time.

Ted also mentioned that he has created some regression testing tools for file systems. You can find these tools at https://github.com/tytso/xfstests. Ted, who co-authored the FAST '17 paper "Evolving Ext4 for Shingled Disks" (in this issue), tried the patches written for improving SMR performances against his regression testing tools. The patches failed, although they were good enough to run the benchmarks used to write the paper. Those patches will eventually be cleaned up and merged into the upstream kernel.

George Amvrosiadis (student at Carnegie Mellon University) mentioned having three thousand lines of code that he shared with members of the file system group. He said he got lots of feedback and started to develop a relationship with this group of kernel hackers. He also wanted a particular tracepoint added to the kernel and hasn't succeeded yet. But he wasn't discouraged by the process.

Ric then shifted the focus to FUSE by asking Sage Weil (Red Hat, key author of Ceph) about his experience working with FUSE. Sage said that although writing user-space software is easier, you still run into kernel issues. For example, you don't control the page cache or writeback queue.

Erez mentioned a paper he co-authored for FAST '17 (Vangoor et al., "To FUSE or Not to FUSE: Performance of User-Space File Systems"), where they played with lots of switches in FUSE to see how those affected performance. He was surprised there was so little documentation for FUSE. George mentioned that the patch he wanted was a tracepoint that would let them know when metadata had been modified. Sage pointed out that with FUSE, the kernel is still doing a lot of work "under the hood" and that FUSE performance has gotten a lot faster over time.

Another person from Red Hat mentioned that one big advantage with using FUSE is that you can run your file system without having to patch a certified kernel. Jeff Darcy (Red Hat) agreed and added that trying to run non-standard kernels in the cloud was a non-starter.

John Grove (Micron) said his group was developing a new file system and that being able to work in FUSE for prototyping was a great help.

The next topic covered had to do with writing "dirty" buffers back to disk. Jonathan Amit (IBM Israel) has a problem with a project that allows customers to write many gigabytes, using multiple threads. But there is just one kernel thread serving the write-back cache, and to get the best performance they just bypass the page cache. Ted answered that using O_DIRECT is the way people who are passionate about performance handle this problem. Jonathan said it was not always easy to use O_DIRECT, and Ted agreed.

Mai Zheng (New Mexico State University) mentioned two cases where bugs in the Linux kernel affected devices' behavior. In one case he tested dozens of SSDs under power faults, and many devices exhibited corruptions in the tests (see "Understanding the Robustness of SSDs under Power Fault" presented at FAST '13). However, after several years, the same tests were performed using a newer kernel. It turns out that a bug patch (by Christoph Hellwig) changes the corruptions observed on some devices (published in 2016 in *ACM Transactions on Computer Systems*). In another case that happened at Algolia datacenter, Samsung's SSDs were blamed for data corruption initially. However, Samsung's engineers eventually found that it was a kernel bug that caused the trouble (http://www.spinics.net/lists/raid/msg49440.html); the bug was patched by Martin K. Petersen.

Ted commented that only enterprise-class SSDs can be relied upon (at all) for safe behavior on power fail. The enterprise-class SSDs have super-capacitors that store enough power to write all data stored in the RAM within the SSD on power fail, and vendors charge three times as much as they do for consumer class SSDs. Some vendors do certify their SSDs, but you should check them under real power-fail conditions, like pulling the plug. Peter Desnoyers (Northeastern University) suggested using an Arduino with a relay for experimenting with cutting off power.

Jonathan then changed the topic to ask about NVME device performance. Christoph replied that he had rewritten that device to make it simpler: no waiting, no polling, and this should be in the 4.9 kernel.

Om Rameshwar Gatla (New Mexico State University) raised a question regarding how robust the local and large-scale file system checkers are besides e2fsck. Christoph replied that even the XFS repair utility is as vulnerable to faults as e2fsck is, and this could be the same with the repair utility of B-tree file system (btrfs). In regards to the robustness of checkers for large-scale file systems, developers of Ceph said that their file system includes many fault-handling techniques such as journaling, data replication, etc. by which this situation may be mitigated.

Ric Wheeler commented that many repair utilities, such as XFS repair, consume a lot of memory and that this problem could serve as a good research topic. The other topic discussed regarding `fsck` was its running time. Ric suggested running all file system checkers of an aging, fragmented file system on a hard disk whose sizes are on the magnitude of terabytes and observe the memory consumption and total run times. The results from these experiments may provide a good research opportunity. Ted added that the problem that e2fsck's slowness is because EXT file systems maintain lots of bitmaps to track information on all the inodes, direct and indirect blocks, etc., but the overall memory consumption of e2fsck is far less than any other file system checker. To support his argument, Ted gave an example where they ran e2fsck on a 6 TB hard disk that was 80% full and had the Hadoop layout. e2fsck consumed less than 9 MB of memory to complete. Ted added that having a large number of hard links creates the greatest challenge for `fsck`.

Niels De Vos (Red Hat) mentioned that GlusterFS uses extended attributes (xattrs) in ext4, and if users edit the attributes, you really get into big trouble. Of course, there's no way that an fsck could check for that. They also do erasure coding for files, which means that checking involves reading files on multiple servers.

Om also asked about the error reporting mechanism from file systems or lower layers. He wanted to know more details when facing some errors (e.g., why a volume is reported "unmountable"). Ted, Christoph, Ric, and some others commented that the current mechanism relies on error numbers (errno). The overhead of passing more detailed information around might be high. Also, `dmesg` is a good place to look for more detailed error messaging in current systems.

There was some discussion about mapping and providing low-level block information to higher level software. Ted commented that debugfs (https://www.kernel.org/doc/Documentation/filesystems/debugfs.txt) provides such a mechanism. Mai commented that in his project about analyzing the bugs in databases and file systems, debugfs has helped a lot for examining the relationship between the corruption at low-level I/O blocks and the impact on database logs.

Jonathan asked about why mmaping two terabytes of memory takes so long. Andrew pointed out that populating two terabytes working with four-kilobyte pages was always going to take a long time, leading Jonathan to wonder whether the Persistent Memory (PMEM) driver supported huge pages.

Pankaj Mehra (Western Digital) said that people so far don't understand PMEM, as they are not using `mmap` (see Andy Rudoff's article "Persistent Memory Programming" in this issue). Ted agreed: you don't want a POSIX layer, you want to `mmap` PMEM into your process memory. You can treat PMEM as superflash, but there's lots of overhead there.

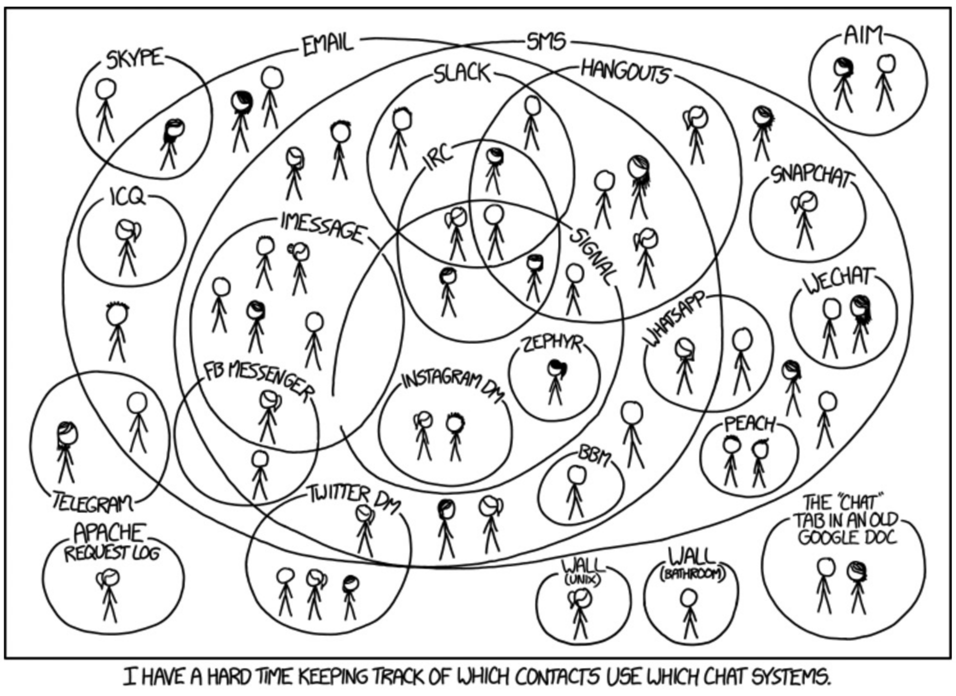
Pankaj replied that if you have PMEM, you are going to want to manage it, which includes encryption, snapshots, naming, permissions, and free space. Sam Fineberg (Consultant) pointed out that the traditional way of dealing with memory errors in Linux is to use ECC or to crash. Ric mentioned that the Micron-Intel XPoint PMEM will be able to report bad memory. Mai mentioned a paper published in EuroSys '13 which makes the `msync()` system call robust ("Failure-Atomic `msync()`: A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data"). Christoph confirmed that the idea as well as the findings in a follow-up paper from the same group have been incorporated into the Linux kernel.

Pankaj continued: "When we first came up with the term PMEM, we were very careful. The way we handled this is the way Rudoff describes it: one instruction per address. When you do a store, we will store. If you want PMEM to do transactions, you lose the performance benefits."

In the (near) final topic of the day, Ted said that he is currently working on data encryption at the file system level and that there are many challenges to it, such as how to provision crypto keys for encryption and decryption, and where to store them securely. Ted also said that the efficiency is highly architecture-dependent, with Intel Skylake able to encrypt one word per cycle, but ARM CPUs having no native support.

The final topic concerned tuning the page cache, and Ric pointed out that there is a tool called `tuned` that helps with picking appropriate sets of tuning for storage, and that you can actually find tuned profiles for different use cases.

XKCD **xkcd.com**



Persistent Memory Programming

ANDY RUDOFF

Andy Rudoff is a Senior Principal Engineer at Intel Corporation, focusing on non-volatile memory programming. He is a contributor to the SNIA NVM Programming Technical Work Group. His more than 30 years' industry experience includes design and development work in operating systems, file systems, networking, and fault management at companies large and small, including Sun Microsystems and VMware. Andy has taught various operating systems classes over the years and is a co-author of the popular *UNIX Network Programming* textbook. andy.rudoff@intel.com

In the June 2013 issue of *;login:,* I wrote about future interfaces for non-volatile memory (NVM) [1]. In it, I described an NVM programming model specification [2] under development in the SNIA NVM Programming Technical Work Group (TWG). In the four years that have passed, the spec has been published, and, as predicted, one of the programming models contained in the spec has become the focus of considerable follow-up work. That programming model, described in the spec as NVM.PM.FILE, states that persistent memory (PM) should be exposed by operating systems as memory-mapped files. In this article, I'll describe how the intended persistent memory programming model turned out in actual OS implementations, what work has been done to build on it, and what challenges are still ahead of us.

The Essential Background on Persistent Memory

The terms *persistent memory* and *storage class memory* are synonymous, describing media with byte-addressable, load/store memory access, but with the persistence properties of storage. In this article, I will focus on persistent memory connected to the system memory bus, like a DRAM DIMM, creating a class of non-volatile DIMMs known as NVDIMMs.

To further clarify what I mean by persistent memory, I am only speaking about NVDIMMs that allow software to access the media as memory (some NVDIMMs only support block access and are not covered here). This provides all the benefits of memory semantics, like CPU cache coherency, direct memory access (DMA) by other devices, and cache line granularity access which programmers can treat as byte-addressability. To provide these semantics, the media must be fast enough that it is reasonable to stall a CPU while an instruction is accessing it. NAND Flash, for example, is too slow to be considered persistent memory by itself, since access is typically done in block granularity and it takes long enough that context switching to allow another thread to do work makes more sense than stalling. Where hard drive accesses are typically measured in milliseconds, and NAND Flash SSD accesses are measured in microseconds, persistent memory accesses are measured in nanoseconds. Depending on the exact type of media, an NVDIMM may not be as fast as DRAM, but it is in the neighborhood.

Some NVDIMM products on the market today use DRAM as the media at runtime but automatically back up the contents to NAND Flash on power loss and restore the contents when the power returns. These products provide DRAM performance but also require additional components and an energy source to save the data, giving them a lower per-DIMM capacity and higher cost per gigabyte than DRAM. Emerging non-volatile media, like the 3D XPoint technology announced jointly by Intel and Micron in 2015, promises higher capacity at a price point lower than DRAM. Multiple terabytes per CPU socket are expected, making persistent memory interesting on multiple fronts: persistence, capacity, and cost [3].

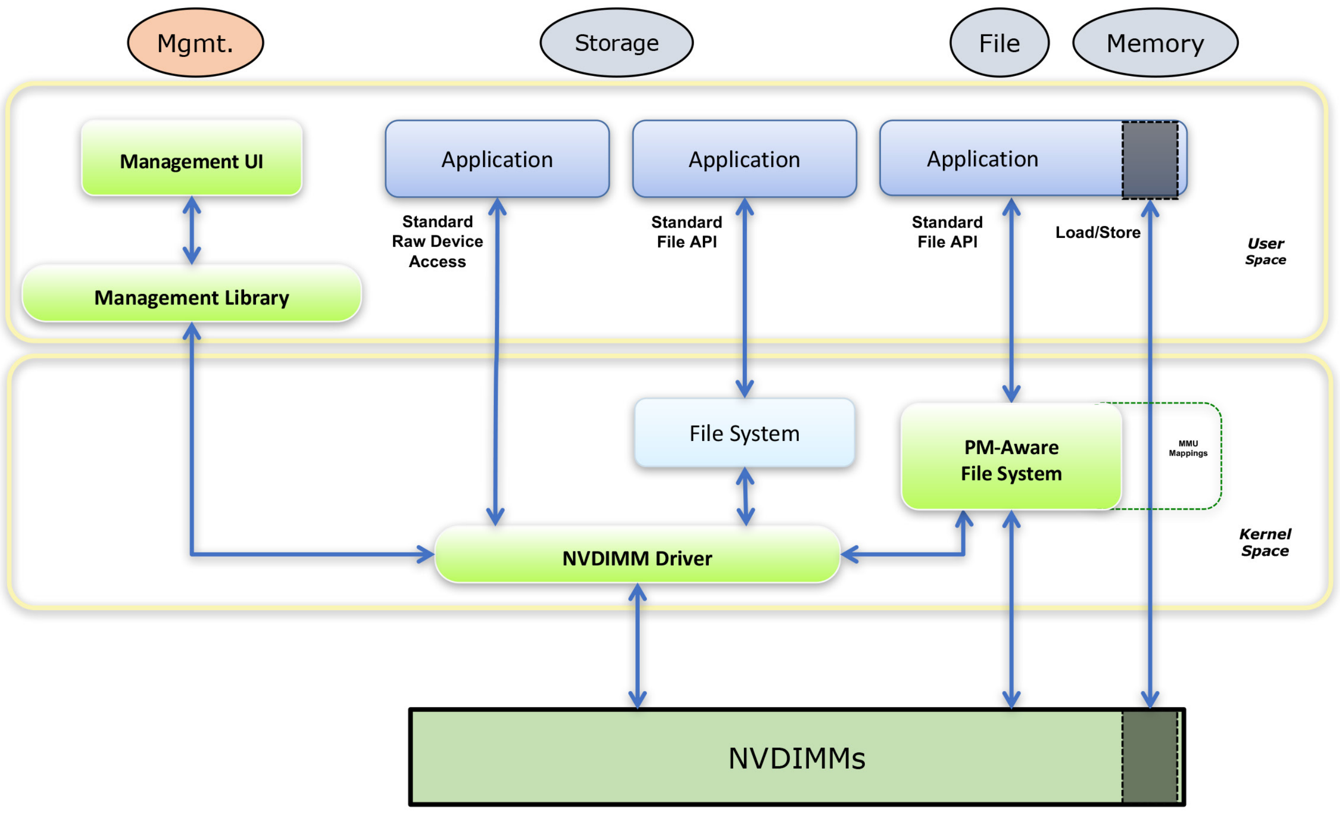


Figure 1: The SNIA persistent memory programming model

The Persistent Memory Programming Model

How does an application get access to persistent memory? Unlike volatile memory, the application needs a way to connect with specific persistent contents; persistent memory isn't anonymous like volatile memory; regions need names so applications can find them, just like files. And also like files, regions of persistent memory need permissions to control which applications have access to the persistent information. The entire point of the persistent memory programming model specified by the SNIA TWG was to recommend that operating systems use standard file semantics to provide naming, permissions, and memory-mapping of persistent memory.

Now that this has been implemented in multiple operating systems, including Linux and Windows, it seems very obvious, and you might wonder why a specification was even necessary. But four years ago when I wrote the first *;login:* article, there were multiple competing ideas on how to expose persistent memory, and software vendors were in danger of having to decide between incompatible programming models from different products. Instead, the ecosystem has unified nicely around the model shown in Figure 1.

The NVDIMM shown at the bottom of the figure represents the persistent memory installed in the system, potentially spread across many NVDIMMs, and potentially interleaved (striped) for performance by the memory controller. On Intel-based systems, the BIOS creates a table called the NVDIMM Firmware Interface Table (NFIT) that enumerates the NVDIMMs installed. This table was added to the ACPI specification in version 6.0 and continues to evolve as NVDIMMs evolve. As shown in the figure, some driver (or collection of drivers) consumes the NFIT information and takes ownership of the persistent memory, exposing it to management software (left side of the figure), potentially exposing it as traditional block storage which is emulated by the driver (middle part of the figure), and exposing it directly to applications through a *persistent memory aware file system* (the right side of the figure).

DAX

My definition of a *persistent memory aware file system*, like the one shown in Figure 1, is a file system that allows direct access to persistent memory without using the system page cache as it would for normal, storage-based files. This feature has been named DAX by the operating systems folks, short for *Direct Access*. Conveniently, both Linux and Windows use the same term for the same feature.

The persistent memory programming model, and the corresponding DAX feature, says persistent memory files can be mapped into memory using standard calls like `mmap()` on Linux or `MapViewOfFile()` on Windows. This results in the far-right arrow on Figure 1, where the application has direct load/store access to the persistence. Once these mappings are set up (and after any initial minor page faults that may be required to create the mappings in the MMU), this provides the shortest possible code path to persistence, allowing the applications to perform loads and stores on the persistent media directly with no kernel involvement. No interrupts, no context switching, no kernel code at all is required for media access.

Making Stores Persistent

Just as persistent memory is accessed using standard memory-mapped files, the steps for making changes persistent follow the same standards. On Linux (actually any POSIX-compliant system), the range-based `msync()` call or file-based `fsync()` call may be used to ensure changes are persistent. On Windows, the combination of `FlushViewOfFile()` and `FlushFileBuffers()` is used. These calls create a *store barrier*, a point after which the program can assume the previous changes it made to the persistent memory are actually persistent. Historically, this store barrier required the operating system to find dirty pages in the system page cache, flushing them to block storage, such as a disk. But since persistent memory doesn't use the page cache, the operating system need only flush the CPU caches, as appropriate, to get changes into the *persistence domain*. I define the persistence domain as the point along the data path taken by stores where they are considered persistent because that point is *power fail safe* (see Figure 2).

The dashed box in Figure 2 shows the persistence domain required by Intel platforms supporting persistent memory. At the platform level, any stores inside the dashed box are either on the DIMM, or still in the *write pending queue* (WPQ) in the memory controller, on their way to the DIMM. Either way, platforms supporting persistent memory are required to have enough stored energy to flush any stores inside the dashed box all the way to persistent media on power loss. This feature, flushing the stores the rest of the way on power failure, is known as *asynchronous DRAM refresh* (ADR) and has been a requirement of NVDIMM products since they first appeared a few years ago.

At the x86 instruction level, simply executing a store instruction is not enough to make data persistent, since the data may be sitting in the CPU caches indefinitely and could be lost by a power failure. Additional cache flush actions are required to make the stores persistent. The following table describes how each of these works.

Looking at Figure 2 and the instructions in the Table 1 might make you wonder why Intel didn't just make the CPU caches part

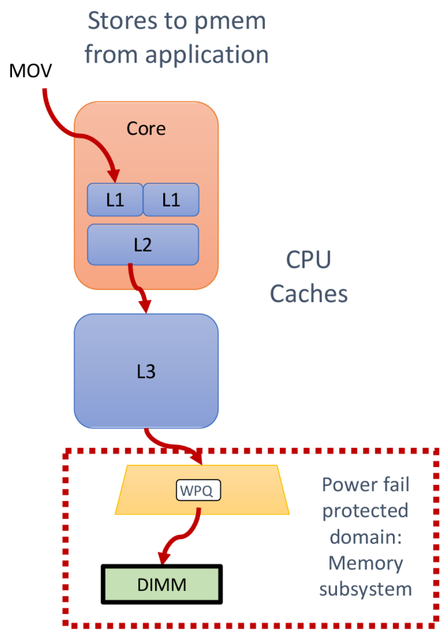

Stores to pmem from application
MOV
Core
L1
L1
L2
CPU Caches
L3
WPQ
DIMM
Power fail protected domain: Memory subsystem

Figure 2: The path taken by a store, and the persistence domain (dashed box)

of the persistence domain. This is technically possible, producing the situation shown in Figure 2 but with the dashed box now including the CPU caches.

The problem with extending the persistence domain to include the CPU caches is that the x86 caches are quite large, and it would take more energy than the capacitors in a power supply can practically provide. This usually means the platform would have to contain battery. Requiring a battery for every server supporting persistent memory is not practical at this time, but it is certainly possible for companies, such as appliance vendors who ship custom hardware, to include a battery in their product. This would allow the cache flush instructions described in Table 1 to be skipped, but the SFENCE instruction would still be required as a store barrier—stores should be considered persistent only when they are globally visible, and that's what the SFENCE ensures.

Because some appliance vendors plan to use batteries as I've described, and because I hope that all platforms will someday include the CPU caches in the persistence domain, a property is being added to ACPI so that the BIOS can notify the operating system when the CPU flushes can be skipped. This allows the operating system to implement calls like `msync()` in the most optimal way.

User Space Flushing to Persistence

With the exception of WBINVD, the instructions I described in Table 1 are supported in user mode by Intel CPUs. Flushing a cache line using CLWB (or CLFLUSHOPT or CLFLUSH) and using non-temporal stores are all supported from user space.

CLFLUSH	This instruction, supported in many generations of CPU, flushes a single cache line. Historically, this instruction is serialized, causing multiple CLFLUSH instructions to execute one after the other, without any concurrency.
CLFLUSHOPT (followed by an SFENCE)	This instruction, newly introduced for persistent memory support, is like CLFLUSH but without the serialization. To flush a range, software executes a CLFLUSHOPT instruction for each 64-byte cache line in the range, followed by a single SFENCE instruction to ensure the flushes are complete before continuing. CLFLUSHOPT is optimized (hence the name) to allow some concurrency when executing multiple CLFLUSHOPT instructions back-to-back.
CLWB (followed by an SFENCE)	Another newly introduced instruction, CLWB stands for *cache line write back*. The effect is the same as CLFLUSHOPT except that the cache line may remain valid in the cache (but no longer dirty, since it was flushed). This makes it more likely to get a cache hit on this line as the data is accessed again later.
NT stores (followed by an SFENCE)	Another feature that has been around for a while in x86 CPUs is the non-temporal store. These stores are "write combining" and bypass the CPU cache, so using them does not require a flush. The final SFENCE instruction is still required to ensure the stores have reached the persistence domain.
WBINVD	This kernel-mode-only instruction flushes and invalidates every cache line on the CPU that executes it. After executing this on all CPUs, all stores to persistent memory are certainly in the persistence domain, but all cache lines are empty, impacting performance. In addition, the overhead of sending a message to each CPU to execute this instruction can be significant. Because of this, WBINVD is only expected to be used by the kernel for flushing very large ranges, many megabytes at least.

Table 1: x86 cache flush instructions for use with persistent memory

This could allow the flushing to persistence directly from user space, without calling into the kernel, a feature documented in the SNIA programming model spec as *Optimized Flush*. The spec describes Optimized Flush as optionally supported by the platform, depending on the hardware and operating system support. Despite the CPU support, it is important for applications to only use Optimized Flush when the operating system says it is safe to use. The operating system may require the control point provided by calls like `msync()` when, for example, there are changes to file system metadata that need to be written as part of the `msync()` operation.

Support for safe userspace flushing is an evolving feature in the current implementations. At the time of this writing, the DAX support in Windows, provided by the NTFS file system, includes unconditional support for Optimized Flush. Windows programs can ensure stores to persistent memory are persistent using instruction sequences like CLWB + SFENCE. On Linux, the two file systems that support DAX, ext4 and XFS, do not currently consider userspace flushing safe. While hoping to work out interfaces with these file systems that tell applications when Optimized Flush is safe, it is an ongoing discussion. Other file systems, like NOVA [4], a research project from UCSD, are designed from the start to support Optimized Flush but are not considered production ready yet. As an interim solution, Linux provides Device-DAX [5], which allows an application to open a persistent memory device (without a file system), memory map it, and utilize userspace flushes to make stores persistent.

To insulate application programmers from this complexity, and to keep them from having to research the current state of affairs while programming for persistent memory, the `libpmem` library provides a function which tells the application when Optimized Flush is safe. Programmers are strongly encouraged to use libpmem to make this determination and to use userspace flushing only when it is safe, falling back on the standard method of flushing stores to memory mapped files otherwise. The libpmem library is also designed to detect the case of the platform with a battery I described above, turning flush calls into simple SFENCE instructions instead. I've got much more to say about libraries below, and all the libraries I describe build on this logic to make sure they transparently depend on the most optimal type of flushing available to the program.

Persistent Memory Challenges

When a modern program changes any data structure in memory, the question of *atomicity* comes up. Is it possible for another thread to access the data structure and see the change only partially complete? With multithreaded programming, this issue is commonly solved using locks to protect data structures. Sometimes it is solved by using instruction sequences that guarantee atomicity in hardware. These issues have been around for years and are very familiar to programmers, library writers, and high-level language designers. In this context, the term *atomicity* really refers to *visibility*, protecting the changes made by one thread from becoming visible by other threads until the changes are complete. Adding persistent memory into this picture, the requirements change from simple atomicity to something more

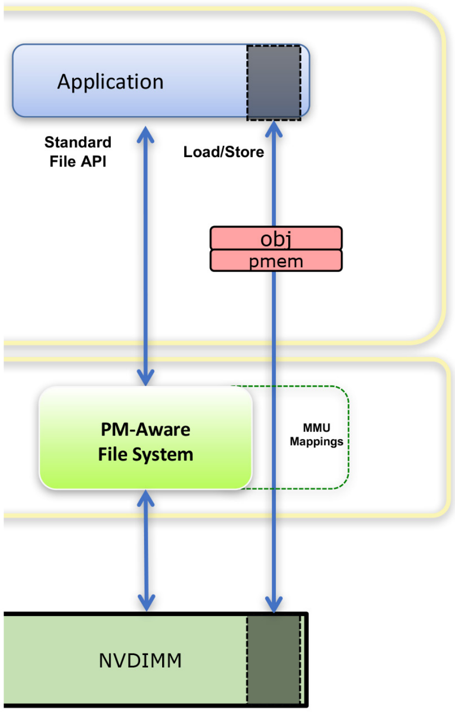


Figure 3: Using the libpmemobj library, which in turn uses the primitives in libpmem

like the ACID semantics required for database transactions on storage [6]. Not only do we want to keep other threads from seeing an incomplete change, we want to handle changes that are interrupted by power failures, program crashes, or exceptions. Everyone who starts writing programs to use persistent memory seems to immediately come to this conclusion: we need transactions that are power fail safe.

Before persistent memory existed, if a store was interrupted by something like power failure, the resulting memory state didn't matter much because it was volatile. But with persistent memory, it is important to understand what is guaranteed by hardware and what is left to software. On Intel, only an eight-byte store, aligned on an eight-byte boundary, is guaranteed to be failure atomic. That means if the store is interrupted by a power failure, the memory contents will contain the previous eight bytes, or the new eight bytes, but not some combination of the old and new data. Anything larger than eight bytes can be torn by power failure and must be handled by software. For example, if you want to update two eight-byte pointers in your program, and you want it to happen atomically, protecting those pointers with a lock will only help you prevent other running threads from seeing the partial update. A power failure might leave the update partially done, and there's no single instruction that will solve that—software must arrange for the update to be transactional by building on the eight-byte power-fail-atomic store provided by hardware. The logic for creating these transactions is a bit tricky, which points to the need for libraries or language features to provide them.

Another persistent memory challenge is more basic: managing the space. Since persistent memory regions are exposed as files, the file system primarily manages that space. But once the file is memory-mapped by an application, what happens within that file is completely up to the application. Functions like C's `malloc()` assume memory is volatile, offering no way on program start-up to reconnect with a persistent heap and taking no steps to make sure the heap is consistent in the face of failure. This adds space allocation to our list of requirements for persistent memory programming.

The need for location-independence is another challenge. Although it is technically possible to require that a range of persistent memory is always mapped at exactly the same address in a program, it can become impractical when the sizes of other mapped items change. A security feature known as *Address Space Layout Randomization* (ASLR) additionally causes operating systems to randomly adjust where libraries and files are mapped. Location-independence means that when one data structure in persistent memory refers to another using a pointer, that pointer must be somehow usable even when the file is mapped at a different address. There are several ways to achieve this, such as relocating pointers after mapping, using relative pointers instead of absolute pointers, or by using some type of *Object ID* (OID) to refer to persistent memory-resident data structures.

The NVM Libraries

The libraries produced by my team at Intel are designed to solve the challenges described above. They are meant as a convenience, not as a requirement for persistent memory programmers. Although I refer to them collectively by the single name NVML, they are really a suite of six libraries (with additional libraries already under development). The libraries are all open source, BSD-licensed, and developed in the open on GitHub. I'll describe the libraries here, but much more information is available at http://pmem.io, including man pages, blog entries, and lots of example code.

The libraries are written in C and are validated and ready for use on 64-bit Linux and Windows systems. Some Linux distros already contain the libraries in their repositories, allowing them to be installed with simple package management commands. Otherwise, you can clone the GitHub tree and use `make install` to install the library from source (details are on the Web site [7]).

Since these are C libraries, it is possible to call them from various languages. When using C, we provide some macros to try to help catch common persistent memory programming errors, but C macros are never a replacement for full language integration. The C++ support recently released in libpmemobj (http://pmem.io/nvml/libpmemobj/; see below) is the cleanest, least error-prone way we have to do persistent memory programming. For this reason, if you're just beginning to explore persistent memory programming, the C++ examples are the best place to start.

Here's an overview of the suite of libraries in NVML. Many examples are available in the examples directory of the source on GitHub, but to save space I will limit my examples to the most commonly used library, libpmemobj.

libpmem: Basic Persistence Support

The libpmem library is small and fairly simple, containing the code that detects which types of flush instructions are supported by the CPU, as well as performance-tuned routines for copying ranges of persistence memory using the best instruction choices for the platform. As mentioned above, a routine that tells the caller whether Optimized Flush is safe is supplied (this routine is called `is_pmem()` for historical reasons—perhaps `optimized_flush_available()` would have been a better name in hindsight).

Even if you decide not to use any of the libraries I describe below, you might still decide to use libpmem (or steal the code) just to avoid the tedious development of code that detects supported instructions, the correct use of non-temporal stores, etc.

libpmemobj: General-Purpose Allocations and Transactions

This is probably the library you want. As you might guess, the "obj" in the name is short for `object`, but by that I mean the variable-sized blob of data referred to by the term *object storage,* not the class with methods in an object-oriented language. Figure 3 shows where this library sits in the programming model. Like all the persistent libraries in the NVML suite, this library builds on the primitives provided by libpmem.

The libpmemobj library allows persistent memory *objects* to be allocated in a way that is power fail safe, allows referring to them by Object IDs (OIDs), which are location-independent, and allows making an arbitrary number of changes atomic by encompassing the changes in a transaction. The library is multithread safe and optimized for multithread scalability (by doing things like maintaining per-thread allocation caches).

As mentioned above, the C++ support in this library provides the cleanest, easiest-to-use interfaces, so I'll use a C++ example. The classic persistent memory example is to link something into a linked list (a *queue* in this example, taken verbatim from the `queue.cpp` example in the NVML examples area), where multiple operations are required to be done as a transaction. The example code below starts by creating a class which defines the struct `pmem_entry`, the entries on the queue:

```
class pmem_queue {

        /* entry in the list */
        struct pmem_entry {
                persistent_ptr<pmem_entry> next;
                p<uint64_t> value;
        };
        /* … */
```

Notice the `persistent_ptr` smart pointer template. This indicates a pointer to an object in persistent memory, namely the next item in the persistent queue. These are the location-independent OIDs I mentioned earlier. Also notice the `p<>` persistent property in the above declaration, used to indicate fields that reside in persistent memory. The result of these C++ declarations is that the code to atomically allocate a new entry, initialize it, and link it into the queue can be done as follows:

```
/*
 * Inserts a new element at the end of the queue.
 */
void
push(pool_base &pop, uint64_t value)
{
      transaction::exec_tx(pop, [&] {
            auto n = make_persistent<pmem_entry>();

            n->value = value;
            n->next = nullptr;

            if (head == nullptr) {
                  head = tail = n;
            } else {
                  tail->next = n;
                  tail = n;
            }
      });
}
```

The above push operation is transactional. More specifically, the code in the C++ *lambda,* indicated by `[&] {…}`, is transactional, meaning if the program or the machine crashes during the execution of that code, libpmemobj automatically rolls any partially done changes back (this includes the allocation done by the `make_persistent` call).

There are many more details available for this example, as well as others, on the pmem.io Web site. The main point of the short example above is to show that, with no compiler or language changes, libpmemobj provides a flexible allocation and transaction mechanism for persistent memory.

libpmemblk and libpmemlog: Support for Specific Use Cases

In addition to libpmemobj and its flexible transaction support, two other libraries target specific use cases. The library libpmemblk is written specifically to maintain a large array of persistent memory blocks, all the same size. This is useful, for example, when an application is managing a block cache. The block size provided by the library is flexible, supporting blocks 512-bytes and larger.

Similarly, the library libpmemlog is written for a specific use case where the application frequently appends to a private log file, one that is read rarely, like during crash recovery. This library takes the relatively long file system append path through the kernel and turns it into a very short memory copy in persistent memory, followed by an atomic pointer adjustment.

Both of these specific use cases are easily solved using the more flexible libpmemobj, but the point of libpmemblk and libpmemlog is they provide APIs that constrain the caller, allowing the library to assume specific cases and optimize for them.

libmemkind: The Volatile Use of Persistent Memory

With the large capacity and cheaper-than-DRAM price points expected for emerging persistent memory products, many volatile use cases have come up. These are cases where the application places some data structures in persistent memory to avoid a large DRAM footprint, but the application doesn't really care that the memory is persistent—it is just using it as a second tier of volatile memory. When NVML was first developed, we created a library called libvmem ("vmem" for *volatile memory*). Since then, another more general library for volatile use cases has been open sourced on GitHub [8]. Some projects have already been written to our libvmem interfaces, but for all future development of volatile use cases, we recommend using libmemkind.

Conclusion

The ideas I outlined in 2013 have come true and have matured into a fairly complete programming model, resulting at the operating system level in the DAX feature for both Windows and Linux (and potentially other operating systems beyond the scope of this article). Next, libraries have been built on that basic model to provide application developers with a menu of APIs to choose from as they leverage the benefits persistent memory has to offer. There's still a long list of interesting and fruitful work to be done, integrating persistent memory support into additional languages and libraries (see our GitHub area at https://github.com/pmem for numerous works-in-progress in this space).

References

[1] A. Rudoff, "Programming Models for Emerging Non-Volatile Memory Technologies," *;login:*, vol. 38, no. 3 (June 2013): https://www.usenix.org/system/files/login/articles/08_rudoff_040-045_final.pdf.

[2] "SNIA NVM Programming Technical Work Group": http://www.snia.org/forums/sssi/nvmp.

[3] "3D XPoint™ Technology Revolutionizes Storage Memory": https://www.youtube.com/watch?v=Wgk4U4qVpNY.

[4] J. Xu and S. Swanson, "NOVA: A Log-Structured File System for Hybrid Volatile/Non-Volatile Main Memories," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*: https://www.usenix.org/system/files/conference/fast16/fast16-papers-xu.pdf.

[5] Dan Williams, "Device-DAX": https://lists.gt.net/linux/kernel/2434768.

[6] T. Haerder, A. Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys,* vol. 15, no. 4 (December 1983), pp. 287–317.

[7] NVML install instructions: https://github.com/pmem/nvml/blob/master/README.md.

[8] libmemkind: https://github.com/memkind.

It's Better to Rust Than Wear Out

GRAEME JENKINSON

Graeme Jenkinson is Senior Research Associate in the University of Cambridge's Computer Laboratory, leading development of distributed tracing for the Causal, Adaptive, Distributed, and Efficient Tracing System (CADETS) project. Prior to working on CADETS, he had 13 years' experience working in the defense and automotive industries.

When a colleague of mine first enthused to me about Rust, I was skeptical. Back in the day, I'd cut my programming teeth developing software for safety-critical systems, and I'd learned the hard way that programming languages are frequently less sane than they first appear. Take C. Despite a considerable standardization effort, the C specification remains riddled with unspecified, undefined, and implementation-defined behaviors [2]. And even in 2016, researchers continue to explore the differences between the C ISO standard and the de facto usage [4].

While not all software engineers need be concerned with the seemingly esoteric issues of what happens when a bit field is declared with a type other than int, signed int, or unsigned int (it's undefined [2]), I'd worked too long with safety-critical and security systems to switch off this retentive part of my brain. And so, somewhat dismissively, I mentally parked Rust along with Go, Haskell, and all the other technologies that sound cool, but which I could never foresee actually using. Then early this year I had the opportunity to revisit Rust, and I found I'd been a bit hasty.

I had been developing a prototype for a distributed tracing framework built on top of DTrace. The prototype, written in C, acted as a DTrace consumer (interfacing with `libdtrace`) and sent DTrace records upstream for further processing (aggregation, reordering, and so on) using Apache Kafka. For a prototype this worked fine, but as the work progressed, I needed to rapidly explore the design space.

This task favored adopting higher-level language, but which one to choose? Like all good engineers, I started to list out my requirements. I needed a language that emphasized programmer productivity. It needed to easily and efficiently interface with libraries written in C (such as `libdtrace`). I also needed easy deployment, therefore languages requiring a heavy runtime (and Java specifically) were complete nonstarters. Good support for concurrency and, ideally, prevention of data races would be nice. And, finally, with my security hat on, I didn't want to embarrass myself by introducing a bucket-load of exploitable vulnerabilities. I thought back to that earlier conversation with my colleague; aren't these requirements exactly what Rust is designed for? And so I decided to give Rust a whirl, and I'm glad that I did, because I really liked what I found.

This article first appeared in the *Free BSD Journal,* Nov/Dec 2016.

So What's Rust All About?

Rust's vision is simple—to provide a safe alternative to C++ that makes system programmers more productive, mission-critical software less prone to bugs, and parallel algorithms more tractable. Rust's main benefits are [5]:

- Zero-cost abstractions
- Guaranteed memory safety (without garbage collection)
- Threads without data races
- Type inference
- Minimal runtime
- Efficient C bindings

The Rust language has a number of comprehensive tutorials, notably the "Rust Book" [5]. Therefore, rather than retreading that ground, I will instead highlight the features of Rust that I find particularly compelling. Along the way, I'll discuss the features of Rust that are most difficult to master. And, finally, I'll show how to get started programming in Rust on FreeBSD.

Fighting the Borrow Checker

Before diving in headfirst and firing up your favorite text editor (`vim`, obviously), it is important to understand Rust's most significant cost, its steep learning curve. On that learning curve, nothing is more frustrating than repeatedly invoking the wrath of the "borrow checker" (the notional enforcer of Rust's ownership system). Ownership is one of Rust's most compelling features, and it provides the foundations on which Rust's guarantees of memory safety are built. In Rust, a variable binding (the binding of a value to a name) has ownership of the value it is bound to. Ownership is mutually exclusive; that is, a resource must have a single owner. It is the borrow checker's job to enforce this invariant, which it does by failing early (at compile time) and loudly.

In the following example, taken from the "Rust Book" (*The Rust Programming Language*, 2016), `v` is bound to the vector `vec![1, 2, 3]`, a Rust macro creating a contiguous, growable array containing the values 1, 2, and 3. The function `foo()` is the "owning scope" for variable binding `v`. When `v` comes into scope, a new vector is allocated on the stack and its elements on the heap; when the scope ends, `v`'s memory (both the components on the stack and on the heap) is automatically freed. Yay, memory safety without garbage collection.

```
fn foo() {
    let v = vec![1, 2, 3];
}
```

Ownership can be transferred through an assignment `let x = y` (move semantics). But remember, ownership is mutually exclusive, so in the example below, when the variable `v` is referenced (in the `println!` macro) after the transfer of ownership to `v2`, the borrow checker cries foul: `error: use of moved value: `v``.

```
let v = vec![1, 2, 3];
let v2 = v;

println!("v[0] is: {}", v[0]);
```

In the next example, calling the function `bar()` passing the vector `v` as an argument transfers the ownership of `v`. When the owning scope, the function `bar`, ends, `v`'s memory is automatically freed as before. Ownership of `v` can be returned to the caller by simply returning `v` from `bar`. This approach would get tedious pretty quickly, and so Rust allows borrowing of a reference (that is, "borrowing" the ownership of the variable binding). A borrowed binding does not deallocate the resource when the binding goes out of scope. This means that after the call to `bar()`, we can use our original bindings once again.

```
fn bar(v: &Vec<i32>) {
    // do something useful v here
}

let v = vec![1, 2, 3];

bar(&v);

println!("v[0] is: {}", v[0]);
```

Immutability by Default

By default, Rust variable bindings are immutable. Having spent many an hour typing `const`, `*const`, and `final` in C and Java, respectively, this feature alone fills me with joy; and what is more, unlike `const`, it actually provides immutability. Variable bindings can be specified as mutable using the `mut` keyword: `let mut x = 10`. Also note the sensible use of type inference. Like variable bindings, references are immutable by default and can be made mutable by the addition of the `mut` keyword (`&mut T`). Shared mutable state causes data races. Rust prevents shared mutable state by enforcing that there is either:

- One or more references (&T) to a resource or
- Exactly one mutable reference (&mut T)

Choosing Your Guarantees

Rust's philosophy is to provide the programmer with control over guarantees and costs. Rust's rule that there can be one or more immutable references or exactly one mutable reference is enforced at compile time. However, in keeping with the overall philosophy, various different tradeoffs between runtime and compile time enforcement are supported.

A reference counted pointer (`Rc<T>`) allows multiple "owning" pointers to the same (immutable) data; the data is dropped and memory freed only when all the referenced counter pointers are out of scope. This is useful when read-only data is shared and it is non-deterministic to when all consumers have finished accessing the data. A reference counted pointer gives a different guarantee (that memory is freed when all owned pointers go out of scope) than the compile time enforced guarantees of the ownership system. However, this comes with additional costs (memory and computation to maintain the reference count). Similarly, mutable state can be shared (using a `Cell<T>` type); this again brings different tradeoffs for guarantees and costs.

Lifetimes

There is one final and rather subtle issue with ownership. Variable bindings exist within their owned scope, and borrowed references to these bindings also exist within their own separate scope. When variable bindings go out of scope, the ownership is relinquished and the memory is automatically freed. So what would happen if a variable binding went out of scope while a borrowed reference was still in use? In summary, really bad things invalidate Rust's guarantees of memory safety. Therefore, this can't be allowed to happen. Lifetimes are Rust's mechanism to prevent borrowed references from outliving the original resource.

In Rust, every reference has an associated lifetime. However, lifetimes can often be elided. The example below shows equivalent syntax with the lifetime (`'a`) of the reference `s` elided and made explicit:

```
fn print(s: &str); // elided
fn print<'a>(s: &'a str); // expanded
```

Global variables are likely to be the novice Rust programmer's first interaction with lifetimes. Global variables are specified with Rust's special static lifetime as follows: `static N: i32 = 5;`. A static lifetime specifies that the variable binding has the lifetime of the entire program (note that string literals possess the type `&'static str`, and therefore live for the entire life of the program). If I were to hazard a guess at where lifetimes next rear their heads, it would be storing a reference in a struct. In Rust, a `struct` is used to create complex (composite) datatypes. When Rust `structs` contain references (that is, they borrow ownership), it is important to ensure that any references to the `struct` do not outlive any references that the `struct` possesses. Therefore, a Rust `struct`'s lifetime must be equal to or shorter than that of any references it contains.

Efficient Inheritance

In contrast to C++ and Java's heavyweight approach to inheritance, Rust takes a muted approach; in fact, the word *inheritance* is studiously avoided. With traditional inheritance gone AWOL, classes are no longer needed. Having been freed from the confines of classes, methods can be defined anywhere, and types can have an arbitrary collection of methods. As in Go, inheritance in Rust has been boiled down to simply sharing a collection of method signatures. This approach is sometimes referred to as objects without classes. Rust Traits group together a collection of methods signatures—a Rust type can implement an arbitrary set of Traits. Thus, Traits are similar to `mixins`.

Fighting the Borrow Checker Redux

What makes Rust's ownership system so tricky to master? Ownership is not a complexity introduced by the Rust language; it is an intrinsic complexity of programming regardless of the language being used. Languages that fail to address ownership fail at runtime with data races and so on. In contrast, Rust makes issues of ownership explicit, allowing the language to fail early and loudly at compile time. Rust's borrow checker is like that friend you couldn't quite get on with on first meeting. Over time, and once they've helped you out multiple times, you realize that they've actually got some pretty great qualities and you're glad to have made their acquaintance.

Foreign Function Interface (FFI)

Another of Rust's features that particularly appealed to me is its support for efficient C bindings: calling C code from Rust incurs no additional overhead. Efficient C bindings support incremental rewriting of software, allowing programmers to leverage the large quantities of C code that are not going away anytime soon. External functions fall beyond the protections of Rust and thus are always assumed to be unsafe. It is important to note that there are many behaviors, such as deadlocks and integer overflows, that are undesirable but not explicitly unsafe in the Rust sense.

In Rust, unsafe actions must be placed inside an unsafe block. Inside the unsafe block, Rust's wilder crazier cousin "Unsafe Rust" rules. "Unsafe Rust" is allowed to break limited sets of Rust's normal rules, the most important being that it is allowed to call external functions.

In practice, calling C functions from Rust isn't always quite so straightforward as tutorials make out. Consider calling the function `dtrace_open()` from libdtrace. The C prototype for `dtrace_open()` is shown below:

```
dtrace_hdl_t *
dtrace_open(int version, int flags, int *errp)
{
        ….
}
```

To call `dtrace_open()` from Rust, we first specify the `dtrace_open()`'s signature in an extern block (`extern "C"` indicates the call uses the platform's C ABI). We can then call that function directly from an unsafe block.

```
extern crate libc;

...
```

```
extern "C" {
    fn dtrace_open(arg1: ::std::os::raw::c_int,
        arg2: ::std::os::raw::c_int,
        arg3: *mut ::std::os::raw::c_int) -> *mut dtrace_hdl_t;
}

fn main() {
    let dtrace_version = 3;
    let flags = 0;
    Let mut err = libc::c_int = 0;
    let handle = unsafe {
        dtrace_open(dtrace_version , flags, &mut err)
    };
}
```

But there is one significant problem: where is the type `dtrace_hdl_t` defined? While `dtrace_hdl_t` can be specified by hand, it contains many, many fields, which in turn use yet more new types that must be defined. Specifying all this by hand would be extremely tedious and error prone. Fortunately, there is a solution. C bindings can be generated automatically using Rust’s bindgen crate, `cargo install bindgen`. Unfortunately, `bindgen` is not a very mature tool. And, as a result, manually tweaking its outputs is often required (usually adding or removing mutability). With SWIG (Simplified Wrapper and Interface Generator) support for Rust not looking imminent, better native tooling for generating Rust bindings is desperately needed.

Package Management

The final, and in many ways most important, feature that attracted me to Rust was its support for modern application package management. Rust provides a flexible system of crates and modules for organizing and partitioning software and managing visibility. Rust crates are equivalent to a library or package in other languages, and Rust modules partition the code within the crate.

A Rust program typically consists of a single executable crate, which optionally has dependencies on one or more library crates. Reusable, community-developed library crates are hosted at crates.io, the central package repository for cargo, Rust’s package management tool (crates.io is broadly equivalent to Python’s PyPI). Rust’s `cargo` tool fetches project build dependencies from crates.io and manages building of the software. Yeah, I know, does the world really need yet another mechanism for packaging software, resolving dependencies, and building software? Well perhaps not, but `cargo` actually works really well, though for those with experience with Maven, the bar hasn’t been set that high.

Getting Started on FreeBSD

Rust’s platform support is divided into three tiers, each providing a different set of guarantees. FreeBSD for x86_64 is currently a Tier 2 platform. That is, it is guaranteed to build but not to actually work. Despite the lack of a guarantee, in practice, things generally seem to work pretty well. Tier 2 platforms provide official releases of the Rust compiler `rustc`, standard library `std` (`pkg install rust`), and package manager `cargo` (`pkg install cargo`). FreeBSD’s binary Rust package is currently (at the time of writing) at v1.12 with v1.13 being the latest stable release. Once installed, Rust can be updated to the latest version by executing the rustup script:

```
curl -sSf https://static.rust-lang.org/rustup.sh | sh
```

32-bit FreeBSD sits in Rust’s lowly third tier where, without guarantees about either building or working, things are pretty unstable. For example, Rust 1.13 recently shipped in spite of a serious code generation bug on ARM platforms using hardware floating point. Here be dragons, so beware!

Where Are We Now?

Rust started life in 2009 as a personal project of Mozilla employee Graydon Hoare. In subsequent years, Rust has transitioned to a Mozilla-sponsored community project with over 1,200 contributors. Since the 1.0 release, delivered in June 2015, Rust has been used in a number of real-world deployments. June 2016 saw another major milestone on the road to maturity, with Mozilla shipping Rust code for the Servo rendering engine in Firefox 48.

So people are using Rust, but does it really deliver on its vision of providing a safe alternative to C++? I think the answer is pretty much yes, though the differences aren’t all that huge. For example, in C++, a `unique_ptr` owns and manages an object and disposes of that object when the `unique_ptr` goes out of scope. Furthermore, ownership can be transferred using `std::move;`, and as a bonus, there is type inference using the `auto` keyword. But in spite of these similarities, smart pointers don’t give everything that Rust’s ownership system does. In the example below [3], accessing `orig` after the move results in a segmentation fault at runtime—a morally equivalent example in Rust would fail to compile. Failing early is a good thing. That a careful and skilled C++ programmer wouldn’t make such mistakes is somewhat of a circular argument, because if such mistakes weren’t widespread, languages attempting to prevent them wouldn’t exist in the first place. C++ also lacks a module system and has a number of pretty ropey features like header files and textual inclusion. These are all wins for Rust.

```
#include <iostream>
#include <memory>

using namespace std;

int main ()
{
    unique_ptr<int> orig(new int(5));

    cout << *orig << endl;
    auto stolen = move(orig);
    cout << *orig << endl;
}
```

How does Rust compare with C++ on performance? Control studies comparing the performance of idiomatic C++ and Rust are hard to find. A comparison between Firefox's Servo and Gecko rendering engines (written in Rust and C++, respectively) reported that the Rust Servo engine was on the order of twice as fast [1]. While these figures should be taken with a pinch of salt, the consensus opinion is that Rust is at least comparable in terms of performance to C++. One of the reasons for this is that Rust features, like genuine immutability, allow optimizations that can't be made in C++. And Rust's semantics bring significant potential for further optimizations.

Despite the advances made in deploying Rust in production environments, problems remain. The Rust ABI is unstable, and as with the Glasgow Haskell compiler, a stable ABI may never happen, almost certainly not anytime soon. This problem most impacts Rust native, shared libraries because without a stable ABI, they are incompatible across major version changes. But ABI instability isn't a showstopper. So is there a technical barrier to upstreaming Rust code to FreeBSD, for instance? In my opinion, I don't think so, but I'd be interested to hear others' opinions on both the technical and political challenges of doing so.

I like Rust. It's fun. And isn't that what really makes us come into work in the morning?

References

[1] B. Anderson, L. Bergstrom, M. Goregaokar, J. Matthews, K. McAllister, J. Moffitt, and S. Sapin, "Engineering the Servo Web Browser Engine Using Rust," in *Proceedings of the 38th International Conference on Software Engineering Companion* (May 2016), pp. 81–89.

[2] L. Hatton, *Safer C*, 1st ed. (McGraw-Hill, 1995).

[3] S. Klabnik, Unique Pointer Problems, Steve Klabnik's home page: http://www.steveklabnik.com/uniq_ptr_problem/.

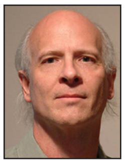
[4] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. Watson, and P. Sewell, "Into the Depths of C: Elaborating the De Facto Standards," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2016), pp. 1–15.

[5] *The Rust Programming Language*, "Getting Started": https://doc.rust-lang.org/book/getting-started.html.

Interview with Eric Allman

RIK FARROW

Eric Allman earned his BS and MS degrees from UC Berkeley in 1977 and 1980. He wrote sendmail and syslog, which became part of BSD in 1981. In 1998, Allman and Greg Olson founded Sendmail, Inc. Currently, Eric works as a Research Assistant on the Swarm project at UC Berkeley. eric.allman@gmail.com



Rik is the editor of *;login:*. rik@usenix.org

I first heard Eric Allman speak during a LISA tutorial. Eric was explaining some of the intricacies of `sendmail`, the mail server software he had written while at UC Berkeley in the early '80s.

I later cornered Eric during a conference reception, an action very unlike me. But I was determined to find out why Eric had included what I thought were three backdoors in `sendmail`, something that turned out to be incorrect. Eric also mentioned wishing he had received even a fraction of one cent for each copy of `sendmail` then in use. He later started a company that provided support for `sendmail`, a company that followed the rise and fall of the Internet boom in the late '90s.

I met with Eric in person last February in Cory Hall at the University of California, Berkeley, where he currently works. We discussed some of his past and current work.

Rik Farrow: Your experience with open source has been interesting to say the least.

Eric Allman: Open source, or if you prefer, free software, existed long before most people thought. They had IBM Share way, way back. One of the main reasons you used to go to USENIX conferences was that you always brought along six tapes with you and you walked away with six tapes, but they weren't the same six tapes you brought in. That was one of the big things about them, not just to go to talks.

RF: I believe that your open source adventure started by creating `delivermail` to handle delivery of mail that required transport beyond the local system.

EA: `delivermail` had no transport mechanisms, like `binmail`, which just delivered mail to a spool file. `delivermail` would examine the email address looking for exclamation points or at signs. If the email address didn't have these punctuations, it just appended the mail to the spool file. If the mail address did include these punctuations, then it would send the email to the correct command. Another difference between `sendmail` and `delivermail` was that `delivermail` didn't do any address translations. People had to become experts in what John Quarterman called "the matrix." One of the goals of `sendmail` was to make it easier for people to survive in this multi-network world, which included Berknet, Arpanet, and UUCP.

RF: I wanted to ask you about the backdoors in `sendmail`. When I first asked you about this many years ago, you told me you were a student maintaining `sendmail` on a small number of systems, and then someone copied `sendmail` to a machine you had no access to. The owners of that machine then demanded that you fix a bug only expressed on that system.

EA: Precisely. So I said let me log in and look at it. And they said we can't allow someone who is not part of the administrative staff onto the machine, which is normally a pragmatic approach to security. I said I will come into your office and someone can watch over my shoulder and make sure I don't do anything bad. They said, no, we can't let you on the machine. Then I can't fix your problem, and they said you have to fix our problem.

RF: A double-bind.

EA: They got more and more insistent, that I had to fix this magically somehow. And that's when the backdoor went into `sendmail`. If they won't let me on the machine, well, here's a new version, why don't we see if it fixes the problem. And it did.

The lesson out of that is the systems, including the humans that maintain them, will find a way around the security to get the job done. They actually lost security, and it would have cost them nothing to just have somebody watch me.

RF: That backdoor stayed in there for a long time.

EA: My mistake was in not taking it out immediately. The backdoor was so convenient, I thought maybe I'll leave it in and it will contribute to development. I pretty much forgot it was there.

RF: Then there was the problem with the frozen configuration file, that meant that the wizard mode password would get deleted when that was used [1].

EA: Yeah. Keep in mind that there was exactly one backdoor. There were other bugs, like ones that allowed you to clobber the stack, and you could do nefarious things there. But these were just flat out bugs.

RF: I thought that the Internet Worm used Debug, where you send a shell script as the recipient [2].

EA: Someone else put that into `sendmail`. Somebody tried to get me to put that in the `sendmail` distribution, and I said, "Are you nuts?"

RF: There's a recent movement called language security, or LangSec for short. LangSec followers believe that a key problem for most software is input parsing. It turned out that there were a lot of bugs in `sendmail` all associated with parsing, and that's because parsing is difficult.

EA: The biggest problems, of course, are buffer overflows, which are the scourge of security everywhere, and those pretty much went away after we had yet another buffer overflow and we said, "Screw it." We are just going to go around and every place we see `*p++` we are going to put a test around it.

RF: Right. In 2003, I remember that LSD had a sort of a cool exploit which wasn't a typical buffer overflow, as they figured out how write a new binary in the right place and essentially replace `sendmail` with a shell attached to an outgoing connection from port 25 [3].

EA: If I recall correctly, I had a fixed-length buffer which was pointers to opening bracket, so when I found the closing bracket I could return a pointer to the correct address.

RF: I had been single-stepping through the code, looking for where the bug was. I did find where the bug was, but by reverse engineering the patch to the source code, which of course is what hackers were doing. That's why when you said `*p++`, that brought up the memory. That was the last `sendmail` bug I remember seeing. But over the years, that whole process was very painful.

EA: Well, all I can say is `sendmail` was never as bad as Flash.

RF: Another thing you said during that short meeting, we probably only talked for 10 minutes, was if you only had a tenth of a cent for each copy of `sendmail` in use. So you eventually started `Sendmail`, Inc. Was that your idea, or did somebody approach you?

EA: I had just come out of a disastrous job, and I was sitting around, getting a little enthusiasm back, thinking what do I do next. I looked around a little bit and someone, I don't remember who, asked me if I thought about commercializing `sendmail`. I didn't know how to do that. Then I ran into a friend of mine whom I had worked for 10 years prior, and he had gone the corporate route. He helped me write a business plan and eventually agreed to come on board as their first CEO. He was a very good CEO for a company in that state. He had the sense to say at some point I'm stepping down, I like starting companies a lot more than I like running them.

RF: Those really are two different things.

EA: `Sendmail`, Inc., was a very interesting place to work, a lot going on, maybe too much. Then the Internet bubble burst [March 10, 2000]. We survived because the co-founder and I who were co-operating the company tended to be a bit more fiscally conservative than a lot of people in those days. We had a board member who said you aren't spending money fast enough, and at the very next board meeting he said you need to downsize instantly, how could you let yourself get so big. He was not my favorite board member.

RF: You survived.

EA: We did survive, but let's not go into corporate politics. `Sendmail`, Inc. got bought by Proofpoint, and the investors, including myself, got nothing out of it. But most of the employees had jobs, and the customers got taken care of. Investors, employees, customers, two out of three ain't bad. I actually was pretty happy with that.

RF: What did you do after `Sendmail`, Inc.?

EA: I kind of retired. I can afford to live as long as my tastes don't get too extravagant. That's fine, I don't have a lot of expenses. I had some offers. Then one came by email, about a new lab [4], with an invitation to come by and see what they were doing. They said they have seminars on Thursdays, including free lunch. So I went for the free lunch. There was a research meeting right after the seminars, and I started staying for that, and at some point the person who became my supervisor said, why don't we pay you?

RF: So you are doing coding? Looks like you are involved in embedded systems here...

EA: I'm working on, loosely speaking, data storage and security for what I hesitate to call the Internet of Things, because everyone thinks they know what that means.

RF: IoT is a very broad term, from video cameras running Linux to tiny sensors with 1K of RAM...

EA: 1K? If you're lucky. There's a paper called "The Cloud Is Not Enough" [5] done by our group. These days, everyone is saying that whatever your problem is, the answer is "cloud." That's when you know you need to be looking elsewhere. The cloud may often be adequate, but there are times where it's nowhere near fast enough.

We are looking at using more distributed storage and computing. You have the cloud there, and if you have big compute jobs, you can send them off to a cloud service. If you are storing massive amounts of data, you can send them off to the cloud. We don't have objection per se to the cloud. But where somebody just unlocks a door or turns on the lights, I see no particular reason why we need to go up to the cloud and back. Our concept is that there are Swarm boxes [he indicates one sitting on the desk beside us, looking a bit like a WiFi router], and these do local processing. This box is fanless, essentially an Intel NUC; we actually have some bigger servers for storage with multiple terabytes of disk on it. Kind of a balance between the two.

RF: And there's the big problem with the Internet of Things: getting devices to play well together. And the big players have been trying to get their separate solutions accepted.

EA: Lots and lots of stovepipes.

RF: Yes, too many stovepipes. And it's *my data*, and I don't want to be sharing it to aid in marketing stuff to me.

EA: That's another point we are trying to address. It's your data, you should have access to it, you should have control over it, and we are doing security stuff: everything that goes into the db is signed, and if not marked public data, then encrypted. There are some performance issues with public key encryption, which is very slow. But that's exactly what we have implemented right now. We do a public key signature on every message that goes in, and, yes, encryption slows our system down. We have techniques for saying only every 10th record gets signed, something like that, and use hash chains. Hashes compute fast, to verify integrity. So that part's important.

References

[1] Steve Bellovin explains why the wizard mode password got deleted when a frozen configuration (sendmail.fc) file got used: http://textfiles.com/internet/sendmailwb.txt.

[2] E. H. Spafford, "The Internet Worm Program: An Analysis," Purdue Technical Report CSD-TR-823: http://spaf.cerias.purdue.edu/tech-reps/823.pdf page 5, paragraph 3.

[3] LSD sendmail reference: http://www.ouah.org/LSDsendmail.html.

[4] Swarm Lab: https://swarmlab.eecs.berkeley.edu/.

[5] B. Zhang, N. Mor, J. Kolb, D. S. Chan, N. Goyal, K. Lutz, E. Allman, J. Wawrzynek, E. Lee, and J. Kubiatowicz, "The Cloud Is Not Enough: Saving IoT from the Cloud," in *Proceedings of the 7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '15)*: https://www.usenix.org/system/files/conference/hotcloud15/hotcloud15-zhang.pdf.

Migrating to BeyondCorp

Maintaining Productivity While Improving Security

JEFF PECK, BETSY BEYER, COLIN BESKE, AND MAX SALTONSTALL

Jeff Peck is a Technical Program Manager for CorpEng in Google. He previously worked at companies large and small around Silicon Valley, doing software engineering and program management for a variety of projects in the telecom, server, and network application domains. He has a BS in computer, information, and control sciences from the University of Minnesota. jpeck@google.com

Betsy Beyer is a Technical Writer for Google Site Reliability Engineering in NYC. She has previously provided documentation for Google Data Center and Hardware Operations teams. Before moving to New York, Betsy was a lecturer in technical writing at Stanford University. She holds degrees from Stanford and Tulane. bbeyer@google.com

Colin Beske is a Technical Program Manager at Google. Since joining in 2010, he has worked on IT support, printing operations, and internal change management. Prior to Google, he held positions in systems and networking engineering. He has a BA in computer science from Oberlin College. beske@google.com

Max Saltonstall is a Technical Director in the Google Cloud Office of the CTO in New York. Since joining Google in 2011, he has worked on video products, internal change management, IT externalization, and coding puzzles. He has a degree in computer science and psychology from Yale. maxsaltonstall@google.com

If you're familiar with the articles about Google's BeyondCorp network security model published in *;login:* [1-3] over the past two years, you may be thinking, "That all sounds good, but how does my organization move from where we are today to a similar model? What do I need to do? And what's the potential impact on my company and my employees?" This article discusses how we moved from our legacy network to the BeyondCorp model—changing the fundamentals of network access—without reducing the company's productivity.

Among the many challenges that a migration to a BeyondCorp-type model entails, several are particularly notable:

- This process affects the entire company. Getting everyone on board and keeping everyone aligned and informed requires commitment and buy-in from all levels of management. That commitment needs to be reinforced through extensive communications with all parties involved, from the teams that own individual services, to management, to support teams, to users.
- The migration can't be done overnight. The process is multi-layered and incremental, with stages of information gathering, trial deployments, corrections to processes and technology, and exceptions and remediation where and when necessary.
- The process requires changes at many or all layers of the stack: networking, security gateways, client platforms, and backend services. Partitioning the changes in order to make progress independently at different layers makes this multi-pronged undertaking more approachable and manageable.

The following sections discuss how we partitioned the BeyondCorp migration effort, and the tools and technologies we used to bring the various layers into alignment while minimizing negative impact on users.

Prerequisites: Commitment and Communications

Before you can undertake a migration to a BeyondCorp-like model, you need buy-in from top level management and other stakeholders in your organization. Step one here is understanding and communicating the motivation for the migration: you want to reduce the threat of a successful cyberattack while maintaining productivity. You need to document the rationale behind the proposed migrations, the threat model, and the costs of doing "business as usual." Then be prepared to explain to each line-of-business why this process is valuable and essential. As with all security operations, deploying a new model comes with a price: new tools, additional processes, and changes in habits to apply. Top-level management needs to actively support this change and drive the motivation and commitment down to all stakeholders.

Armed with a charter and commitment from management, identify and enlist the support of leaders in crucial areas: security, identity, networking, access control, client and server platform software, business-critical application services, and any third-party partners or outsourced IT functions. The leads should identify and enlist the subject matter experts for each area and commit their time and energy to the process. Our BeyondCorp team was a globally distributed virtual team headed by a director responsible for policy decisions and a technical program manager to drive and coordinate execution. Active membership changed over time, but the stakeholders, team leads, and other contributors were consistently linked through online documentation, group email, and regular face-to-face and video conference meetings to stay informed of current processes and project status.

As the effort progresses, the usual rules of change management apply, because each work group will have its own concerns and priorities. Listen to feedback and adapt to the special circumstances and requirements of each contributor or affected group. Publishing plans and information is necessary but insufficient; interactive communication (ideally done in person, but at minimum conducted over video or audio conferencing) speeds assistance and adoption.

Partitioning for Progress

The overall objective of the BeyondCorp program is to transition from a network that allows clients to directly access servers to a new network design, one that removes the privilege of direct access to backend servers. For more details, see "BeyondCorp: A New Approach to Enterprise Security" [1], the first article in this series. To this end, we considered removing privileged access from the legacy VLAN by blocking each application or server in sequence. This strategy was less than ideal for two reasons: it would be difficult to deploy and coordinate at the network layer, and it posed increased risks to productivity at the application layer. Instead, we decided to deploy a new VLAN in its final BeyondCorp configuration. This VLAN only permits access to the server network through access control gateways, ensuring that all traffic flows are authenticated, authorized, and encrypted. Rather than incrementally restricting the privileges of the legacy VLAN, we incrementally moved devices to this new end-state VLAN.

The VLAN migration project achieved the complex but critical goal of removing user devices from the legacy "privileged" network and assigning them to the new Managed Non-Privileged Client (MNP) VLAN. This move had a key constraint: any legacy application that expected or required direct access to the server network would fail when run from a workstation on the new VLAN. Therefore, achieving this migration without breaking business-critical operations was an immediate subgoal. We used a three-pronged strategy to meet this subgoal:

1. Extensively analyzing network traffic logs
2. Identifying and remediating noncompliant applications
3. Migrating devices **after** determining they would be successful on the new network

This approach allowed the network layer to roll out the new configuration and achieve stability independently from other parts of the BeyondCorp program. The BeyondCorp design includes the use of 802.1x for network admission and VLAN assignment, which we utilized to isolate the network layer from the details of the migration policies. Higher level software and data analysis determined each device's VLAN assignment, which the RADIUS servers then communicated to the network layer.

Realizing these goals was a vast undertaking that required changes at almost every layer of the stack. Rather than attempting to introduce change to all of these layers in a single transition (undoubtedly a recipe for disaster), we pursued a partitioned approach that entailed:

- Decoupling network layer projects: new VLANs, 802.1x, RADIUS policy server
- Decoupling client platform upgrades: certificate generation and installation, user authentication tools
- Migrating devices incrementally as we remediated services and workflows
- Continuously refining our processes and procedures

First Steps: An 802.1x Network

In the first phase of BeyondCorp, we installed certificates on each user device and transitioned to 802.1x for all network access grants. This seemingly simple step implied several new developments: a certificate authority, tools to install certificates on company-managed devices (for each OS type), enabling 802.1x on the network switches, and integrating with a policy-driven RADIUS service. We undertook all of these developments in parallel.

The security team designed a new Certificate Authority with APIs to enable the various per-OS platform management teams to obtain and install certificates on their platforms. Each platform team independently deployed the software, tools, and telemetry to enforce and monitor certificate rollout to each device. We created the processes for mass distribution and maintenance of certificates while we were still working on integration with the access switches.

Likewise, re-provisioning the access switches to include the new VLAN definitions proceeded in parallel—we enabled and later required 802.1x and RADIUS-provided VLAN assignments. Automated scripts audited the switch upgrades to identify switches not yet provisioned with the new VLAN. As a result, the RADIUS server would not request a VLAN assignment that wasn't available on a particular switch.

We used 802.1x so we could move control of VLAN assignments from the network layer to a VLAN policy server. Because we wanted to reduce failures caused by the new RADIUS server, the initial policy simply matched the existing assignments (which included complex blacklists and whitelists). We first deployed the policy server in an auditing mode that compared the new assignments with the legacy assignments. When the differences were sufficiently few, we enabled the new policy. From that point on, we could manage device assignment to VLANs in near-real time using high-level software and data-driven policies. Using this simple initial policy allowed us to enable dynamic VLAN assignments in the network while the end-state (and transition) policies were still being developed.

Success-Oriented Migration

It took years to fully deploy the 802.1x layer, and several more years before the inventory-based tiered access VLAN assignments were available as input to the RADIUS policy server [2]. While those developments were underway, we wanted to identify our two main groups of users and application services: those that were ready for BeyondCorp versus those that needed to upgrade their network and security capabilities to become BeyondCorp compliant. Our first step was to capture and analyze traffic from the network routers. By logging and analyzing a fractional sample of all traffic through the corporate routers, we discovered patterns of noncompliant usage. As a second-order benefit, this analysis also helped us discover unusual, unexpected, and unauthorized traffic on the network. Identifying these applications meant we could start the reengineering earlier and avoid disrupting the users of these systems.

Some networking use cases, such as workstations using an NFS/CIFS file server, were obviously noncompliant. Although a NFS/CIFS file server is a simple way for users to maintain a single, common copy of their files, the underlying protocol didn't support our desired security properties (strong encryption and authentication). To eliminate this dependency, we initiated a major project early on to accomplish two goals: moving NFS home directories to local disk with automatic backup to secure cloud storage, and replacing other NFS usage with Google Drive or other secure file-sharing technologies. Even so, some applications, like CAD (computer-aided design) editors, are deeply dependent on NFS and required special solutions before we could move their users and workstations to the restricted MNP VLAN. We discuss the details of our framework for handling these special requirements in the "Remediating Difficult Use Cases" section below.

Other noncompliant workflows were not so obvious but would nevertheless fail when subjected to the restrictions of the MNP network ACL. This failure was by design, as we couldn't assume that NFS, RDP, SQL, etc. had adequate authentication, authorization, and encryption. Detecting these workflows and re-enabling productivity by changing the device's network assignment is difficult and time-consuming when remediation must happen at the network layer. To avoid large impacts on productivity (not to mention user morale), we needed an analysis-driven strategy to detect failing workflows and correct them before assigning users to the MNP VLAN.

To facilitate easy analysis and user workflow testing on the non-privileged network, we created a client-based network ACL simulator that identified network packets that would be blocked by the MNP ACL. The underlying technology used Capirca (see [4] for the source code) to create local iptables or Packet Filter rules from the actual MNP network ACL. During the analysis and migration phase, user devices continued to operate on the privileged network, while the MNP-simulator monitored network traffic and logged the source and destination of all non-MNP-compatible traffic to a central repository. The IP source address identified the failing user, and the IP destination address identified the failing service. By analyzing the logs over time (with appropriate privacy constraints in place), we could identify devices with MNP-compliant traffic and assign them to the MNP VLAN. Likewise, we could identify devices, users, and services that relied on noncompliant traffic and initiate projects to move those services to alternative solutions. Over time, more devices became compliant and were automatically assigned to the MNP VLAN.

In a second mode, the MNP-simulator can actually block/drop the non-MNP traffic, thereby enforcing the MNP ACL without relying on network level deployment of the MNP VLAN and the 802.1x pipeline. Although we ultimately enforce the ACL in the network equipment, where it is isolated from user (or hacker) abuse, enabling and disabling this "enforcement" mode in the client workstation is much easier and faster during the trial and transition period. Client-side enforcement served as both an important step in the migration process and a self-service tool for testing. Without this feature, we wouldn't have gained the confidence we needed to move devices to MNP at nearly the speed (or with the high level of success) that we did.

Figure 1 shows the pipeline for moving Google computers to the Managed Non-Privileged (MNP) network.

Handling Easy Use Cases with the Access Proxy

Google's basic security policy requires that all traffic that flows from workstations to servers is:

- Authenticated (to identify the device and user making the request)
- Authorized (to verify that the user and device are allowed to access the backend resource)
- Encrypted (to prevent eavesdropping)
- Independently logged (to aid in forensic analysis)

The Access Proxy [3] achieves all these requirements for HTTP/S traffic and for our HTTP-encapsulated SSH traffic.

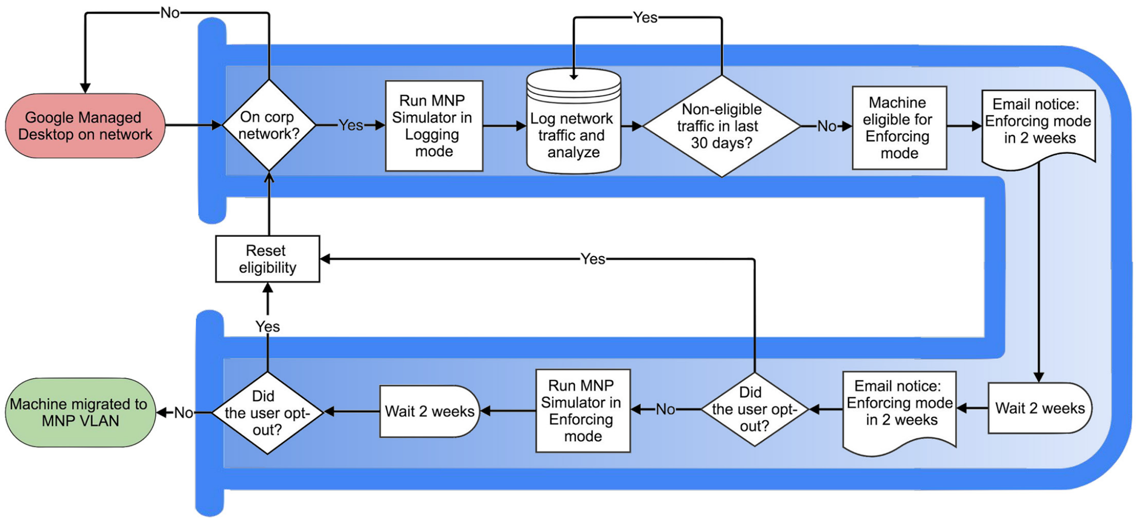

Google Managed Desktop on network
No
On corp network?
Yes
Run MNP Simulator in Logging mode
Log network traffic and analyze
Yes
Non-eligible traffic in last 30 days?
No
Machine eligible for Enforcing mode
Email notice: Enforcing mode in 2 weeks
Reset eligibility
Yes
Yes
Wait 2 weeks
Email notice: Enforcing mode in 2 weeks
Did the user opt-out?
No
Run MNP Simulator in Enforcing mode
Wait 2 weeks
Did the user opt-out?
No
Machine migrated to MNP VLAN

Figure 1: The pipeline for moving Google computers to the Managed Non-Privileged (MNP) network

Happily, most of our high-usage applications are browser-based Web applications. This condition is both "happy" and by design: Google is somewhat unique in the industry in its core philosophy of using browser-based applications when possible. We provided tools and documentation to each Web application provider so each could configure their application to run behind the Access Proxy.

When an application is behind the Access Proxy, corporate and public DNS contains a CNAME that resolves to the Access Proxy, so the URLs for such applications work from both corporate and public networks with equivalent ease and security. The ability to access corporate applications from public networks meant that authenticated remote users could access the corporate Web applications without diverting to initiate a VPN connection. As a result, the overhead for using and supporting VPN connections for remote work immediately and dramatically decreased. According to our rough estimates, the resultant productivity gains easily outweigh the implementation costs of BeyondCorp.

Once browser-based applications were secured behind the Access Proxy, we could make dramatic progress. We activated an automatic process for analyzing, verifying, and migrating devices to the non-privileged network; within a year this process moved over 50% of the fleet to non-privileged network access.

Remediating Difficult Use Cases

While we could handle the vast majority of applications via the Access Proxy, other applications weren't so easy. Our plans and schedules also had to address the reality of the long tail of non-Web cases that required additional time and resources to migrate. Evolving these use cases to become compliant required new tools, technology, and workflow modifications.

In particular, some of our workgroups use third-party desktop or "thick client" applications that are not HTTP-based, which entail a special set of problems. For example:

- Some tools are intrinsically designed to rely on network mounted file shares.
- Java applications may use RMI (Remote Method Invocation) or other direct socket connections.
- Many tools may be linked to license servers using non-HTTP sockets and protocols.

Even applications that use HTTP may be problematic due to obscure, unexpected failure modes. For example, some applications aren't designed to present a client certificate or proper user credentials, while some have logic built into the load balancing layer that doesn't mesh well with the Access Proxy. For some of these cases, we tweaked the Access Proxy to allow traffic coming from the MNP VLAN to pass without a certificate. We felt comfortable with this temporary strategy because the device had to present a certificate in order to access MNP. Each problematic case required a diagnosis and remediation project.

To address the class of hard cases, we developed a solution using a multi-port encrypted tunnel to carry application traffic between the client and server:

- When initiating a connection from client to server, the Access Proxy applies the usual user and device authentication and authorization.
- Routing tables on the client direct packets to a TUN device that captures and encrypts traffic to specific backend servers.
- The encrypted packets flow directly between the client and encryption server using a UDP-based encapsulation protocol.
- The encryption servers only allow traffic to the services and ports for which the application needs access.

Use Case	Solution
Browser-based HTTP/S	Access proxy
Naive HTTP cmd-line applications: We provide a client-side proxy server that supplies the platform certificate to achieve an authenticated and encrypted connection to the Access Proxy. We then direct the naive application to that localhost proxy.	Local authenticating proxy
Single TCP connection: For applications that need a TCP socket to a server, we can often arrange to establish an SSH connection to a backend bastion, and tunnel the port for the naive TCP application.	SSH tunnel and port forwarding
Many ports or unpredictable port numbers	Encrypted service tunnel
Latency-sensitive, real-time, UDP flow	Encrypted service tunnel

Table 1: Approaches to solving problematic workflows

This approach allows legacy third-party applications to more securely connect to their servers from any network and still assert the BeyondCorp invariants of authentication, authorization, and encryption.

Table 1 shows our general approach to resolving difficult workflows. For more detailed information, see "BeyondCorp Part III: The Access Proxy" [3]. In some cases, the solution shown in the table also required users to modify a workflow by running a script or providing the necessary authentication before accessing the backend resources.

Some essential framework services were noncompliant. Rather than block all migration, we temporarily opened access from MNP to the specific ports or servers for these critical services. To prevent these temporary exceptions from becoming commonplace and subverting the basic goals of BeyondCorp, we only allowed such exceptions when a service had a concrete plan for implementing and deploying a compliant solution.

As we remediated each application or use case, the automated process for analysis, verification, and migration moved more users and devices to the non-privileged VLAN. As we progressed, the network logging and analysis provided ready metrics about the number of users and devices that were successful on MNP.

Incrementally Rolling Out and Continually Refining Our Approach

The MNP simulator, analysis pipeline, and the subsequent automatic assignment of devices to the MNP VLAN was a significant software development and process creation project. As such, we developed and deployed it incrementally: we tested each phase on small groups, continuously fixed the software, adjusted user messaging when appropriate, trained the tech support team, and then gradually expanded to full-scale deployment.

The simulation and pre-analysis approach helped us avoid negative impact on users while we identified users of noncompliant workflows. However, because this approach assigned all newly provisioned, unanalyzed devices to the privileged network and didn't prevent unmigrated users from using or creating new noncompliant applications, it wasn't an acceptable long-term strategy. After reducing the number of exceptions by remediating the high volume use cases, we changed our approach to a policy of "MNP by default." Proceeding site by site, we assigned all devices to MNP, granting exceptions to devices belonging to users in job functions that use unremediated applications. This policy-based assignment marked the evolution from "*Prove* the user will be successful before migrating their devices" to "*Assume* the user will be successful and migrate their devices."

Scaling Support to Minimize Impact on Employees

Using the tools and processes discussed above, we were able to automatically identify, contact, and migrate entire groups of users. However, we also needed ways to assist people and communicate with users, both in advance of change and when something went wrong. A combination of specialized training for tech support and strategies to scale user communications and interactions was critical in shifting workflows to the new model.

Empowering Tech Support

We trained a select group of technicians in our support organization to become champions of the new BeyondCorp model and primary local points of contact. From the early stages of rollout, these techs helped affected users return to work quickly without compromising migration strategies, and also efficiently escalated appropriate issues to implementation and policy experts.

Initially, these specially trained technicians were granted more advanced access to remediation systems than their fellow technicians. As the first observers of the BeyondCorp rollout, they could anticipate what access, tools, and processes the rest of tech support would need. Additionally, they trained the rest of the support organization through global tech talks, discussion lists, brown bag lunches, and office hours. As knowledge was disseminated, we expanded system access to all of support.

Establishing local subject matter experts enabled us to engage directly with teams that had incompatible workflows. By working with one knowledgeable point of contact, teams had direct lines of communication to project experts and could collaboratively find solutions. Simultaneously, technicians were empowered and encouraged to add new temporary workarounds or fixes to internal documentation as soon as they identified problems. Distributing the power to solve problems to as wide a network as possible enabled us to efficiently share knowledge and scale support.

Self-Service Help
To avoid a flood of queries and concerns, we needed a way to minimize confusion and answer common questions without personal intervention by support personnel. When a user was selected for migration, we automatically sent them an email containing a clear timeline, an idea of how the migration would impact their work, and links to project information, FAQs, self-help, and escalation points.

We also provided a self-service Web portal that allowed users subject to business-critical time constraints to delay their migration. To answer questions and further disseminate knowledge at scale, we created an internal discussion list where people could crowdsource answers. Using analysis of common questions, we were able to quickly iterate the initial email communication and project documentation.

Throughout the rollout we also iterated and improved error messaging with a dedicated Web application. This application clearly identified common problems (for example, explaining why a user was denied access to a certain resource), provided steps for resolution, and linked to knowledge-base articles. Users could fix common issues such as group membership and certificate problems themselves, further reducing tech support requests. The Web application also helped technicians by coalescing information from the many different layers and systems into a single series of actions to solve an error.

Internal Publicity Campaign
To raise awareness of BeyondCorp, we ran an internal publicity campaign with laptop stickers, common logos and wording, and visible articles posted throughout our offices. These materials pointed to self-service help and office hours open to anyone with any question. By focusing on informing, educating, and helping, we directly built trust, goodwill, and buy-in with our users. Corporate communications and tech writer involvement were critical throughout the process—especially in the early phases, when we needed to provide a clear picture of the program's intent and impact.

Phased Rollout
BeyondCorp began as a small-scale pilot, geographically close to the project team. We increased the rollout over time by progressively targeting locations with local technical experts, eventually expanding to increasingly risky workflows and sites further from the project team. We didn't migrate critical business workflows until we had a history of success, strong buy-in from users, and confidence in our strategy. During this process, tech support load decreased as rollout size and affected workflows increased. Phasing our approach was a key element of its success.

End Result
By continually analyzing and improving all of the methods described above, we built a system that ensured the BeyondCorp rollout could scale globally without negatively affecting business, support, or user experience. Rather than simply "throwing more people at the issue," we scaled our efforts by building systems and processes to efficiently handle questions, escalations, and training. Additionally, we were able to trust our users to help us enable change by relying on information, openness, and agreement on a shared goal.

We carefully tracked support incidents caused by the BeyondCorp rollout as we moved more and more of the company onto this model. In recent months, BeyondCorp is responsible for only 0.3% of issues handled by our tech support organization. From an initial rate of 0.8%, escalations have steadily decreased with the help of improved documentation, training, messaging, and rollout methodology. Compared to similar wide-scale internal IT changes at Google, BeyondCorp has caused 30% fewer support issues.

Conclusion
There is always tension between the urgency to improve security and resistance to changing the habits of end users. When infrastructure and workflow changes threaten to impact productivity, this tension only escalates. Achieving a balance between progress and stability is more art than science. BeyondCorp's keys to success and acceptance were analysis, adaptive planning, and proactive communications.

By partitioning BeyondCorp changes into independent units, we could make progress in parallel, and user impact at each stage was minimal. Although it took years to deploy BeyondCorp across its many layers, each milestone came with benefits.

Cumulatively, we made remote access significantly easier and faster, simplified network management, and strengthened our security posture.

Creating the technology to implement the BeyondCorp security model is a challenge. Planning the rollout and managing the migration of users to that technology is just as challenging. It's essential to ensure that each transition has minimal impact on users and does not break ongoing productivity. Each successful transition brings fresh awareness of the value of the program and provides continued enthusiasm and acceptance of the program goals by both users and management. We succeeded by empowering a cross-functional team with representatives from each of the technology and implementation teams, security and policy stakeholders, and specialists in end-user support and communications.

At Google, we've been able to apply what we learned during the BeyondCorp effort to other programs and services—most notably, the new services we've recently added to Google's Cloud Platform (such as the Identity-Aware Proxy). One of the biggest lessons of BeyondCorp was the importance of phasing a project and continuing to refine and develop our strategies as we encountered additional use cases. While this article focuses on Google's specific experience, the lessons it shares can be adopted at any organization, regardless of size, so long as the effort has solid backing from relevant stakeholders.

Acknowledgments
Thanks to the following people for their help in making this article possible: Heather Adkins, Jeff Baird, Darren Bilby, John Brady, Victor Escobedo, Cynthia Horiguchi, Michael Janosko, Rob Peasegood, Dan Polsby, Val Stiris, and Rory Ward.

References

[1] R. Ward and B. Beyer, "BeyondCorp, A New Approach to Enterprise Security," *;login:*, vol. 39, no. 6 (December 2014), pp. 6–11: https://www.usenix.org/system/files/login/articles/login_dec14_02_ward.pdf.

[2] B. Osborn, J. McWilliams, B. Beyer, and M. Saltonstall, "BeyondCorp: Design to Deployment at Google," *;login:*, vol. 41, no. 1 (Spring 2016), pp. 28–35: https://www.usenix.org/publications/login/spring2016/osborn.

[3] L. Cittadini, B. Spear, B. Beyer, and M. Saltonstall, "BeyondCorp Part III: The Access Proxy," *;login:*, vol. 41, no. 4 (Winter 2016), pp. 28–33: https://www.usenix.org/publications/login/winter2016/cittadini.

[4] Capirca is a tool designed to utilize common definitions of networks, services, and high-level policy files to facilitate the development and manipulation of network access control lists: github.com/google/capirca.

Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data

TYLER HUNT, ZHITING ZHU, YUANZHONG XU, SIMON PETER, EMMETT WITCHEL

Tyler Hunt is a PhD student at the University of Texas at Austin, working with Emmett Witchel. His research interests involve designing and building systems with interesting security properties. thunt@cs.utexas.edu

Zhiting Zhu has been a PhD student at the University of Texas at Austin since 2014, where he works with Emmett Witchel. He is interested in operating systems. zhitingz@cs.utexas.edu

Yuanzhong Xu received his PhD in computer science from the University of Texas at Austin in 2016. He is generally interested in systems and security. He currently works for Facebook as a research scientist. yxu@utexas.edu

Simon Peter is an Assistant Professor at the University of Texas at Austin, where he conducts research in operating systems and networks. He received a PhD in computer science from ETH Zurich in 2012 and an MSc in computer science from the Carl von Ossietzky University of Oldenburg, Germany, in 2006. Before joining UT Austin in 2016, he was a Research Associate at the University of Washington from 2012 to 2016. simon@cs.utexas.edu

Emmett Witchel is a Professor in Computer Science at the University of Texas at Austin. He received his doctorate from MIT in 2004. He and his group are interested in operating systems, security, and concurrency. witchel@cs.utexas.edu

Ryoan provides a distributed sandbox, leveraging hardware enclaves (e.g., Intel's software guard extensions (SGX)) to protect sandbox instances from potentially malicious computing platforms. The protected sandbox instances confine untrusted data-processing modules to prevent leakage of the user's input data. Ryoan is designed for a request-oriented data model, where confined modules only process input once and do not persist state about the input. We present the design and prototype implementation of Ryoan and evaluate it on a series of challenging problems, including email filtering, health analysis, image processing, and machine translation.

Data-processing services are widely available on the Internet. Individual users can conveniently access them for tasks, including image editing (e.g., Pixlr), tax preparation (e.g., TurboTax), data analytics (e.g., SAS OnDemand), and even personal health analysis (e.g., 23andMe). However, user inputs to such services, such as tax documents and health data, are often sensitive, which creates a dilemma for the user. In order to leverage the convenience and expertise of these services, she has to disclose sensitive data to them, potentially allowing them to disclose the data to third parties. If she wants to keep her data secret, she either has to give up using the services or hope that they can be trusted—that their service software will not leak data (intentionally or unintentionally), and that their administrators will not read the data while it resides on the server machines.

Companies providing data-processing services for users often wish to outsource part of the computation to third-party cloud services, a practice called "software as a service (SaaS)." For example, 23andMe may choose to use a general-purpose machine learning service hosted by Amazon. SaaS encourages the decomposition of problems into specialized pieces that can be assembled on behalf of a user, e.g., combining the health expertise of 23andMe with the machine learning expertise and robust cloud infrastructure of Amazon. However, 23andMe now finds itself a user of Amazon's machine learning service and faces its own dilemma—it must disclose proprietary correlations between health data and various diseases in order to use Amazon's machine learning service. In these scenarios, the owner of secret data has no control over the data-processing service.

We propose Ryoan [1], a distributed sandbox that forces data-processing services to keep user data secret, without trusting the service's software stack, developers, or administrators. Ryoan's name is inspired by a famous dry landscape Zen garden that stimulates contemplation (Ryōan-ji). First, Ryoan provides a sandbox to confine individual data-processing modules and prevent them from leaking data; second, it uses trusted hardware to allow a remote user to verify the integrity of individual sandbox instances and protect their execution; third, the sandbox can be configured to allow confined code modules to communicate in controlled ways, enabling flexible delegation among mutually distrustful parties. Ryoan gives a user confidence that a service has protected her secrets.

A key enabling technology for Ryoan is hardware enclave-protected execution (e.g., Intel's Software Guard Extensions (SGX) [2]), a new hardware primitive that uses trusted hardware to protect a user-level computation from potentially malicious privileged software. The processor hardware keeps unencrypted data on chip but encrypts data when it moves into RAM. The hypervisor and operating system retain their ability to manage memory (e.g., move memory pages onto secondary storage), but privileged software sees only an encrypted version of the data that is protected from tampering by a cryptographic hash. Haven [3] and SCONE [4] are examples of systems that use enclaves to protect a user's computation from potentially malicious system software, including a library operating system to increase backward compatibility.

Ryoan faces issues beyond those faced by enclave-protected computation systems such as Haven. Enclaves are intended to protect an application that is trusted by the user, which does not collude with the infrastructure, though it may unintentionally leak data via side channels. In Ryoan's model the application and the infrastructure are under the control of an adversary and may collude to steal the user's secrets. Even if the application itself is isolated from the infrastructure using enclave protection, the adversary could exercise its control to construct covert channels between the application and the platform. Ryoan's goal is to prevent such covert channels and stop an untrusted application from intentionally and covertly using users' data to modulate events like system call arguments or I/O traffic statistics, which are visible to the infrastructure.

An untrusted application in Ryoan is confined by a trusted sandbox. For the Ryoan prototype we use Native Client (NaCl) [5, 6], which is a state-of-the-art user-level sandbox. NaCl can be built as a standalone binary independent from the browser. NaCl uses compiler-based techniques to confine untrusted code rather than relying on address space separation, a property necessary to be compatible with SGX enclaves. The Ryoan sandbox safeguards secrets by controlling explicit I/O channels, as well as covert channels such as system call traces and data sizes.

The Ryoan prototype uses SGX to provide hardware enclaves. Each SGX enclave contains a NaCl sandbox instance that loads and executes untrusted modules. The NaCl instances communicate with each other to form a distributed sandbox that enforces strong privacy guarantees for all participating parties—the users and different service providers. Confining untrusted code [7] is a longstanding problem that remains technically challenging, but Ryoan benefits from hardware-supported enclave protection. Ryoan also assumes a request-oriented data model, where confined modules only process input once and cannot read or write persistent storage after they receive their input. This model makes Ryoan applicable only to request-oriented server applications—but such servers are the most common way to bring scalable services to large numbers of users.

Ryoan's security goal is simple: prevent leakage of secret data. However, confining services over which the user has no control is challenging without a centralized trusted platform. We make the following contributions:

- A new execution model that allows mutually distrustful parties to process sensitive data in a distributed fashion on untrusted infrastructure.
- The design and implementation of a prototype distributed sandbox that confines untrusted code modules (possibly on different machines) and enforces I/O policies that prevent leakage of secrets.
- Several case studies of real-world application scenarios to demonstrate how they benefit from the secrecy guarantees of Ryoan, including an image processing system, an email spam/virus filter, a personal health analysis tool, and a machine translator.
- Evaluation of the performance characteristics of our prototype by measuring the execution overheads of each of its building blocks: the SGX enclave, confinement, and checkpoint/rollback. The evaluation is based on both SGX hardware and simulation.

Background and Threat Model

Ryoan assumes a processor with hardware-protected enclaves, e.g., Intel's SGX-enabled Skylake (or later) architecture. The address space of a protected enclave has its privacy and integrity guaranteed by hardware. Hardware encrypts and hashes memory contents when it moves off chip, protecting the contents from other users and also from the platform's privileged software (operating system and hypervisor). Code within an enclave can manipulate user secrets without fear of divulging them to the underlying execution platform. Code within an enclave cannot have its code or control manipulated by the platform's privileged software.

SGX's security guarantees are ideal for Ryoan's distributed NaCl-based sandbox. The sandbox confines the code it loads, ensuring that the code cannot leak secrets via storage, network, or other channels provided by the underlying platform. Ryoan instances communicate with each other using secure TLS connections. By collecting SGX measurements and by providing trusted initialization code, Ryoan can demonstrate to the user that their processing topology has been set up correctly.

Threat Model

We consider multiple, mutually distrustful parties involved in data-processing services. A service provider is not trusted by the users of the service to keep data secret; if the service provider outsources part of the computation to other services, it becomes a user of them and does not trust them to provide secrecy, either. Each service provider can deploy its software on its own computational platform, or it can use a third-party cloud platform that is mutually distrustful of all service providers. We assume that users and providers trust their own code and platform but do not trust each other's code or platforms. Everyone must trust Ryoan and SGX.

A service provider might be the same as its computational platform provider, and the two might collude to steal secrets from their input data. Besides directly communicating data, untrusted code may use covert channels via *software interfaces,* such as syscall sequences and arguments, to communicate bits from the user's input to the platform.

A user of a service does not trust the software at any privilege level in the computational platform. For example, the attacker could be the machine's owner and operator, a curious or even malicious administrator, an invader who has taken control of the operating system and/or hypervisor, the owner of a virtual machine physically co-located with the VM being attacked, or even a developer of the untrusted application or OS writing code to directly record user input.

Although we consider covert channels based on software interfaces like system calls, we do not consider side or covert channels based on *hardware limitations* or *execution time*. Untrusted enclaves can leak bits by modulating their cache accesses, page accesses, execution time, etc. While we do not claim to prevent the execution-time channel, Ryoan does limit the use of this channel to once per request.

Intel Software Guard Extensions

Software Guard Extensions (SGX), available in new Intel processors, allow processes to shield part of their address space from privileged software. Processes on an SGX-capable machine may construct an *enclave,* which is an address region whose contents are protected from all software outside of the enclave via encryption and hashing. Code and data loaded into enclaves, therefore, can operate on secret data without fear of unintentional disclosure to the platform. These guarantees are provided by the hardware [2].

SGX provides attestation of enclave identity, which for Ryoan is a hash of the enclave's initial state, that is, memory contents and permissions offset from the enclave base address. Ryoan assumes that the initial state of an enclave cannot be impersonated under standard cryptographic assumptions. Ryoan uses SGX to attest that all enclaves have the same initial state and thus the same identity. Ryoan loads service provider code after it initializes. Before the service code is loaded and before passing sensitive data to Ryoan, a user will request an attestation from SGX and verify the identity of the enclave.

Enclave code may access any part of the address space which does not belong to another enclave. Enclave code does not, however, have access to all x86 features. All enclave code is unprivileged (ring 3), and any instruction that would raise its privilege results in a fault.

Hardware security limitations

Enclaves on modern Intel processors have security limitations including page faults [8], cache timing, address bus monitoring, and the information exposed by processor monitoring units. We believe these limitations must be addressed independently from Ryoan, and we hope they will be. Each of these limitations compromise Ryoan's security goals. If there are other hardware limitations, they also must be addressed independently from Ryoan. Part of the purpose in constructing the Ryoan prototype is to demonstrate the importance of addressing these hardware-based information leaks.

Native Client

Google Native Client (NaCl) is a sandbox for running x86/x86-64 native code (a NaCl module) using software fault isolation. NaCl consists of a verifier and a service runtime. To guarantee that the untrusted module cannot break out of NaCl's software-based fault isolation sandbox, the verifier disassembles the binary and validates the disassembled instructions as being safe to execute.

NaCl executes system calls on behalf of the loaded application. System calls in the application transfer control to the NaCl runtime which determines the proper action. Ryoan cannot allow the application to use its system calls to pass information to the underlying operating system. For example, if Ryoan passed read system calls from the application directly to the platform, the application could use the size and number of the calls to encode information about the secret data it is processing. We discuss the details of the confinement provided by Ryoan in the section "Ryoan's Confined Environment," below.

Design

Ryoan is a distributed sandbox that executes a directed acyclic graph (DAG) of communicating untrusted modules which operate on sensitive data. Ryoan's primary job is to prevent the modules from communicating any of the sensitive data outside the confines of the system, including external hosts and the platform's privileged software.

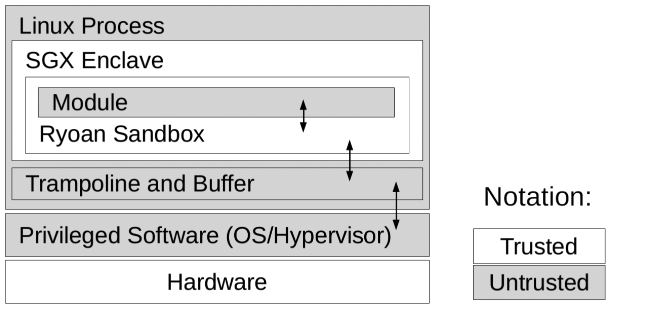


Figure 1: A single instance of Ryoan's distributed sandbox. The privileged software includes an operating system and an optional hypervisor.

Ryoan prevents modules from leaking sensitive data by decoupling externally visible behaviors from the content of secret data. SGX hardware limits externally visible behaviors to explicit stores to unprotected memory and use of system services (syscalls).

Unprotected stores are eliminated by the NaCl tool chain and runtime. Ryoan mostly eliminates system calls by providing their functionality from within NaCl. For example, Ryoan provides `mmap` functionality by managing a fixed-sized memory pool within the SGX enclave. However, untrusted modules must read input and write output, so Ryoan provides a restricted I/O model that prevents data leaks: for example, the output size is a fixed function of input size. A module cannot communicate the contents of the input data by changing the size of the output.

Figure 1 shows a single instance of the Ryoan distributed sandbox. A principal—for example, a company providing software as a service—can contribute a module which Ryoan loads and confines, enabling the module to safely operate on secret data. A module consists of code, initialized data, and the maximum size of dynamically allocated memory. The NaCl sandbox uses a load-time code validator to ensure that the module cannot violate the sandbox by reaching outside of its address range or making syscalls without Ryoan intervention.

Ryoan executes inside of hardware-protected enclaves and does not trust the operating system nor the hypervisor. SGX generates an unforgeable remote attestation for the user that a Ryoan instance is executing in an enclave on the platform. The user can establish an encrypted channel that she knows terminates within that Ryoan instance. SGX guarantees the enclave cryptographic secrecy and integrity against manipulation by privileged software.

Enforcing Topology

The user either defines the communication topology of confined modules or explicitly approves it. A topology is a DAG of modules with unidirectional links. The DAG specification is first passed to an initial enclave which we call the *master*. The master contains standard, trusted initialization code provided by Ryoan. The master requests that the operating system start enclaves that contain Ryoan instances for modules listed in the specification. These enclaves can be hosted on different machines. The master uses SGX to perform local or remote attestation to verify the validity of individual Ryoan enclaves, then lets neighbor enclaves in the DAG establish cryptographically protected communication channels via key exchange using the untrusted network or local inter-process communication as a transport. The user can verify the validity of the master via attestation and ask it whether a desired topology has been initialized. After that, the user establishes secure channels with the entry and exit enclaves of the DAG and starts data processing.

Figure 2 shows an example of Ryoan processing input from user Alice whose sensitive data is processed by both 23andMe and Amazon. Each Ryoan instance executes in an enclave on the same or different machines. The host machine(s) might be provided by 23andMe, Amazon, or a third party. In all cases, Ryoan ensures no leakage of the user's secrets and also prevents leakage of any trade secrets used by 23andMe and Amazon.

Data-Oblivious Communication

One of the primary safety functions of Ryoan is to prevent the computational platform from inferring secrets about the input data by observing data flow among modules. Therefore, data flow must be independent from the contents of the input data: Ryoan never moves data in response to activity under the control of the untrusted module once the module has read its input data. This safety property is sometimes called being data oblivious [9].

Units of work can be any size, but Ryoan ensures that data flow patterns do not leak secrets from input data by making module output size a fixed, application-defined function of the input size. Ryoan protects communication with the following rules: (1) each Ryoan instance reads its entire input from every input-connected Ryoan instance before the module starts processing; (2) the size of the output is a polynomial function of the input size, specified as part of the DAG, and Ryoan pads/truncates all outputs to the exact length determined by the polynomial and the size of the input; (3) Ryoan is notified by the module when its output is complete, and it writes the module's output to all output-connected Ryoan instances. Ryoan encapsulates module output in a message that contains metadata which describes what is module output and what is padding (if any). The metadata is interpreted, and any padding is stripped away by the next Ryoan instance before exposing the data to its module. Each Ryoan instance must receive the complete input of a work unit before executing its module. These rules are sufficient because they ensure that output traffic is independent from input data

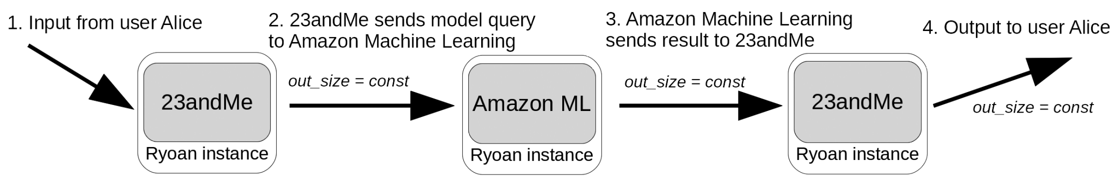


Figure 2: Ryoan's distributed sandbox. In this example, the application spans the administrative domains of 23andMe and Amazon.

(though there are possible alternatives—for example, each request could specify its output size).

Consider the scenario in Figure 2. Each input comes from a user. The user can choose to leak the size of the input, or he can hide the size by padding the input. The description of the DAG specifies that (1) the output of 23andMe's first module is padded to a fixed size, defined by 23andMe, which can hold the largest possible model query; (2) the output of Amazon Machine Learning's classifier module is padded to a fixed size to encode the classification result; and (3) the response to the user from 23andMe's second module is also padded to a fixed size that can hold the largest possible result.

Ryoan's Confined Environment

Any module with access to user data is executed in Ryoan's confined environment, which prevents information leakage while reducing porting effort. When a module receives the secret data contained within a request, it enters the confined environment and loses the ability to communicate with the untrusted OS via any system call. Therefore, Ryoan must provide a system API sufficient for most legacy code to function properly. To reduce porting effort, Ryoan provides an *in-memory virtual file system* and supports anonymous memory mappings from a pre-allocated memory region to support module dynamic memory.

Ryoan's confined environment is sufficient for many data-processing tasks. For example, ClamAV, a popular virus scanning tool, loads the entire virus database during initialization; when scanning the input such as a PDF file, ClamAV creates temporary files to store objects extracted from the PDF. Ryoan's in-memory file system satisfies these requirements.

Module Life Cycle

A Ryoan instance enforces the following life cycle on its module: creation, initialization, wait, process, output, destruction/reset. The sandbox begins by validating its module and verifying that its identity matches the DAG specification. The instance allows the module to initialize with full access to the system services exposed by vanilla NaCl. Nonconfined initialization makes module creation more efficient and makes porting easier because initialization code can remain unchanged.

Modules signal Ryoan when initialization is complete by calling `wait_for_work`, a routine implemented by Ryoan. Once a module is initialized, the module processes a request, generates its output, and then is destroyed or reset to prevent accumulating secret data. Ryoan instances are request oriented: input can be any size, but each input is an application-defined "unit of work." For example, a unit of work can be an email when classifying spam or a complete file when scanning for viruses. Each module gets a single opportunity to process its input data.

Checkpoint-Based Enclave Reset

Creating and initializing modules often requires far more CPU time than processing a single request. For instance, loading the data necessary for virus scanning takes 24 seconds; orders of magnitude greater than the ≈0.124 seconds it takes to process a single email. Ryoan manages the module life cycle efficiently using checkpoint-based enclave reset.

Ryoan provides a checkpoint service that allows the application to be rolled back to an untainted, but initialized, memory state (Figure 3). In our prototype this state is at the first invocation of `wait_for_work`. Ryoan does not allow an enclave that has seen secret input to be checkpointed, because its data model is request-oriented: modules should not depend on past requests to operate. Checkpointing a module that has seen secret data would (potentially) give that module multiple execution opportunities on a single request's data.

Checkpoint restore allows Ryoan to save the cost of tearing down and rebuilding the SGX enclave, and it saves the cost of executing the application's initialization code. Ryoan checkpoints are taken once but restored after each request is processed. Therefore, Ryoan makes a full copy of the module's writable state and simply tracks which pages get modified, avoiding a memory copy during processing. Only the contents of pages that were modified during input processing are restored. SGX provides a way for enclave code to verify page permissions and be reliably notified about memory faults, which is necessary to track which pages are written.

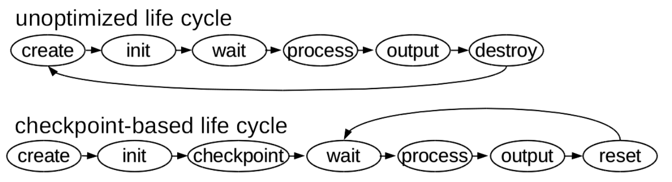


Figure 3: Instance life cycle: unoptimized vs. checkpoint based

Use Cases

This section explains four scenarios where Ryoan provides a previously unattainable level of security for processing sensitive data. For all examples, the Ryoan instances could execute on the same platform or on different platforms, e.g., the entire computation might execute on a third-party cloud platform like Google Compute Engine, or a provider's module might execute on its own server. Ryoan's security guarantees apply to all scenarios.

Email Processing

A company can use Ryoan to outsource email filtering and scanning while keeping email text secret. We consider spam filtering and virus scanning, using popular legacy applications—DSPAM 3.10.2 and ClamAV 0.98.7. The computation DAG for this service contains four Ryoan instances, each confining a data processing module (see Figure 4). An email arrives at the entry enclave over a secure channel. The entry enclave simply distributes the email text and attachments to the enclaves containing DSPAM and ClamAV, respectively. The results of virus scanning and spam filtering are sent to a final post-processing enclave, which constructs a response to the user over a secure channel.

Personal Health Analysis

Consider a company (e.g., 23andMe) that provides customized health reports for users based on a variety of health data. 23andMe accepts a user's genetic data, medical history, and physical activity log as input, extracts important health features from these data, and predicts the likelihood of certain diseases.

Secrecy for both users and 23andMe is protected with a DAG (see Figures 2 and 4). Amazon provides the classifier, which queries a model as a Ryoan module. Users provide their genetic information, medical history, and activity log in a request. Upon receiving a user's request, 23andMe's first module constructs a Boolean vector of health features and forwards it to Amazon's module. Amazon's module generates predictions based on the model and forwards the result to 23andMe's second enclave, which then forwards the result back to the user.

Image Processing

Image classification as a service is an emerging area that could benefit from Ryoan's security guarantees. We envision a scenario where a user wants different image classification services based on her expertise. For example, one service might be known for accurate identification of adult content while another might do an excellent job recognizing and segmenting horses. The image processing DAG in Figure 4 shows an example where an image filtering service outsources different subtasks to different providers and then combines the results. Our prototype implements all of these detection tasks using OpenCV 3.1.0, and each detection task loads a model that is specialized to the detection task and would represent a company's competitive advantage.

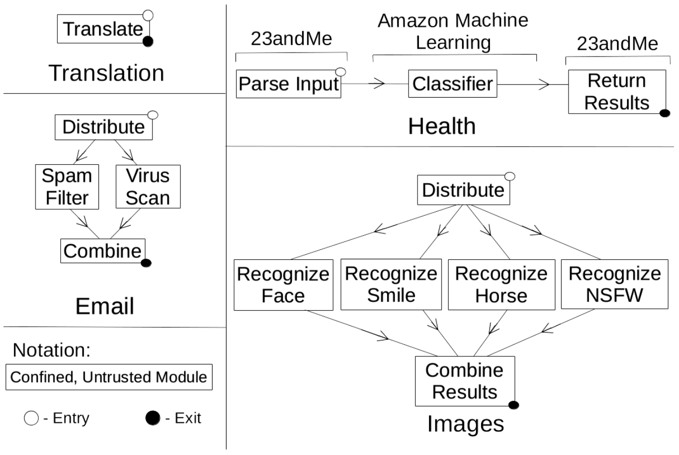


Figure 4: Topologies of Ryoan example applications. Nodes in the graph are Ryoan instances, though we identify them by their untrusted module. Users establish secure channels with trusted Ryoan code for the source and sink nodes to provide input and get output, respectively.

Translation

A company uses Ryoan to provide a machine translation service while keeping the uploaded text secret. Users upload text to the translation enclave and get the translated text back. Our prototype uses Moses, a statistical machine translation system. We train a phrase-based French to English model using the News Commentary data set released for the 2013 workshop in machine translation [10].

Evaluation

We evaluated Ryoan's overhead on realistic workloads for each of these use cases. Slowdowns range from 27% to 419%. The Ryoan prototype relies on some unreleased SGX features. Therefore, our evaluation involves an SGX performance model where applicable. For evaluation details see the original publication [1].

Conclusion

Ryoan allows users to safely process their secret data with software they do not trust, executing on a platform they do not control, thereby benefiting users, data processing services, and computational platforms.

References

[1] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pp. 533–549: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/hunt.

[2] Intel(R) Software Guard Extensions Programming Reference, 2014: https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[3] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pp. 267–283: https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-baumann.pdf.

[4] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pp. 689–703: https://www.usenix.org/system/files/conference/osdi16/osdi16-arnautov.pdf.

[5] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted X86 Native Code," in *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009, pp. 79–93: http://regmedia.co.uk/2008/12/09/native_client_paper.pdf.

[6] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting Software Fault Isolation to Contemporary CPU Architectures," in *Proceedings of the 19th USENIX Security Symposium (USENIX Security '10)*, pp. 1–11: https://www.usenix.org/legacy/event/sec10/tech/full_papers/Sehr.pdf.

[7] B. W. Lampson, "A Note on the Confinement Problem," *Communications of the ACM*, vol. 16, no. 10, October 1973.

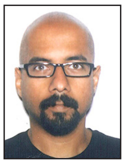
[8] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2015, pp. 640–656: http://www.ieee-security.org/TC/SP2015/papers-archived/6949a640.pdf.

[9] O. Ohrimenko, F. Schuster, C. Fournet, S. Nowozin, A. Mehta, K. Vaswani, and M. Costa, "Oblivious Multi-Party Machine Learning on Trusted Processors," in *Proceedings of the 25th USENIX Security Symposium (USENIX Security '16)*, pp. 619–636: https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_ohrimenko.pdf.

[10] Shared Task: Machine Translation: http://www.statmt.org/wmt13/translation-task.html.

Securing Software Updates for Automotives Using Uptane

TRISHANK KARTHIK KUPPUSAMY, LOIS ANNE DELONG, AND JUSTIN CAPPOS



Trishank Karthik Kuppusamy is a fifth-year PhD student at the NYU Tandon School of Engineering, where he works with Professor Justin Cappos on the design and deployment of software update security systems. He led the specification for Uptane, a new technology to secure software updates for automotives. trishank@nyu.edu

Lois Anne DeLong is a Research Associate and Technical Writer for the Secure Systems Lab at NYU Tandon School of Engineering. She has served as a writer and editor for technical journals, and has also taught technical writing and basic composition courses. lad278@nyu.edu

Justin Cappos is an Assistant Professor at NYU in the Tandon School of Engineering. Justin's research interests focus on improving the security of real-world systems in a variety of practical applications. His more recent work on software updaters has been standardized by the Python community and deployed by Docker. jcappos@nyu.edu

Does secrecy improve security or impede securing software updates? The automotive industry has traditionally relied upon proprietary strategies developed behind closed doors. However, experience in the software security community suggests that open development processes can find flaws before they can be exploited. We introduce Uptane, a secure system for updating software on automobiles that follows the open door strategy. It was jointly developed with the University of Michigan Transportation Research Institute (UMTRI), and the Southwest Research Institute (SWRI), with input from the automotive industry as well as government regulators. We are now looking for academics and security researchers to attempt to break our system before black-hat hackers do it in the real world—with possibly fatal consequences.

Security Should Not Be a Competitive Advantage

Imagine that you get into your car and turn on the ignition, but the engine does not start. You turn the key again, but the only sound you hear is the automatic door locks closing. After a few more futile attempts to start the car—and to open the doors—you notice a message on the screen of your infotainment system: "$500 in Bitcoin if you want to get out of your car." A hacker has just exploited a security flaw in the system used to deliver software updates to one of your car's on-board computing units, and the result is this simple but effective cyber-attack. We need your help in preventing this scenario from happening in the real world.

Presently, vehicle manufacturers purchase proprietary software update systems from third-party suppliers. This helps to keep costs competitive, because a manufacturer need not worry about developing its own system. These systems are proprietary in nature, and, thus, their security guarantees are unclear. A manufacturer may not even have access to the source code used in parts created by one of their suppliers. What is known is that these systems have been hacked repeatedly [1–3]. At a time when computing units continue to proliferate on vehicles, and where the cost of security flaws in code can be measured in human lives, many manufacturers still follow the design principle of security by obscurity, which has resulted in a substantial number of successful attacks.

We strongly believe that the security of your car should not be based upon which supplier can market their solution best to the car companies. It would not be a desirable outcome for a manufacturer or supplier to advertise that compromises of their software update system only harmed hundreds of people, while their competitors' compromises harmed thousands. Open security reviews have been used time and time again in the design of critically important systems, such as cryptographic algorithms, anti-censorship software, and secure software update systems. Designing software systems in a more open manner can benefit manufacturers, suppliers, and the public simultaneously.

Uptane, a new, secure software update system, is a direct product of such an open process. Uptane was designed in collaboration with major vehicle manufacturers and suppliers responsible for 78% of vehicles on US roads, as well as government regulators. We have shared technical documents and a reference implementation to aid manufacturers and suppliers to build, customize, and deploy their own variants of this system. A supplier has begun selling a product that includes Uptane, and a few others are integrating it as we speak. As adoption grows, we are looking to the open source community to give our code a test drive. We welcome white-hat hackers to try to break Uptane and to give us feedback before you, and millions of others, are betting your life on its security.

A Quick Primer on Computers in Vehicles

While most people think of a car as a collection of mechanical parts such as the engine, door locks, and brakes, a modern vehicle is actually a sophisticated container for a collection of microcomputers called *electronic control units* (ECUs). Like any other computer, these ECUs are responsible for executing specific functions, from tightening a seat belt during an accident to adjusting a passenger side mirror. Where ECUs differ from traditional computers is in how heterogeneous their computational speed, memory, and network capabilities are. For example, some ECUs, such as the telematics or infotainment units, have general-purpose CPUs with high speed, large memory, and a wireless connection to the outside world, whereas other ECUs, such as the seat belt pretensioner ECU, use specialized CPUs with low speed, small memory, and no external network connection.

An *original equipment manufacturer* (OEM), such as Ford or General Motors, chooses the ECUs that will reside on a vehicle model. However, these units are usually produced by third-party suppliers, such as Bosch or Lear. The software for an ECU is maintained by its supplier and delivered to the OEM to be distributed to vehicles.

To distribute software updates, the OEM maintains a software *repository*, which hosts and distributes images and metadata. An *image* is a self-contained archive of code and/or data required for an ECU to function. *Metadata* is information about images or other metadata files. Typically, this metadata lists the cryptographic hashes and file sizes of images.

This metadata should be signed, using well-protected keys, so that attackers cannot tamper with images without being detected. However, some manufacturers and suppliers do not provide signed metadata about images. As a result, ECUs can be reflashed over the network if attackers know the fixed challenge-response algorithm used to unlock ECUs. Although these fixed algorithms are supposed to be secret, they are known by the car tuning community [1, 2]. To take another example, Tesla did not, to the best of our knowledge, sign its images at all until security researchers used a wireless connection to rewrite software on its ECUs and exert physical control over its vehicles [3]. Although it is important to sign metadata, the security of ECUs depends on precisely how it is signed.

Existing Software Update Systems Do Not Fit the Automotive Industry

Existing software update systems force an unacceptable tradeoff upon OEMs. To achieve maximum security, they often have to sacrifice the customizability that allows them to offer different images to different vehicles. On the other hand, other systems offer customizability but no security when attackers have compromised the repository itself.

Some security systems use *online keys*, or signing keys that are accessible from the repository, to sign metadata, protecting ECUs from man-in-the-middle attacks. For example, these systems may use the SSL/TLS or CUP transport protocol to sign images and metadata in transit. The upside of using an online key is that it allows *on-demand customization of vehicles*, an attribute that was considered very important by our industry collaborators for various legal and technical reasons.

Unfortunately, the downside of using an online key to sign all metadata is that attackers who compromise the repository can also immediately abuse this key to sign and distribute malware. This is true even if the online key is protected behind a Hardware Security Module (HSM).

To solve this problem, some security systems use *offline keys*, or signing keys that are not accessible from the repository, to sign all metadata. These systems may use, for example, the PGP/GPG or RSA cryptographic schemes for this purpose. The upside of using offline keys is that it provides *compromise-resilience*: attackers who compromise the repository are unable to tamper with images without being detected. In practice, however, it is typically a precarious form of compromise-resilience, because often a single offline key is used to sign all metadata.

Unfortunately, the downside of using only offline keys to sign all metadata is that we have lost on-demand customization of vehicles. This is because the repository cannot dynamically respond to fresh information that indicates what is currently installed on a vehicle and decide what should be installed next.

Besides the on-demand customization of vehicles, there are other critical constraints in designing a secure software update system for automotives. Above all else, the system must be simple for manufacturers and suppliers to implement, customize, and deploy. Another important constraint is that ECUs are often limited by speed, memory, or network connection. Many

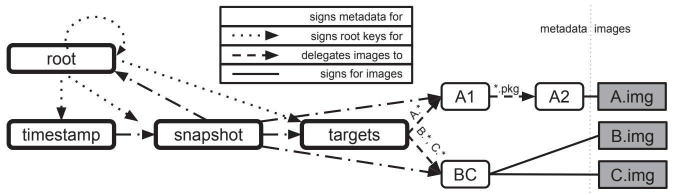


Figure 1: Separation of duties between roles on a compromise-resilient repository

ECUs are highly optimized for a specific function in order to keep costs low. Thus, many ECUs may not have enough storage space to maintain a large amount of metadata, may not have a direct network connection to the repository, and may not be able to compute or verify a signature in a reasonable amount of time.

Uptane: A New, Secure Software Update System

Uptane is a new, secure software update system that is specifically designed to solve problems in the automotive domain [4]. The key idea is to use two repositories, one to provide compromise-resilience and the other to provide on-demand customization of vehicles.

Uptane uses four design principles that help to achieve compromise-resilience [5, 6]. First, different types of metadata are signed using different keys, so that the impact of a key compromise is minimized and does not necessarily affect the security of the whole system. As illustrated in Figure 1, and summarized in Table 1, there are four top-level roles on a repository: the root, timestamp, snapshot, and targets roles. Second, a threshold number of signatures may be required to sign a metadata file, so that a single key compromise is insufficient to publish malicious images. Third, there must be a way to revoke keys when they are compromised. Keys can be revoked explicitly by publishing new keys to replace old ones, or they can be revoked implicitly by setting expiration timestamps in metadata files. Finally, use of offline keys can minimize the risk of a key compromise for high-value roles whose compromise can lead to malicious images.

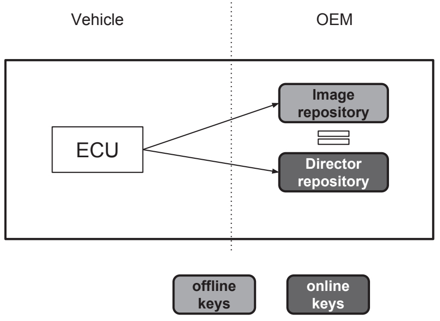


Figure 2: Using two repositories to provide both compromise-resilience and on-demand customization of vehicles

On the *image repository*, offline keys are used to sign all metadata about all images for all ECUs on all vehicles manufactured by the OEM. Metadata for the top-level roles are signed by the OEM's administrators. The OEM may delegate the signing of images to their respective suppliers, or it may sign them itself. This repository provides compromise-resilience but not on-demand customization of vehicles.

The *director repository* instructs vehicles on what should be installed next, given information about what they have currently installed. This repository uses online keys to sign fresh timestamp, snapshot, and targets metadata for each vehicle that indicate which images from the image repository should be installed next.

As depicted in Figure 2, vehicles install images only if both repositories agree on their contents. That is, the contents of images chosen for installation by the director repository must match the contents of the same images available on the image repository. Since the director repository has more complicated functionality, it is more likely to contain vulnerabilities that can be remotely exploited, and thus compromised. By separating both repositories, we are able to prevent attackers who compromise one repository from being able to distribute malicious images.

Role	Responsibilities
Root	The **root** role is the locus of trust. It indicates which keys are authorized for the **targets, snapshot,** and **timestamp** roles. It also lists the keys for the **root** role itself.
Targets	The **targets** role provides crucial metadata about images, such as their hashes and lengths. This role may delegate the signing of images to their respective suppliers.
Snapshot	The **snapshot** role indicates the latest versions of all metadata on the repository. This prevents an ECU from installing outdated images.
Timestamp	The **timestamp** role is responsible for indicating if images or metadata have changed.

Table 1: A summary of responsibilities of the top-level roles on a repository

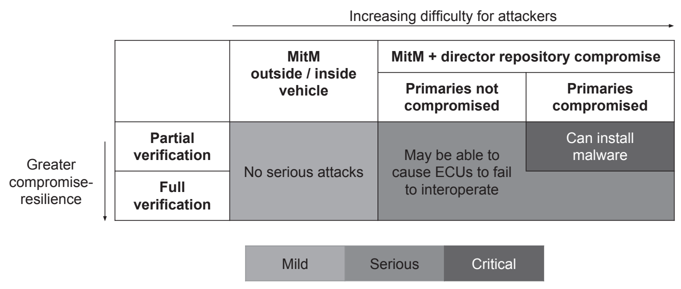

Increasing difficulty for attackers
MitM outside / inside vehicle
MitM + director repository compromise
Primaries not compromised
Primaries compromised
Greater compromise-resilience
Partial verification
Full verification
No serious attacks
May be able to cause ECUs to fail to interoperate
Can install malware
Mild
Serious
Critical

Figure 3: A brief security analysis of ECUs using Uptane, depending on which repositories and ECUs attackers have compromised

There are two types of ECUs. A *primary* downloads, verifies, and distributes images and metadata to secondaries. A *secondary* receives them from a primary, and installs a new image only if it has been successfully verified against the signed metadata.

There are two types of metadata verification designed to accommodate ECUs with different security and cost requirements. *Full verification* requires checking that the images chosen for installation by the director repository match the same images on the image repository. Primaries always perform full verification in order to protect secondaries from security attacks. *Partial verification* requires checking only that the signatures from the director repository are valid.

A brief security analysis is illustrated in Figure 3. The difference between ECUs that perform full and partial verification is in how resilient they are against a repository compromise. When there are only man-in-the-middle attacks but no key compromise, attackers do not pose a serious threat. When attackers have compromised the director repository, there are two cases: primaries that have been compromised and primaries that have not.

If attackers have not compromised primaries, then they may be able to cause both types of ECUs to fail to interoperate. This is because attackers can control which images are installed on which ECUs. However, it is possible to limit the attackers' choices by including metadata that prevent ECUs from installing incompatible or conflicting images. Nevertheless, they cannot install malicious updates, because primaries always perform full verification on behalf of secondaries.

However, attackers that have compromised primaries can install malicious updates, but only on partial verification ECUs. Attackers cannot install malicious updates on full verification ECUs, even if they have also compromised the image repository, because they must also compromise offline keys.

In summary, Uptane offers basic security guarantees for all ECUs and greater compromise-resilience for ECUs that can afford additional computation and storage space. In addition, by separating concerns over multiple repositories, Uptane also provides on-demand customization of vehicles.

A Call to Action

We believe that Uptane provides the strongest solution to a real-world problem, without sacrificing usability and flexibility. However, we do not know of a better way to guarantee the security of any system than subjecting it to a critical, rigorous, and open review. We want you to scrutinize Uptane and find any design flaws before the black-hat hackers use them against us. You can drop us comments on our Google Docs or report issues and send pull requests on our GitHub projects. To do so, please visit our Web site at https://uptane.github.io.

Acknowledgments and Disclaimers

Our co-authors include Akan Brown (NYU), Sebastien Awwad (NYU), Damon McCoy (NYU), Russ Bielawski (UMTRI), Sam Weber (NYU), John Liming (SWRI), Cameron Mott (SWRI), Sam Lauzon (UMTRI), and André Weimerskirch (UMTRI/Lear Corporation).

Uptane is supported by US Department of Homeland Security grants D15PC00239 and D15PC00302. The views and conclusions contained herein are the authors' and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the US Department of Homeland Security (DHS) or the US government.

References

[1] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "Experimental Security Analysis of a Modern Automobile," 2010 IEEE Symposium on Security and Privacy: http://www.autosec.org/pubs/cars-oakland2010.pdf.

[2] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, "Comprehensive Experimental Analyses of Automotive Attack Surfaces," in *Proceedings of the 20th USENIX Security Symposium*, 2011: http://www.autosec.org/pubs/cars-usenixsec2011.pdf.

[3] A. Greenberg, "Tesla Responds to Chinese Hack with a Major Security Upgrade," *Wired*, September 27, 2016: https://www.wired.com/2016/09/tesla-responds-chinese-hack-major-security-upgrade/.

[4] T. K. Kuppusamy, A. Brown, S. Awwad, D. McCoy, R. Bielawski, C. Mott, S. Lauzon, A. Weimerskirch, J. Cappos, "Uptane: Securing Software Updates for Automobiles," 14th ESCAR Europe 2016: https://ssl.engineering.nyu.edu/papers/kuppusamy_escar_16.pdf.

[5] J. Samuel, N. Mathewson, J. Cappos, R. Dingledine, "Survivable Key Compromise in Software Update Systems," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10)*, pp. 61–72: https://ssl.engineering.nyu.edu/papers/samuel_tuf_ccs_2010.pdf.

[6] T. K. Kuppusamy, S. Torres-Arias, V. Diaz, and J. Cappos, "Diplomat: Using Delegations to Protect Community Repositories," in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*, pp. 567–581: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/kuppusamy.

Stay ahead—and learn faster—with Safari.

Get a **free 10-day trial** on the world's most comprehensive technology and business learning platform, now featuring live online training.

oreilly.com/safari

O'REILLY® Safari

Blockchain
Hype or Hope?

RADIA PERLMAN

Radia Perlman's work has had a profound impact on how computer networks function today, enabling huge networks, like the Internet, to be robust, scalable, and largely self-managing. She has also made important contributions in network security, assured delete, key management for data at rest encryption, DDoS defense, and user authentication. She is currently a Fellow at Dell EMC, and has taught as adjunct faculty at University of Washington, Harvard, and MIT. She wrote the textbook *Interconnections* and co-wrote the textbook *Network Security*. She holds over 100 issued patents and has received numerous awards, including induction into the Inventor Hall of Fame, lifetime achievement awards from ACM's SIGCOMM and USENIX, election to the National Academy of Engineering, induction into the Internet Hall of Fame, and an honorary doctorate from KTH. She has a PhD in computer science from MIT. radia@alum.mit.edu

In this article, I describe the technology behind Bitcoin's blockchain, and its scalability, security, and robustness. Most of what is written about "blockchain technology" talks about how it will revolutionize all sorts of applications without contrasting it with alternative solutions. To complicate matters, there are all sorts of proposed variants of the original blockchain (the technology behind Bitcoin), making the definition of "blockchain technology" very unclear. I explain how Bitcoin's blockchain technology works, along with its performance implications.

A lot has been written about "blockchain technology" recently, but most of it talks about how it "is being investigated" for various applications and how it is a revolution in computing that will change the world [1]. It is not that easy to discover, from these sorts of articles, how the technology works or what its true properties are. These articles treat "blockchain" as a sort of black box that stores and retrieves data, with certain properties:

- Append-only log
- "Immutable"
- No central point of control

The term *blockchain* was introduced as the name of the technology that powers Bitcoin. Given that Bitcoin's technology is widely deployed and unlikely to change very dramatically, it is possible to describe how it works and what its scalability, robustness, and security properties are. It is not clear how much this system can be modified and still be called *blockchain technology*. Therefore, with the term blockchain technology being less and less well-defined, I will not attempt to describe the properties of every variant proposed, and for the rest of this article, when I say "blockchain," I am referring to Bitcoin's blockchain.

Description of Blockchain
In this section I'll give an overview of the Bitcoin blockchain technology.

Bitcoin
Bitcoin was introduced to the world in a 2008 article [2] and, shortly thereafter, was released as open source software. The concepts are described in the paper, but the details are defined by the implementation. The open source community in control of the software may make changes, but the more widely deployed it is, the more difficult it is to make incompatible changes.

The design goal of Bitcoin was to create a currency that could not be controlled by any government or any known organizations. This design is intended to foil the ability of governments to do things like:

- Enforce tax laws
- Follow a money trail
- Prohibit payments to certain countries or organizations
- Inhibit criminals from anonymously collecting ransom money

These may or may not be desirable goals for a currency, but I will examine the performance implications of a design with these goals, and whether applications other than cryptocurrency really benefit from a design without known entities at the center.

The basic concepts behind blockchain:

- A large (thousands) community of anonymous entities called "miners" collectively agree upon the history of transactions, in an append-only data structure known as "the ledger."
- Users of Bitcoin are not identified with names, but rather, with public keys, and a user is allowed (even encouraged) to change public keys often, to make transactions more anonymous.
- The ledger contains a list of every Bitcoin transaction since Bitcoin was invented.
- A transaction records that a public key X pays a certain amount of Bitcoin to public key Y.
- In order to add transactions to the ledger, a miner must validate the transactions and compute a valid block containing them.
- A valid block contains a hash of the previous block in the blockchain, a set of new valid transactions, and a random number chosen so that the hash of the block meets certain conditions. A valid block is, by design, just hard enough to compute that the collective compute power of the miner community will find a new block at some cadence (about every 10 minutes).
- The miner who is lucky enough to be the first to find the next valid block is awarded with some amount of Bitcoin.

Now I will describe these steps in more detail.

Format of the Ledger

Each block in the blockchain contains the hash of the previous block, a nonce (a random number), the public key of the lucky miner who was the first to find a valid next block, and valid transactions that have not yet been recorded in the ledger (Figure 1).

Transactions

The information in transactions looks like this:

A transaction (with hash T1) consists of the payer (public key X) signing away all of the Bitcoins that X had been paid in some previous transaction (with hash T2).

In order for the transaction T1 to be valid,

- There must be a prior transaction with hash T2, in which X was the payee of the amount of Bitcoin being paid in transaction T1.
- The signature on T1 must properly validate, using public key X.
- There must be no other transaction in the ledger in which X has already spent the proceeds of T2.

There are extra details. For example, notice in the third line of Figure 2 (the transaction with hash x17), A is signing over to C the results of the transaction with hash x15, in which X received 74.92 Bitcoins. But A is only paying 74.21 in transaction x17, even though in transaction x15, A had received 74.92. The difference (74.92 – 74.21) is a transaction fee, paid to the miner who adds a block to the blockchain that contains transaction x17. This rewards the miner for including this transaction in the new block.

The Hash

The mining community imposes conditions on the hash of a valid block. These conditions are designed to be just difficult enough to meet, that it will take the community about 10 minutes to find a block with the appropriate hash.

A good cryptographic hash is like a random number. Given random input, it should have probability 0.5 that the first bit in the hash will be 0, or probability 0.25 that the first two bits would both be 0. The method that blockchain uses to adjust the difficulty of computing the hash is to have a maximum value that the hash must have. Currently, the maximum value of the hash has about 70 leading zeroes. That means that for any block, the probability of its hash having 70 leading 0s is $1/(2^{70})$. Using a brute force search, and the collective compute power of the mining community, it takes about 10 minutes for at least one miner to find a block with a small enough hash. If blocks are found too quickly, then the maximum hash value is adjusted to be smaller. If blocks are found too slowly, then the maximum hash value is adjusted to be larger.

Traditional Integrity Checks vs. Blockchain Hash

Traditional public key cryptography creates digital signatures that can be efficiently computed, if and only if the signer knows

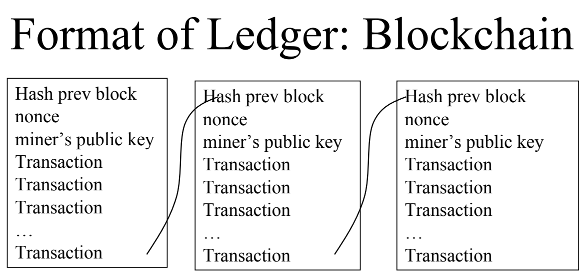
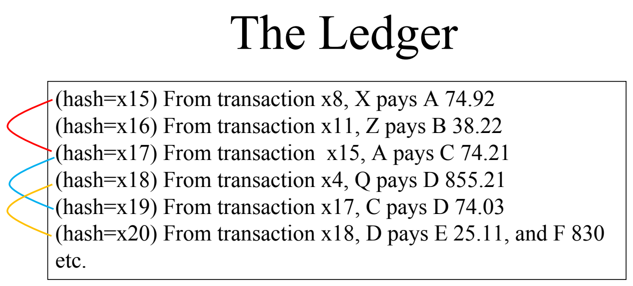


Figure 1



(hash=x15) From transaction x8, X pays A 74.92
(hash=x16) From transaction x11, Z pays B 38.22
(hash=x17) From transaction x15, A pays C 74.21
(hash=x18) From transaction x4, Q pays D 855.21
(hash=x19) From transaction x17, C pays D 74.03
(hash=x20) From transaction x18, D pays E 25.11, and F 830
etc.

Figure 2

a secret known as the *private key*. The signature can be verified by anyone with knowledge of the associated *public key*. And an essential component of any public key system is that there will be some way of making sure that the public key is well-known.

With a traditional public key system, the cryptography ensures that there is an enormous gap between the computation needed for someone with knowledge of the private key to generate a signature, and someone without knowledge of the key to forge a signature. With RSA, the computation necessary to generate a signature (knowing the private key) is a small power of the length of the key (between 2 and 3). In contrast, brute force breaking of a key is almost exponential in the length of the key. So, for instance, for a 1024 bit RSA key, it is about 2^{63} times more expensive to forge a signature than to generate one. Increasing the key size increases the gap between forging and generating signatures. If an RSA key were increased from 1024 bits to 2048 bits, the gap becomes about 2^{94} times more expensive to forge rather than generate a signature.

Since it's hard to imagine these huge numbers, another way to say it is that signing with RSA 1024 takes about a millisecond on a typical CPU, and signing with RSA 2048 might take 6 milliseconds on the same CPU. However, breaking RSA 1024 takes about as much computation as all the Bitcoin miners do in an hour. Breaking RSA 2048 takes about as much computation as all the Bitcoin miners would do if they continued at the present rate for a million years.

The startling aspect of the Bitcoin hash is that it is *equally* difficult for the community of miners to compute a hash as for someone to forge a hash. This means that the security of Bitcoin depends on the assumption that no entity or collection of entities can amass as much compute power as the Bitcoin mining community. This is a very surprising assumption. It would indeed be easy for a nation-state to amass more compute power than the Bitcoin community.

What could a malicious set of miners, with more compute power than the honest Bitcoin miners do? They could discriminate against certain transactions, refusing to ever record them in the ledger. They could compute an alternate ledger, where transactions they had previously spent were not recorded anymore, and then they could double-spend.

And not only is the security assumption highly questionable, since it is hard to believe that the community of honest miners has cornered the market on all computation power on the planet, but it means that the computation required by the honest miners is mind-bogglingly huge.

What Would Motivate Someone to Be a Miner?

Miners have to do a lot of computation if they ever hope to be rewarded with any Bitcoins. Currently, the miner community earns about 2 million US dollars every day. And reports are that this barely covers the amount they are spending on electricity. That amount of electricity is estimated to be equal to what a nuclear power plant generates per day, about 500 megawatts.

So any application of this technology must somehow generate revenue for the miners.

Other Costs

It is also necessary to store the entire ledger so that transactions can be checked for validity. Currently, the ledger is about 100 GB and is stored in thousands of places around the network. Also, there is a huge amount of network bandwidth to broadcast transactions and new blocks to all the Bitcoin nodes, as well as to be able to download the entire ledger to any node that is joining the community.

What Is Novel about Blockchain?

If "blockchain" is truly a revolution in computing, there must be something about it that did not exist before. What could it be?

Is It Having a "Ledger"?

Blockchain's "ledger" is an append-only log that needs to be kept in its entirety, and needs to be world-readable and world-writable. Very few applications really want these properties. Much more flexible databases have of course existed for a long time.

Is It Replicating the Data?

Blockchain highly replicates the ledger so that it will not easily get lost. Obviously, the more locations in which something is stored, the less likely it is that it will become permanently lost. Large public clouds tend to store data in perhaps six places, carefully chosen to be located in different locations so that a natural disaster in one location will not wipe out all copies of the data. If any copy is lost, the public cloud quickly replicates the data to new locations to replace the ones that have lost the data. In contrast, blockchain stores the ledger in thousands of locations.

To store something in *N* places requires *N* times as much storage, as well as network bandwidth to communicate the data to all the places. What is the optimal number of locations? It is unlikely that the extra redundancy of thousands vs. six merits the storage cost and network bandwidth for replication. And despite how many copies are kept, there have been many clones of Bitcoin that eventually failed due to lack of interest, and all of the copies then were lost, because there is no obligation for a node in a blockchain system to maintain the data.

Is It Being "Immutable"?

The term *immutable* means the data cannot be modified. The term "immutable ledger" isn't quite true. The data can certainly be modified, but the assumption is that there is an integrity check that can be used to detect whether the data has been modified. Blockchain did not invent the concept of an integrity check, just the concept of a horrendously expensive-to-compute integrity check. Traditional cryptography has long known about easy-to-compute integrity checks that are computationally infeasible to forge.

Furthermore, the ledger in blockchain is not actually immutable. Forks can occur, starting from, say, block *N*, where multiple different subsequent blocks *N+1* and further might be found. The hope is that this situation would be resolved quickly, because a miner seeing two different valid chains will only accept the longer one. However, a fork can persist for a long time if there were an Internet partition, or if the gossip network connecting the miners got partitioned, due to some highly connected node going down, perhaps. Also, if there were any incompatibility in code, such that a transaction looked valid in one version of the code and invalid in a different version, then the miners running different versions will ignore each other's chains. This situation actually occurred in 2013. If blockchain were truly decentralized, then this situation would be permanent. However, there are a few people who really are paying attention and in charge, and after the fork in 2013, they decided which version of the blockchain should live.

Is It Being Decentralized?

The concept of having a ledger agreed upon by consensus of thousands of anonymous entities, none of which can be held responsible or be shut down by some malevolent government, is fairly unique. However, most applications would not require or even want this property. And, as demonstrated by the Bitcoin community's reaction to forks, there really are a few people who are in charge who can control the system, by, for example, making a decision on which fork should be chosen.

The concept of general distributed databases is very old. For instance, this is a survey paper about the state of such systems from 1981 [3]. Such systems are more complicated than Blockchain, because they handle things like having multiple nodes simultaneously attempting to update the same location and atomic transactions. In contrast, Blockchain is an append-only log.

If all that were needed was an append-only log, and an application (e.g., a consortium of banks) wished to collaborate on maintaining the log, a very simple solution would be to have an entry signed by any of the trusted parties in the consortium appended to the log. To handle Byzantine failures (where a minority of the entities in the consortium might become untrustworthy), the simple solution would be to require an entry to be signed by a majority of the consortium before it is appended to the log.

So the novel part of Blockchain is having a consortium of *unknown* entities maintain the ledger.

Blockchain vs. Traditional Solutions for Sample Applications

In this section we'll examine some applications that have been proposed as uses for blockchain and compare more traditional approaches. Since these systems are not actually deployed, it's not possible to completely predict the details of a blockchain-based approach, but we'll mention some issues.

DNS Names

Assigning DNS names is an interesting application. DNS is quite political. Which organization controls the names in a domain? What is the definition of a country? It might be tempting to "democratize" DNS names to first-come first-served, without any organization deciding who is allowed to have which name. With blockchain technology, we could do without any central organizations. And there is indeed a revenue stream for paying the miners, since people would still have to pay to rent a name.

However, people have come to assume that names have some meaning. They assume that the owner of the name *usenix.org* has some affiliation with the organization USENIX. And someone will still need to maintain the servers to map DNS names to IP addresses, along with all the other information stored in DNS.

So it would be preferable to have some mediation of names by a large, identifiable organization that could be held accountable if it misbehaved. And the current system is much less expensive than a blockchain system would be.

Health Records

When switching doctors, or when visiting several doctors with different specialties, it is important for them all to have access to your health records. However, is a universal, world-readable unstructured database with everyone's medical data the best answer? The sheer size of the database is daunting, especially when, as proposed by some blockchain enthusiasts, all medical devices attached to all people would report their readings into the blockchain. And this database would be stored in thousands of places.

Clearly with medical information, people will not want their information world-readable. Which leads to many questions that blockchain doesn't answer. Data must be encrypted. Who manages the keys? Who authorizes a new doctor you are meeting to see your records? What if you are in an accident? And, furthermore, who authorizes you, a doctor or a device, to write something about you in the blockchain?

With traditional technology, there would be a database stored with several trusted organizations, organized so that data for a particular patient could be quickly retrieved (rather than needing to have all the pieces found by searching through the blockchain). And even if encrypted, there would likely be access control on the data. And maintaining the database would be much less expensive if one organization, or a few large organizations, were using traditional digital signatures as an integrity check on the data.

Timestamping

One of the applications claimed for blockchain is the ability to prove that something happened before some time, because of where it appears in the blockchain. For instance, to prove you invented something, you could write a paper about it and store a hash of the paper on the blockchain.

However, there is much less expensive technology that can accomplish this. A trusted timestamping service can take a hash, append a timestamp, and sign it. Since this is such an inexpensive service, there could be hundreds or thousands of them. If Alice wants to be able to prove to Bob that something existed before some time, she needs to collect multiple signed copies to ensure that, when she needs to prove a timestamp to Bob, at least one of the timestampers she used is trusted by Bob. It is less expensive for everyone who wants this service to store their own signed copies than to store them publicly in a large blockchain.

Conclusion

Blockchain technology is extremely expensive in terms of computation, storage, and network bandwidth. With traditional technology, it is possible to replicate data, and public clouds are careful to do so. But there would be a handful of replicas; not thousands. Also, databases would be more structured than an append-only log combining information from all users and for many applications.

Most applications (such as financial ones) do want to have some collection of well-known organizations at the heart of the technology to mediate disputes and be held responsible if things go wrong. If it is distasteful to have a single organization in the center, it could be a consortium of several, and transactions could be considered valid only after a majority of the inner circle of organizations have signed the transaction. This would be immensely less expensive, and be a more natural trust model, than thousands of anonymous miners.

And traditional cryptographic integrity checks (digital signatures) by well-known organizations are practical and inexpensive.

References

[1] K. Torpey, "Why the Bitcoin Blockchain Is the Biggest Thing Since the Internet," *Bitcoin Magazine*, April 19, 2016: http://www.nasdaq.com/article/why-the-bitcoin-blockchain-is-the-biggest-thing-since-the-internet-cm608228.

[2] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," Bitcoin: https://bitcoin.org/bitcoin.pdf.

[3] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *Computing Surveys,* vol. 13, no. 2, June 1981: https://people.eecs.berkeley.edu/~brewer/cs262/concurrency-distributed-databases.pdf.

Internet of Pwnable Things

Challenges in Embedded Binary Security

JOS WETZELS

Jos Wetzels is a Research Assistant with the Distributed and Embedded Security (DIES) Research Group at the University of Twente in The Netherlands. He currently works on projects aimed at hardening embedded systems used in critical infrastructure, where he focuses on binary security in general and exploit development and mitigation in particular, and has been involved in research regarding on-the-fly detection and containment of unknown malware and advanced persistent threats. He has assisted teaching hands-on offensive security classes for graduate students at the Dutch Kerckhoffs Institute for several years. a.l.g.m.wetzels@gmail.com

Embedded systems are everywhere, from consumer electronics to critical infrastructure, and with the rise of the Internet of Things (IoT), such systems are set to proliferate throughout all aspects of everyday life. Due to their ubiquitous and often critical nature, such systems have myriad security and privacy concerns, but proper attention to these aspects in the embedded world is often sorely lacking. In this article I will discuss how embedded binary security in particular tends to lag behind what is commonly expected of modern general purpose systems, why bridging this gap is non-trivial, and offer some suggestions for promising defensive research directions.

Embedded Systems Security

Because embedded systems are so diverse, the threat landscape is equally varied, ranging from life-threatening sabotage of cyber-physical systems (e.g., electrical blackouts, smart-car crashes, insulin pump tampering) to economic (e.g., cable TV piracy, smart meter fraud) and privacy (e.g., smart-home surveillance) threats. Embedded security priorities also differ from those in the *general purpose* (GP) world. Whereas the latter tend to be mostly concerned about threats to confidentiality, embedded systems tend to prioritize availability and integrity. You want nuclear reactors to operate safely and automotive braking and flight control systems to function properly at all times.

Compared to GP systems, attention to embedded security is relatively recent, something that is especially visible in the industrial control systems (ICS), which form the technological backbone of electric grids, water supplies, and manufacturing environments. These systems were never designed to be connected to untrusted networks in the first place but, over the years, have steadily become more and more networked and exposed. As a result, these systems do not have corresponding security improvements. And concerns here are far from hypothetical as high-profile attacks have damaged nuclear facilities in Iran, caused blackouts on the Ukrainian power grid, and physically damaged a German steel mill.

This situation is compounded by the challenges of embedded patch deployment. Whereas in the GP world, patch management is often centralized and automated, the embedded world is faced by a myriad of problems (absence of hot-patching capabilities, safety recertification upon introduction of new code, extreme availability requirements, long device lifespans exceeding vendor support, etc.) complicating such an approach. This creates a situation of prolonged vulnerability exposure and exploits with long shelf-life capable of targeting millions of vulnerable, unpatched, and connected embedded devices.

Memory Corruption, Safe Languages, and Exploit Mitigations

When it comes to embedded systems, memory corruption issues (e.g., buffer overflows) consistently rank among the most prevalent categories of vulnerabilities as exemplified by a 2016 Kaspersky study of ICS vulnerabilities [1]. This prevalence is largely due to the dominance of unsafe languages such as C++ and assembly in embedded software development. As

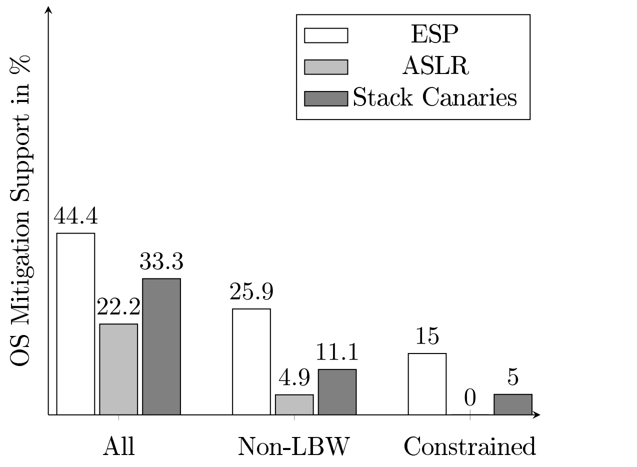


Figure 1: Exploit mitigation support among 36 popular embedded OSes. Non-LBW means Non-Linux, BSD, and Windows-based OSes, and Constrained indicates those tiny, minimalistic OSes designed for so-called deeply embedded systems.

someone once said: "C is a terse and unforgiving abstraction of silicon." Ideally, this problem would be mitigated by widespread adoption of safe languages, and while some are currently used (e.g., Ada, which is used in civilian and military avionics and aerospace systems) or show potential (e.g., Rust, which provides memory safety without garbage collection) for future adoption in the embedded world, there are some serious limitations.

First of all, the "close to metal" nature of C makes it well-suited for writing similarly bare-metal software (e.g., OSes or firmware) in a way that almost all safe languages are not. Note that Rust seems promising in this regard as shown by the intermezzOS and Tock [2, 8] OSes. Secondly, there's the issue of portability as there are billions of lines of legacy code written in unsafe languages, and there already are C toolchains for nearly every platform out there. Hence, even if the ideal embedded safe language existed right now, it would still take quite a while for an industry-wide shift in development practices to take off, never mind what to do with all that legacy code. So safe languages are a long-term solution at best, and we live in a short-term world that needs short-term solutions.

Exploit mitigations are just such a short-term solution since they seek to complicate exploitation of existing vulnerabilities rather than prevent their introduction in the first place. Exploit development can be conceptualized as the programming of so-called "weird machines" [3] through composition of "exploit primitives" into a chain. Complicating this chain means *making each link harder to forge* by making mitigations harder to overcome and *lengthening the chain* by crafting mitigations for various steps of the exploitation process in order to raise attacker cost and eliminate practical exploitability of certain vulnerabilities altogether.

Ever since memory corruption vulnerabilities started getting widespread attention with Aleph One's 1996 *Phrack* article "Smashing the Stack for Fun and Profit," various exploit mitigations have been proposed, implemented, broken, and improved until we've arrived at the present-day situation, where exploiting a stack buffer overflow on a modern GP system often requires you to at least either find an information leak to bypass stack canaries or overwrite a function pointer, find an information leak to bypass ASLR, craft a ROP (return-oriented programming) chain to bypass non-executable memory, and find a sandbox escape: that's two to three additional bugs (though less if one has a flexible enough vulnerability) on top of the actual vulnerability itself to achieve arbitrary code execution.

Embedded Exploitation: Blast from the Past

Compared to the GP world, embedded exploitation often feels like it's stuck somewhere in the '90s. Consider, for example, the Shadow Brokers incident [4] last year, where an unknown threat actor managed to obtain exploit and implant code used by the top-tier, probably state-sponsored, Equation Group threat actor and published part of the plunder online. This included exploits targeting enterprise firewalls used in very sensitive environments; what stood out here is that none of the exploits needed bypasses for any mitigation whatsoever.

In order to get an idea of what the situation with respect to embedded mitigation adoption looks like, I surveyed 36 popular embedded operating systems (ranging from high-end Linux-based ones to tiny proprietary real-time microkernels) for support of the "bread & butter" baseline of mitigations: Executable Space Protection (ESP, also known as DEP, NX, or W^X memory), Address Space Layout Randomization (ASLR), and stack canaries (also known as stack cookies or stack smashing protection). Briefly put: ESP forces attackers to use code-reuse payloads (such as ROP chains) by making data memory non-executable, while ASLR complements this by ensuring memory layout secrecy in order to prevent attackers from constructing such code-reuse payloads. Stack canaries are orthogonal to the former mitigations and work by inserting a randomized secret value, between stackframe metadata and local variables, that is checked for integrity when a function returns in order to detect whether it has been overwritten as part of a stack-smashing attack.

As you can see in Figure 1, only a minority supports these mitigations, and this becomes a negligibly small minority once you discard the Linux-, BSD-, and Windows-based OSes or only consider the most constrained OSes. And note that this survey was an optimistic one: if a mitigation is supported by an OS for even a single platform, no matter implementation quality, it was marked as supported. It's pretty safe to say embedded binary security lags behind the GP world significantly.

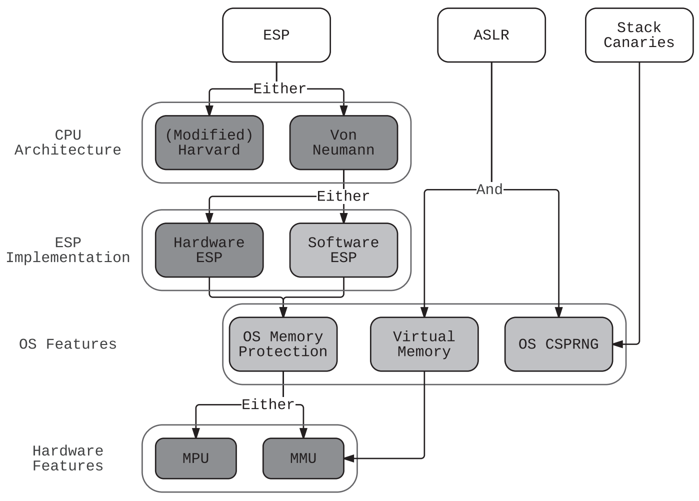


Figure 2: Exploit mitigation hardware and OS feature dependencies

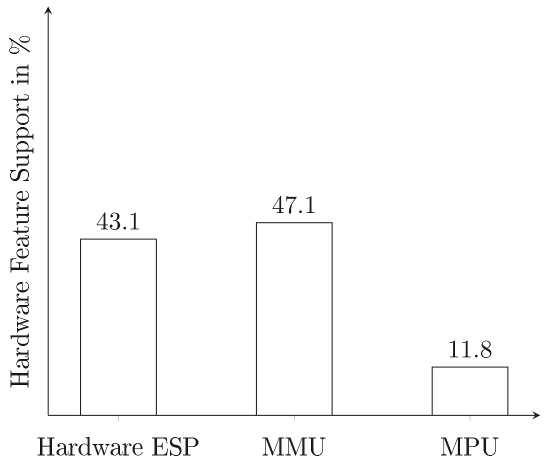


Figure 3: Hardware feature support among 51 popular von Neumann embedded core families

Dependencies, Constraints, and Possible Solutions

So what's the reason for this adoption gap? Well, it turns out that if you map out the hardware and software dependencies of these mitigations (Figure 2), there's some serious constraints that complicate adoption. Embedded devices are designed for a specific task and tend to have limited resources as well as often being headless and diskless. The hardware is often simple and lacking in advanced features, and the software is tailored for such constraints. And on top of all that there are often real-time and safety-critical requirements.

In order to get an idea of the state of mitigation dependency support among common embedded hardware and OSes, I surveyed 51 popular von Neumann-style embedded core families (Figure 3) and mapped out OS feature dependency support (Figure 4) among the 36 previously surveyed OSes. As shown in these figures, widespread support for key dependencies is lacking, which presents a significant hurdle to mitigation adoption. To see why these dependencies are so crucial and to provide some suggestions for research directions that can potentially overcome existing limitations, let's take a look at each mitigation in our baseline in detail.

Stack Canaries and Embedded Random Number Generators (RNGs)

Stack canary mechanisms are implemented as a compiler feature but require some sort of (secure) random number generator to be present on the target OS to generate the master canary value when the binary in question is loaded. This is best left to the cryptographically secure pseudo-random number generator (CSPRNG) provided by the OS itself (e.g., */dev/urandom* on UNIX-like systems), but as Figure 4 shows, only 41.7% of surveyed embedded OSes provide a system CSPRNG, and this number drops to 22.2% if you eliminate Linux-, BSD-, and Windows-based ones and becomes negligible altogether if you only consider the most constrained operating systems.

I've discussed the issues with embedded OS CSPRNGs in more detail in my recent 33C3 talk "Wheel of Fortune: Analyzing Embedded OS Random Number Generators." To put it briefly, it's not trivial to port existing designs from the GP world, mainly because of a combination of resource constraints in terms of processing speed, memory and power consumption, and a general low entropy environment. These systems are designed for limited, specific tasks, often in a machine-to-machine setting without human activity, and are designed to perform those tasks in a reliable, predictable fashion. This is a major stumbling block because PRNGs need sources with some external entropy in order to stretch their output into longer sequences of pseudo-random output.

On GP systems common sources for entropy are user input devices like the mouse, keyboard, or disk activity, but since many embedded systems are headless and/or diskless this is not an option. Depending on the embedded device in question, potentially suitable entropy sources might be available from sensor values, radio measurements, accelerometer data, etc., but from an OS designer's point of view these sources cannot be assumed to be universally present on all devices the OS is to be deployed on. This problem would ideally be solved by having omnipresent on-chip high-throughput true random number generators (TRNGs), but this is quite unrealistic considering accompanying cost increases. In addition, it doesn't help with existing legacy systems.

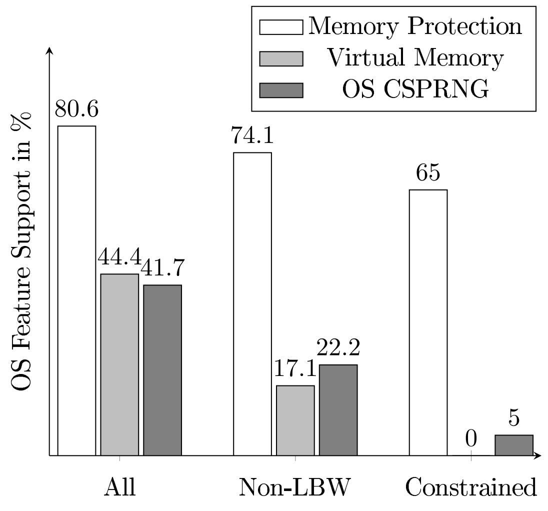


Figure 4: OS feature support among 36 popular embedded OSes

Two promising research directions upon which embedded OS CSPRNG designers could draw are advances in lightweight cryptography and investigation of omnipresent entropy sources. The former encompasses various cryptographic primitives designed for highly constrained systems. Initiatives such as the ACRYPT project [5] have produced a "zoo" of IoT-oriented lightweight primitives, with accompanying implementation footprint information (in terms of code size, memory usage, and execution time), which can serve as building blocks in a larger OS CSPRNG design. The embedded entropy problem is a more fundamental one and doesn't lend itself well to a one size fits all solution, but a thorough exploration of the suitability of potential entropy sources, which are virtually omnipresent in embedded systems, such as startup values of on-chip SRAM, clock jitter, and so on, would definitely be worthwhile.

Executable Space Protection (ESP)

Essentially there are two main CPU architectural styles: Harvard and von Neumann. The former has physically separate code and data memories, while the latter has a single memory holding both code and data. There are many possible nuances to these "pure" styles, but when it comes to the goals of ESP the only thing that matters is that memory can't be both writable and executable so that attackers can't easily inject malicious shellcode payloads into memory. As such, Harvard architectures trivially provide ESP, but for von Neumann-style CPUs, ESP will have to be implemented either in a hardware-assisted way or through software emulation. The former case is implemented in the form of a dedicated hardware feature (x86 NX bit, ARM XN bit, etc.), usually as part of the memory management unit (MMU) regulating memory executability at a certain granularity level on a per-page basis. In the case of software emulation, there are multiple approaches all outside the scope of this article, the most famous of them probably being the PaX project's implementation [6], but all of them incur at least some overhead and tend to be architecture-specific.

As shown in Figure 3, 43.1% of surveyed core families have hardware ESP support, something you need to consider in light of the fact that software emulation-based approaches to ESP only exist for a limited number of OS and architecture combinations (e.g., Linux on x86). Both ESP implementations require memory protection support (and as such an MMU or more lightweight memory protection unit (MPU)) on the part of the OS to allow for memory permission management. And while most embedded OSes offer memory protection support, we can see in Figure 3 that only 47.1% of all surveyed core families have MMU support and only 11.8% have MPU support, leaving 41.1% unable to accommodate memory protection. Now some microcontrollers might offer (limited) memory permission management capabilities without featuring an MPU/MMU, and for some processors there are external MMUs available, like the Motorola 68851, but apart from these edge cases, there's a significant "gap segment" of embedded systems without support for the core ESP dependencies.

Ideally, embedded systems designers would start consciously using either Harvard CPUs (AVR, 8051, PIC, etc.) or von Neumann ones with hardware ESP support (ARMv6+, MIPS32r3+, x86, etc.), but for those systems where this is not an option we will need widespread embedded OS adoption of a multi-architecture, low-overhead software emulation ESP approach. This does, however, still leave us with the open problem of how to deal with MPU-/MMU-less systems that cannot offer any form of memory protection to begin with.

Address Space Layout Randomization (ASLR)

In order to craft the code-reuse payloads used to bypass ESP, attackers will have to know the addresses of particular code fragments (so-called "gadgets") to incorporate into their payload. ASLR aims to complicate this by ensuring memory layout secrecy through randomization, which is done by placing various different memory objects—the stack, heap, main program image, loaded libraries—at randomized addresses. In order to do this, ASLR has three key dependencies: a CSPRNG, OS virtual memory support, and hardware with an MMU. The ASLR randomization takes place at load-time and draws upon an OS CSPRNG, as we've seen earlier, and is far from omnipresently available in all embedded operating systems.

Virtual memory provides memory isolation between different tasks/processes and thus prevents shared memory conflicts that might otherwise arise from ASLR's memory object randomization. If we look at Figure 4, however, we can see that only 44.4% of all surveyed embedded operating systems provide virtual memory support, and this number drops to a mere 17.1% if we eliminate the Linux-, BSD-, and Windows-based OSes. Even worse are the most constrained operating systems, none of which support virtual memory for various reasons, such as being designed for MMU-less and diskless targets or having conflicting hard real-time requirements.

This widespread lack of embedded virtual memory and MMU support are two major obstacles to widespread ASLR adoption that are not going away anytime soon, which means that we need an embedded alternative to ASLR. ASLR's dependency on virtual memory arises from the fact that it is a *load-time software diversification* technique [7]. This dependency does not apply, however, to diversification techniques operating at earlier points in the software life cycle such as at compile or install time. In these cases either a compiler feature or a dedicated transformation program produce diversified binaries by randomizing code layout and/or individual instruction sequences. Such approaches achieve a similar effect to ASLR by randomizing the addresses (and nature) of code-reuse gadgets but have the downside of being less effective since they only diversify between different software builds or individual device instances rather than between individual boots or program runs as well as only diversifying code memory. There are currently no mature, widely adopted implementations of such schemes that I know of, nor has their applicability to the embedded world been covered, but they seem to be a promising embedded ASLR alternative.

A Call to Action

So where do we go from here? First of all, security researchers should continue to demonstrate the urgency and impact of embedded vulnerabilities to drive the point home that embedded systems cannot afford to keep lagging behind when they are becoming increasingly ubiquitous and interconnected. Secondly, work on short-term solutions (researchers addressing the challenges outlined in this article working together with OS developers to push for embedded exploit mitigation adoption) should be conducted alongside work on more long-term solutions such as embedded safe language research and development of secure embedded patching and updating mechanisms. And, finally, with the rise of the Internet of Things there is a real need for IoT standardization, policy, and regulation that focuses on *security by design* rather than leaving it as an afterthought or something that has to be retrofitted after the first vulnerabilities are discovered due to a vendor focus on novel features and time-to-market.

References

[1] O. Andreeva, S. Gordeychik, G. Gritsai, O. Kochetova, E. Potseluevskaya, S. I. Sidorov, and A. A. Timorin, "Industrial Control Systems Vulnerabilities Statistics," Kaspersky Lab, 2016: https://kasperskycontenthub.com/securelist/files/2016/07/KL_REPORT_ICS_Statistic_vulnerabilities.pdf.

[2] http://intermezzos.github.io/.

[3] S. Bratus, S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, "Exploit Programming: From Buffer Overflows to 'Weird Machines' and Theory of Computation," *;login:* vol . 36, no. 6 (December 2011): https://www.usenix.org/system/files/login/articles/105516-Bratus.pdf.

[4] M. Al-Bassam, Equation Group Firewall Operations Catalog, 2016.

[5] A. Biryukov, D. Dinu, J. Großschädl, D. Khovratovich, Y. Le Corre, L. Perrin, "The ACRYPT Project: Lightweight Cryptography for the Internet of Things," CRYPTO 2015 Rump Session, 2015.

[6] NOEXEC, PaX project Documentation, 2003.

[7] P. Larsen, A. Homescu, S. Brunthaler, M. Franz, "SoK: Automated Software Diversity," 2014 IEEE Symposium on Security and Privacy: https://www.ics.uci.edu/~perl/automated_software_diversity.pdf.

[8] https://www.tockos.org/.

COLUMNS

Revisiting Pathlib

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

Over the last few years Python has changed substantially, introducing a variety of new language syntax and libraries. While certain features have received more of the limelight (e.g., asynchronous I/O), an easily overlooked aspect of Python is its revamped handling of file names and directories. I introduced some of this when I wrote about the pathlib module in *;login:* back in October 2014 [1]). Since writing that, however, I've been unable to bring myself to use this new feature of the library. It was simply too different, and it didn't play nicely with others. Apparently, I wasn't alone in finding it strange--pathlib [2] was almost removed from the standard library before being rescued in Python 3.6. Given that three years have passed, maybe it's time to revisit the topic of file and directory handling.

The Old Ways

If you have to do anything with files and directories, you know that the functionality is spread out across a wide variety of built-in functions and standard library modules. For example, you have the open function for opening files:

```
with open('Data.txt') as f:
    data = f.read()
```

And there are functions in the os module for dealing with directories:

```
import os

files = os.listdir('.')     # Directory listing
os.mkdir('data')            # Make a directory
```

And then there is the problem of manipulating pathnames. For that, there is the os.path module. For example, if you needed to pull a file name apart, you could write code like this:

```
>>> filename = '/Users/beazley/Pictures/img123.jpg'
>>> import os.path

>>> # Get the base directory name
>>> os.path.dirname(filename)
'/Users/beazley/Pictures'

>>> # Get the base filename
>>> os.path.basename(filename)
'img123.jpg'

>>> # Split a filename into directory and filename components
>>> os.path.split(filename)
('/Users/beazley/Pictures', 'img123.jpg')
```

```
>>> # Get the filename and extension
>>> os.path.splitext(filename)
('/Users/beazley/Pictures/img123', '.jpg')
>>>

>>> # Get just the extension
>>> os.path.splitext(filename)[1]
'.jpg'
>>>
```

Or if you needed to rewrite part of a file name, you might do this:

```
>>> filename
'/Users/beazley/Pictures/img123.jpg'
>>> dirname, basename = os.path.split(filename)
>>> base, ext = os.path.splitext(basename)
>>> newfilename = os.path.join(dirname, 'thumbnails', base+'.png')
>>> newfilename
'/Users/beazley/Pictures/thumbnails/img123.png'
>>>
```

Finally, there are an assortment of other file-related features that get regular use. For example, the `glob` module can be used to get file listings with shell wildcards. The `shutil` module has functions for copying and moving files. The `os` module has a `walk()` function for walking directories. You might use these to search for files and perform some kind of processing:

```
import os
import os.path
import glob

def make_dir_thumbnails(dirname, pat):
    filenames = glob.glob(os.path.join(dirname, pat))
    for filename in filenames:
        dirname, basename = os.path.split(filename)
        base, ext = os.path.splitext(basename)
        origfilename = os.path.join(dirname, filename)
        newfilename = os.path.join(dirname, 'thumbnails', base+'.png')
        print('Making thumbnail %s -> %s' % (filename, newfilename))
        out = subprocess.check_output(['convert', '-resize',
          '100x100', origfilename, newfilename])

def make_all_thumbnails(dirname, pat):
    for path, dirs, files in os.walk(dirname):
        make_dir_thumbnails(path, pat)

# Example
make_all_thumbnails('/Users/beazley/PhotoLibrary', '*.JPG')
```

If you've written any kind of Python code that manipulates files, you're probably already familiar with this sort of code (for better or worse).

The New Way

Starting in Python 3.4, it became possible to think about pathnames in a different way. Instead of merely being a string, a pathname could be a proper object in its own right. For example, you could make a Path [3] instance and do this like this:

```
>>> from pathlib import Path
>>> filename = Path('/Users/beazley/Pictures/img123.jpg')
>>> filename
PosixPath('/Users/beazley/Pictures/img123.jpg')
>>> data = filename.read_bytes()
>>> newname = filename.with_name('backup_' + filename.name)
>>> newname
PosixPath('/Users/beazley/Pictures/backup_img123.jpg')
>>> newname.write_bytes(data)
1805312
>>>
```

Manipulation of the file name itself turns into methods:

```
>>> filename.parent
PosixPath('/Users/beazley/Pictures')
>>> filename.name
'img123.jpg'
>>> filename.parts
('/', 'Users', 'beazley', 'Pictures', 'img123.jpg')
>>> filename.parent / 'newdir' / filename.name
PosixPath('/Users/beazley/Pictures/newdir/img123.jpg')
>>> filename.stem
'img123'
>>> filename.suffix
'.jpg'
>>> filename.with_suffix('.png')
PosixPath('/Users/beazley/Pictures/img123.png')
>>> filename.as_uri()
'file:///Users/beazley/Pictures/img123.jpg'
>>> filename.match('*.jpg')
True
>>>
```

Paths have a lot of other useful features. For example, you can easily get file metadata:

```
>>> filename.exists()
True
>>> filename.is_file()
True
>>> filename.owner()
'beazley'
>>> filename.stat().st_size
1805312
>>> filename.stat().st_mtime
1388575451
>>>
```

There are also some nice directory manipulation features. For example, the glob method returns an iterator for finding matching files:

```
>>> pics = Path('/Users/beazley/Pictures')
>>> for pngfile in pics.glob('*.PNG'):
...     print(pngfile)
...
/Users/beazley/Pictures/IMG_3383.PNG
/Users/beazley/Pictures/IMG_3384.PNG
/Users/beazley/Pictures/IMG_3385.PNG
...
>>>
```

If you use rglob(), you will search an entire directory tree. For example, this finds all PNG files in my home directory:

```
for pngfile in Path('/Users/beazley').rglob('*.PNG'):
      print(pngfile)
```

The Achilles Heel…And Much Sadness

At first glance, it looks like Path objects are quite useful—and they are. Until recently, however, they were a bit of an "all-in" proposition: if you created a Path object, it couldn't be used anywhere else in the non-path world. In Python 3.5, for example, you'd get all sorts of errors if you ever used a Path with more traditional file-related functionality:

```
>>> # PYTHON 3.5
>>> filename = Path('/Users/beazley/Pictures/img123.png')
>>> open(filename, 'rb')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: invalid file: PosixPath('/Users/beazley/Pictures/
img123.png')

>>> os.path.dirname(filename)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.5/posixpath.py", line 148,
in dirname i = p.rfind(sep) + 1
AttributeError: 'PosixPath' object has no attribute 'rfind'
>>>

>>> import subprocess
>>> subprocess.check_output(['convert', '-resize', '100x100',
filename, newfilename])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.5/subprocess.py", line 626,
in check_output **kwargs).stdout
  File "/usr/local/lib/python3.5/subprocess.py", line 693,
in run with Popen(*popenargs, **kwargs) as process:
  File "/usr/local/lib/python3.5/subprocess.py", line 947,
in __init__ restore_signals, start_new_session)
  File "/usr/local/lib/python3.5/subprocess.py", line 1490,
in _execute_child restore_signals, start_new_session, preexec_fn)
TypeError: Can't convert 'PosixPath' object to str implicitly
>>>
```

Basically, pathlib partitioned Python into two worlds—the world of pathlib and the world of everything else. It's not entirely unlike the separation of Unicode versus bytes, which is to say rather unpleasant if you don't know what's going on. You could get around these limitations, but the fix involves placing explicit string conversions everywhere. For example:

```
>>> import subprocess
>>> subprocess.check_output(['convert', '-resize', '100x100',
str(filename), str(newfilename)])
>>>
```

Frankly, that's pretty annoying. It makes it virtually impossible to pass Path objects around in your program as a substitute for a file name. Everywhere that passed the name a low-level function would have to remember to include the string conversion. Modifying the whole universe of Python code is just not practical. It's forcing me to think about a problem that I don't want to think about.

Python 3.6 to the Rescue!

The good news is that pathlib was rescued in Python 3.6. A new magic protocol was introduced for file names. Specifically, if a class defines a __fspath__() method, it is called to produce a valid path string. For example:

```
>>> filename = Path('/Users/beazley/Pictures/img123.png')
>>> filename.__fspath__()
'/Users/beazley/Pictures/img123.png'
>>>
```

A corresponding function fspath() that was added to the os module for coercing a path to a string (or returning a string unmodified):

```
>>> import os
>>> os.fspath(filename)
'/Users/beazley/Pictures/img123.png'
>>>
```

A corresponding C API function was also added so that C extensions to Python could receive path-like objects.

Finally, there is also an abstract base class that can be used to implement your own custom path objects:

```
class MyPath(os.PathLike):
    def __init__(self, name):
        self.name = name
    def __fspath__(self):
        print('Converting path')
        return self.name
```

The above class allows you to investigate conversions. For example:

```
>>> p = MyPath('/Users/beazley/Pictures/img123.jpg')
>>> f = open(p, 'rb')
Converting path
>>> os.path.dirname(p)
Converting path
'/Users/beazley/Pictures'
>>> subprocess.check_output(['ls', p])
Converting path
b'/Users/beazley/Pictures/img123.png\n'
>>>
```

So far as I can tell, the integration of Path objects with the Python standard library is fairly extensive. All of the core file-related functionality in modules such as os, os.path, shutil, subprocess seems to work. By extension, nearly any standard library module that accepts a file name and uses that standard functionality will also work. It's nice. Here's a revised example of code that uses pathlib:

```
from pathlib import Path
import subprocess
def make_thumbnails(topdir, pat):
    topdir = Path(topdir)
    for filename in topdir.rglob(pat):
        newdirname = filename.parent / 'thumbnails'
        newdirname.mkdir(exist_ok=True)
        newfilename = newdirname / (filename.stem + '.png')
        out = subprocess.check_output(['convert', '-resize','100x100',
                                       filename, newfilename])
if __name__ == '__main__':
    make_thumbnails('/Users/beazley/PhotoLibrary', '*.JPG')
```

That's pretty nice.

Potential Potholes

Alas, all is still not entirely perfect in the world of paths. One area where you could get tripped up is in code that's too finicky about type checking. For example, a function like this will hate paths:

```
def read_data(filename):
    assert isinstance(filename, str), "Filename must be a string"
    ...
```

If you're a library writer, it's probably best to coerce the input through os.fspath() instead. This will report an exception if the input isn't compatible. Thus, you could write this:

```
def read_data(filename):
    filename = os.fspath(filename)
    ...
```

You can also get tripped up by code that assumes the use of strings and performs string manipulation to do things with file names. For example:

```
def make_backup(filename):
    backup_file = filename + '.bak'
    ...
```

If you pass a Path object to this function, it will crash with a TypeError since Path instances don't implement the + operator. Shame on the author for not using the os.path module in the first place. Again, the problem can likely be solved with a coercion.

```
def make_backup(filename):
    filename = os.fspath(filename)
    backup_file = filename + '.bak'
    ...
```

But be aware that file names are allowed to be byte-strings. Even if you make the above change, the code is still basically broken. The concatenation will fail if a byte-string file name is passed.

C extensions accepting file names could also potentially break unless they are using the new protocol. Hopefully, such cases are rare—it's not too common to see libraries that directly open files on their own as opposed to using Python's built-in functions.

Final Words

All things considered, it now seems like pathlib might be something that can be used as a replacement for os.path without too much annoyance. Now, I just need to train my brain to use it—honestly, this might be even harder than switching from print to print(). However, let's not discuss that.

References

[1] D. Beazley, "A Path Less Traveled," *;login:*, vol. 39, no. 5 (October 2014), pp. 47–51: https://www.usenix.org/system/files/login/articles/login_1410_10_beazley.pdf.

[2] pathlib module: https://docs.python.org/3/library/pathlib.html.

[3] PEP 519—Adding a file system path protocol: https://www.python.org/dev/peps/pep-0519/.

Practical Perl Tools

Perl on a Plane

DAVID N. BLANK-EDELMAN

David Blank-Edelman is the Technical Evangelist at Apcera (the comments/views here are David's alone and do not represent Apcera/Ericsson) . He has spent close to 30 years in the systems administration/DevOps/SRE field in large multiplatform environments including Brandeis University, Cambridge Technology Group, MIT Media Laboratory, and Northeastern University. He is the author of the O'Reilly Otter book *Automating System Administration with Perl* and is a frequent invited speaker/organizer for conferences in the field. David is honored to serve on the USENIX Board of Directors. He prefers to pronounce Evangelist with a hard 'g'. dnb@usenix.org

I travel a great deal these days for my work, so it isn't uncommon for me to find myself on an airplane hoping to get some work done with only dribbles of WiFi. In those cases, you often have to make do with whatever is already on your laptop. I thought it might be interesting to explore what sort of goodies you might have available under those conditions from a stock Perl installation. To make this column extra realistic, let me report that as I write this I am flying at 34,153 ft at a speed of 437 mph over Lake Ontario (honest truth). Right before I left for the airport, I used perlbrew to install a stock version of the stable version of Perl (5.24.1) on my laptop. Let's switch to it and start our exploration:

```
$ source ~/perl5/perlbrew/etc/bashrc
$ perlbrew --notest install perl-5.24.1
$ perlbrew use perl-5.24.1
```

Perlbrew is a lovely tool for installing a discrete installation of Perl on a machine without perturbing any version of Perl shipped with the system. It will pull down the source for the version you desire and compile it. In the second line, I had to add `--notest` because 5.24.1 appears to have an issue on the version of OS X I'm running, which has a few broken tests in Time::Hires to be fixed in future versions of Perl. After hitting that failure a few times, I didn't think it would materially change what happens in this column, so I chose to skip the tests normally run as part of installing Perl.

The first place to look for interesting material is in the documentation system. Say what you'd like about Perl, no one can accuse it of not shipping with enough documentation. If I type "perldoc perl" it lists the following (heavily excerpted) list:

```
Overview
    perl            Perl overview (this section)
    perlintro       Perl introduction for beginners
    perlrun         Perl execution and options
    perltoc         Perl documentation table of contents

Tutorials
    perlreftut      Perl references short introduction
    perldsc         Perl data structures intro
    perllol         Perl data structures: arrays of arrays

    perlrequick     Perl regular expressions quick start
    perlretut       Perl regular expressions tutorial

    perlootut       Perl OO tutorial for beginners

    perlperf        Perl performance and optimization techniques

    perlstyle       Perl style guide
```

```
        perlcheat         Perl cheat sheet
        perltrap          Perl traps for the unwary
        perldebtut        Perl debugging tutorial

        perlfaq           Perl frequently asked questions

    Reference Manual
        perlsyn           Perl syntax
        perldata          Perl data structures
        perlop            Perl operators and precedence
        perlsub           Perl subroutines
        perlfunc          Perl built-in functions
...
        perluniintro      Perl Unicode introduction
...
        perlunitut        Perl Unicode tutorial
        perlebcdic        Considerations for running Perl on
                          EBCDIC platforms

        perlsec           Perl security

        perlmod           Perl modules: how they work
...
    Internals and C Language Interface
        perlembed         Perl ways to embed perl in your C or C++
                          application
        perldebguts       Perl debugging guts and tips
        perlxstut         Perl XS tutorial
...
    Miscellaneous
        perlbook          Perl book information
        perlcommunity     Perl community information

        perldoc           Look up Perl documentation in Pod
                          format

        perlhist          Perl history records
        perldelta         Perl changes since previous version

        perlexperiment    A listing of experimental features in Perl
...
    Language-Specific
        perlcn            Perl for Simplified Chinese (in EUC-CN)
        perljp            Perl for Japanese (in EUC-JP)
        perlko            Perl for Korean (in EUC-KR)
        perltw            Perl for Traditional Chinese (in Big5)

    Platform-Specific
        perlaix           Perl notes for AIX
        perlamiga         Perl notes for AmigaOS
        perlandroid       Perl notes for Android
        perlbs2000        Perl notes for POSIX-BC BS2000
...
```

Be sure to run that command to see the full list for yourself. There are 178 documents in all. So even if you just decide to spend your time reading Perl docs on a plane, you've got plenty of material available to you.

The Weirdest Module Search You Ever Did See

There's a straightforward way to find the modules installed with Perl, but let's go looking for interesting modules the hard way. What if we searched for all of the modules mentioned in the Perlfaq documents and used that as the starting place for our exploration? There are more sophisticated ways to find all of the modules, but let's start with a crude hammer and look for all of the :: sequences in the FAQs. And as we do it, let's eliminate all of those mentioned with CPAN on the same line (since we theoretically don't have great access to it here in the air):

```
for i in 1 2 3 4 5 6 7 8; do
   perldoc perlfaq$i|grep '::'|grep -v CPAN
done
```

This yields 307 lines (not all of which actually include non-CPAN-dwelling module names), so I'm going to cherry-pick a few that look interesting and talk about them:

Module::CoreList—Why, yes, there is a madness in my method. Wait, strike that, reverse that. Module::CoreList is a great place to start because it is a module that can help us find and describe the modules that have shipped with Perl (core) over the years. We could either use the command line utility that comes with it (`corelist`) or write little snippets of code like:

```
use Module::CoreList;

print join("\n",Module::CoreList->find_modules('^Text::',$]));
```

This will display all of the Text::* modules that ship in core with the current version of Perl. `find_modules()` searches for a regular expression and also takes a second argument describing which Perl versions it should consider. The magic variable `$]` returns the current version of Perl. We print this using a join just to place each element in the returned array on its own line. And, yes, this would be a fine and dandy way to find all of the modules shipped with the current copy of Perl. Something like this:

```
print join("\n",Module::CoreList->find_modules('',$]));
```

But if I told you that, it might cut short our little wandering walk together, so let's keep this between the two of us. As a small aside, it probably would have made my cherry-picking of modules to discuss here more efficient if I had run them through `Module::CoreList::is_core` first.

ExtUtils::Installed—Okay, really I'm not cooking the books here. This is the next module that comes up in the FAQ. ExtUtils::Installed gives you a way to figure out the names of all of the modules installed and the files and directories for each.

This is distinct from the previous module that talks about what modules are shipped with the core vs. the ones that are currently installed (core + whatever else you installed). It does this by inspecting the special "dot file droppings" that get installed with a module (.packlist). When I first tried out this module, it briefly puzzled me. I wrote:

```
use ExtUtils::Installed;

my $inst = ExtUtils::Installed->new();
print join("\n",$inst->modules());
```

And it printed:

```
Perl
```

That's right, just "Perl." It turns out that `ExtUtils::Installed` attempts to be smart. It knows which modules are considered "core" and lumps those all into "Perl." When I ran the same script using an older version of Perl that had more modules that I had expressly installed, it did indeed report the list of installed modules in addition to just "Perl." `ExtUtils::Installed` can do other tricks like show you the files and directories installed by a module—for example:

```
print join("\n",$inst->directories("Perl"))
```

will indeed show you all of the directories of all of the modules shipped with that version of Perl live from your file system.

TimePiece—If you've ever found it annoying to use `localtime()` or `gmtime()` in Perl because it either returns an array of fields you have to guess how to index to find the field you want or (in a scalar context) just a string:

```
$ perl -de 0
DB<1> x localtime()
0  24
1  47
2  20
3  20
4  2
5  117
6  1
7  78
8  1
DB<2> x scalar localtime()
0  'Mon Mar 20 20:47:27 2017'
```

Time::Piece can help. It lets you write code that looks like this instead (to quote the docs):

```
use Time::Piece;

my $t = localtime;
print "Time is $t\n";
print "Year is ", $t->year, "\n";
```

`localtime()` now returns an object that has methods you can call to retrieve the part of the time structure you want (for example, "$t->hour" will return the current hour). It also gives you some convenience methods like "->isdst" to determine if it is currently daylight savings time. Check out the documentation for the full list.

TieFile—In a previous column many moons ago I went gaga for the cool and cruel things you can do with the Perl `tie()` function. This function lets you essentially run arbitrary code as part of the process of retrieving and setting variable contents. For example, instead of getting a value from memory when asking for `$weather{'Boston'}`, Perl could query some weather service on the Web and return the information instead. `Tie::File` isn't that futuristic, but it can do something pretty cool. If you use it like this:

```
use Tie::File;
tie @array, 'Tie::File', filename
```

you can access lines of the file (getting and setting) by just reading or changing array values. The doc gives these examples:

```
  $array[13] = 'blah';     # line 13 of the file is now 'blah'
  print $array[42];        # display line 42 of the file
```

If you truncate the array by changing its size, so too does the file change. Your other standard array operations (`push`, `pop`, etc.) behave exactly as you would expect. Oh, and here's a fun tidbit from the doc:

```
The file is not loaded into memory, so this will work even for
gigantic files.
```

FileCopy—Yup, does what you would expect.

FilePath—Probably not what you would expect. Use this to create or delete directory trees.

FileTemp—Use this, and probably only this, for dealing with temporary files.

TextBalanced—If you ever read Jeffrey Friedl's *Mastering Regular Expressions* you know that trying to extract things from delimited text (for example, some text that has parentheses around it, like this one) can be less than straightforward. This comes up in all sorts of situations, like when parsing HTML or XML, program source code, and so on.

TermANSIColor—I'm almost tempted not to mention this one because it has such a potential to be overused (thus allowing you to write code that outputs "angry fruit salad"), but I'm going to assume that we're all adults here and that with great power…

Yup, time to write code like (from the doc):

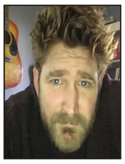
```
use Term::ANSIColor;
print color 'bold blue';
print "This text is bold blue.\n";
print color 'reset';
print "This text is normal.\n";
print colored("Yellow on magenta.", 'yellow on_magenta'), "\n";
```

Do me a favor and don't tell anyone where you got this superpower. On a serious note, I would commend you to consider that a larger part of the population than you probably think has some sort of color blindness (bring yourself up to speed about color blindness via a quick online search). Please consider this when writing code where the color of the output is significant and important.

And with that fun set of modules, I'm going to stop. Since I find myself on a plane too often it is entirely likely that this will be the first part in a several part series. Do let me know what you think of the idea. Take care, and I'll see you next time.

iVoyeur
2 Bits

DAVE JOSEPHSEN



Dave Josephsen is a book author, code developer, and monitoring expert. His continuing mission: to help engineers worldwide close the feedback loop. dave-usenix@skeptech.org

I guess you could say I'm between jobs at the moment. *I* won't say it, because I don't want to sound clichéd and self-conscious about being unemployed, but if *you* said it, it'd be fine.

You'd be right.

Don't worry. Everything's fine. Mostly. It's not like I was scandalously terminated or that I rage-quit in a righteous whirlwind of well-justified sanctimony. I kind of wish it were that interesting, but no, I loved them, they loved me, it was great. And yet, filled to the brim with what can only be described as a heaping pile of privileged old dude problems, I quit.

You see, I had this plan, or maybe it was more like a nagging daydream. I couldn't shake it. I have a little money in the bank—not really an impressive amount by my-startup-got-acquired standards, but enough to take a little time if I wanted, so I thought: "Why not just quit and drive away?"

I'd jump in my 30-year old truck and drive it north until it broke down. In that place, wherever it was, I'd talk to people who were physically standing in front of me. I'd read books made of paper, purchased from a physical store that sold books made of paper. I'd look at the clouds in the sky rather than the clouds on the other side of my VPN connection. I'd drink until my neurons realigned to real life—until *character* began to sound to my ears like a collection of personality traits rather than a Unicode rune, and *string* became a thing you tied stuff up with. I wouldn't think about JSON, or Jinja, JVMs, or how best to organize data into structures.

I know every millimeter of exactly how stupid that sounds. They have all that stuff right where I live. Books...clouds...strings...real life. But like I said, I couldn't shake it; like technical debt, it just seemed to keep growing, ominous and ever-present, until there was no other choice but to take a deep breath and wade in. I can hear you thinking *burnout* or *mid-life crisis*, and you're probably right. I have no idea what I'm doing. I can say, however, that I haven't bought a sports car, and I have no desire to write a novel, and anyway I can't help but feel like *suddenly he took a road-trip* is a pretty insipid mid-life crisis, so my money is on burnout.

I'm not super worried about putting a name on it, but I became utterly convinced that indulging myself in this sad, half-baked escapist scheme would cure me. Either I'd grow back some passion for this career I'd stumbled into so many years ago, or I'd get eaten by a bear. Either outcome seemed equally likely, and I was fine with that (as long as they never caught the bear). My point is, at some point I cognitively crossed this threshold where the daydream seemed less like selfish indulgence and more like life-saving necessity.

So instead of seeing a therapist like a reasonable person, I quit (having already burned up all my vacation days and then some). Not waiting for my two-weeks' notice to be up, I hit the road immediately. My team members were somewhat confused to suddenly find me in a Missouri coffee-shop at the next morning's stand-up meeting, but we're all work-from-homers anyway and my problem reports kept rolling in, so it wasn't a huge deal. Then, as Missouri became Illinois, and Illinois became Iowa, and eventually everything became South Dakota, I feel like it became somewhat normal, if not even a little entertaining for them.

And then finally my two weeks were up, and I awoke jobless and snowed-in, in Rapid City, South Dakota, my freedom finally secured, my escape complete, my insurance revoked. I didn't waste a single moment. I reached right into my bag, cracked open my laptop, and dug right in to Facebook's paper on in-memory time-series databases.

Sorry, I'm new at this burnout thing. I'm sure I'll get the hang of it eventually. On the bright side, at least I have something to share with you in this month's column.

Gorilla

If you haven't read Facebook's paper, "Gorilla: A Fast, Scalable, In-Memory Time Series Database," then you're really missing out [1]. They had a problem that is extremely common in our line—er, that is to say *your* line—of work. Namely, too many metrics.

Having outgrown `graphite`, many of us—er, you—turn to OpenTSDB, the google-scale map-reduce-for-metrics system. Facebook had reached this level several years hence, and their in-house analog of OpenTSDB [2] had grown to petabyte-levels of data. Their read latency had grown in kind, such that their 90th percentile read latency was seconds long.

Facebook's solution to this problem was to create a write-through in-memory cache system called *Gorilla*, which banks on a series of key observations to provide massive improvements to query-times without impacting writes.

Following that most fundamental of software engineering principles that states every problem can be solved with one additional layer of abstraction, Gorilla is inserted between the metrics-sending client nodes and Facebook's ODS data store. Accepting posted metrics in lieu of the real persistence layer, it proxies the data to the real back end while keeping a highly compressed in-memory copy for itself. Clients can then directly query Gorilla for rapid access to recently persisted data.

One of the aforementioned observations around which Gorilla was built is that recently stored data are more valuable than older measurements. This is not surprising, but Facebook quantified it, analyzing their own query habits and discovering that 85% of their query volume targeted data less than 26 hours old.

One way Gorilla was task-optimized for its user-base is, therefore, that it only holds 26 hours worth of data. In fact, Gorilla may be the single most thoroughly spec'd out monitoring system in the history of mankind, having been specifically designed to index two billion unique time series, ingest 700 million data points per minute, and service 40,000 queries per second, to name a mere few of its many overly specific sounding design criteria. The engineers at Facebook also designed it to be horizontally scalable and resilient against their most common failure scenarios, namely, individual node failures and network partitions affecting entire regions.

There are quite a few fascinating design features in the paper, but among them, their novel approach to data compression certainly stands out.

Most metrics-oriented monitoring systems report metrics as a tuple of name (string), date (int), and value (double). Another fundamental observation the Facebook engineers made was that the timestamps in the tuples submitted to their ODS metrics system were highly periodic (data arrived on regular intervals). They therefore reasoned that rather than storing raw timestamps for every measurement in a given series, they could instead store the delta of the delta of the timestamps. For example, a hypothetically perfect time-series that reported every 60 seconds would always have a delta of 60 and delta-of-deltas of 0. By comparison, a somewhat malfunctioning time series might report at: 2:30:00, 2:31:01, and 2:31:59. These deltas would be 60, 61, and 59, and the subsequent delta-of-deltas would be 0, 1, and -1.

Writing a periodic header with a real epoch value every two hours or so would hypothetically enable you to store a much smaller numerical representation of the ongoing datestamps for a given series (0 instead of an epoch value like 1490064897). I say hypothetically because 0 actually requires `len(int)` bits of memory to internally represent. In other words, inside the computer, 0 is actually 0000000000, because computers are dumb, so in real life, storing 0 instead of 1490064897 doesn't actually save you any space.

The Facebook engineers therefore eschew generic types for their own variable-length binary encoding to store these delta values. Their design works like this (where *D* is the value of the delta-of-the-delta for a given measurement):

- If *D* is zero, store binary 0 (only requires 1 bit of memory).
- If *D* is between [-63, 64], store '10' followed by the value (7 bits).
- If *D* is between [-255, 256], store '110' followed by the value (9 bits).
- If *D* is between [-2047, 2048], store '1110' followed by the value (12 bits).
- Otherwise store '1111' followed by *D* using 32 bits.

Because the measurement values themselves begin life as double-precision floats rather than ints, their compression is more complex, but only slightly more so. The values are XOR'd instead of delta'd, and a similar variable-length binary encoding is employed that is based on discarding the insignificant digits of the resultant XOR'd values.

The paper reports that 96% of all inbound timestamps compress to a single-bit (i.e., stuff is mostly '0') due to the periodicity of the input data (based on a random sampling of 440,000 real-world series in use at Facebook). The paper goes on to find that, for sample series that are recorded long enough (two hours seems to be the sweet spot), the double-precision floating point measurement values can achieve a compression ratio of 1.37 bits per data point.

Assuming 64-bit doubles, that's 460800 uncompressed bits in a two-hour series to 9864 compressed, or a 46x compression ratio though the paper only claims a 10x compression improvement. I infer the 10x number was derived by comparing Facebook's Gorilla implementation's overall storage footprint to that of their HBase system.

Gorilla has also achieved the scalability, fault-tolerance, and impressive sub-millisecond read latency goals set forth by its designers, though it's worth noting that a successful read yields compressed data (decompression is handled client-side).

Again, if you haven't read it, it's pretty fantastic work, and you should have a look. I mean *I* read it, and I don't even work with computers, so I don't know what *you're* waiting for. It's also worth noting that there is already some subsequent work based on Gorilla. Facebook itself has open-sourced a general-purpose reference implementation of the Gorilla daemon plus client software called Beringei [3].

Other examples include libraries that implement Gorilla's compression algorithm, like `go-tsz` [4] as well as some open-source data stores like Raintank's MetricTank [5], which uses Gorilla's compression algorithm inside its own Cassandra-based storage back-end.

By the time you read this, I'll hopefully still be happily unemployed—but I kind of doubt it. I'll hold out as long as I can. Think of me when you look at the northern hemisphere.

Take it easy.

References

[1] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, K. Veeraraghavan, "Gorilla: A Fast, Scalable, In-Memory Time Series Database," in *Proceedings of the VLDB Endowment*, vol. 8, no. 12 (August 2015), pp. 1816–1827: http://www.vldb.org/pvldb/vol8/p1816-teller.pdf.

[2] V. Venkataraman, C. Thayer, L. Tang, "Facebook's Large Scale Monitoring System Built on HBase," Strata + Hadoop World 2012: https://conferences.oreilly.com/strata/stratany2012/public/schedule/detail/25540.

[3] Berengei, Facebook's Gorilla client: https://github.com/facebookincubator/beringei.

[4] `go-tsz` compression library: https://github.com/dgryski/go-tsz.

[5] Metric Tank: https://github.com/raintank/metrictank.

Turning Problems into the Known

JEANNE SCHOCK

Jeanne Schock has a background in Linux/FreeBSD/Windows system administration that includes working at a regional ISP, a large video hosting company, and a DNS and top-level domain registry services provider. She is a certified Expert in the IT Infrastructure Library (ITIL) process framework with in-the-trenches experience in change, incident, and problem management. Jeanne also has a pre-IT academic and teaching career and is an experienced trainer and public presenter. jeanneschock@gmail.com

System administrators rightly associate problem management with identifying and removing the underlying root causes of incidents. But finding and resolving root causes requires resources and interdepartmental political will that are not always available. Living with a documented yet unresolved problem is not a failure of your team. Nor is it a failure of process. The real failure would be to overlook the smaller, incremental improvements that can be gained by addressing the factors and conditions that contribute to incidents. Make sure that your problem process is focused on outcomes beyond technical solutions to root causes: knowledge gain, effective decision-making, elimination of negative activities such as finger-pointing, and the only metric that really matters in IT—customer satisfaction.

The Problem Management Process

We define an incident as a disruption to normal IT service, and a problem as the unknown cause of an incident. It is unknown because we don't know either what caused the incident or how to prevent it from happening again. Incidents don't become problems; rather, problems cause incidents. A problem management process manages the life cycle of problems: identification/categorization/prioritization, establishing workarounds, tracking activities, changes in status and decisions, investigating and determining causes, finding and implementing permanent solutions. One purpose of the process is to prevent recurrence of incidents through permanent solutions. A second, often-overlooked, purpose is to minimize the impact of unavoidable incidents. Reasons why future incidents might be unavoidable include resource limitations, technical limitations, or practical decisions to live with the problem based on a cost to benefit analysis.

Establish Good Workarounds

Where does this leave a technical team? With a good workaround, we can both mitigate the impact of an unresolved problem and buy more time for implementing a permanent solution. Our objectives are to quickly detect the conditions indicative of a recurrence of the incident, to reduce the mean time to resolve, and to continually improve the workaround as we learn more about the problem. The workaround may be good enough to complete a project that entirely eliminates the software or application or network device that underpins all other causes of the incident. We speak disparagingly of workarounds as "duct tape," because we know from experience that duct-tape solutions usually become permanent. That's why it is critical to build into your problem management process routine reviews of all workarounds. Define the review schedule based on the priority of the problem: e.g., high-weekly, low-monthly. Then define the behavior that you want and write a policy to match: all workarounds must be continually reassessed for effectiveness in mitigating the impact of the incident, cost in staff time to maintain, and level of confidence that it will continue to work.

Document and Track Known Errors

If you have recorded the problem, and you either have a workaround or preliminary investigation hints at the nature of the problem, you can declare a known error. This may sound like an attempt to mask the problem. But turning an unknown cause of incidents into a known error has value. Which situation would you prefer: a published list of known errors, linked to incidents they are suspected to have caused, perhaps even with a confirmed workaround, or undocumented problems that may re-manifest at any time in the form of another service interruption? Turning the unknown into the known is time well spent. You already use the known error concept when you are waiting for in-house developers or vendors to patch software bugs and vulnerabilities. In these examples, the known error is well-understood, and you are just waiting for a solution. But you can also say that you have a known error when you have even a vague understanding of the correlation between causes of the incident. The objective isn't necessarily to solve every single root cause, but to make good decisions and to track and continually reassess those decisions.

This may require you to shift your thinking about your role as problem solver to provider of strategies, information, and tools that can help your company manage problems. The known error database (which could simply be a list maintained in a Google doc or wiki page) informs better risk analysis and decisions around authorization of change requests. It is a useful reference for on-call shift changes. As a list of "improvement opportunities," it helps with departmental planning and goal setting. Tracking a known error includes linking new incidents, or even older incidents that you come to realize were likely caused by the same problem. This helps build a case for developers to prioritize a bug fix, or convince management to re-prioritize resources in ways usually reserved for root cause investigations after large or embarrassing incidents. If you find your team is tracking an increasing number of known errors that are beyond your control to repair, try building relationships outside your team that you can leverage to resolve some of those problems.

Root Cause

Let's talk about root cause. System administrators know well that complex technologies most likely have multiple correlated causes. At a purely semantic level, you should be comfortable replacing "the root cause" with "multiple root causes." Don't assume that you must always delve into an investigation for underlying causes. Think of the problem process as one of the many tools in your toolbox that you can use for improving the services for which you are responsible. Improvement can come in the form of large-scale leaps, but it is more likely to result incrementally from small, iterative steps forward. Ask yourself, what is within your control to improve:

- What tools could you build that would enable you to detect the conditions that were present during past manifestations of the problem?
- Can you automate the workaround?
- What tools or knowledge would reduce the time required to resolve the incident?
- Can you reduce the conditions that contribute to slow troubleshooting by asking the DBAs to train your team over lunch on replication errors?
- Can you be prepared to collect more data during the next occurrence of the incident that would enhance your understanding of the problem?
- Can you revise alerts and escalation procedures to get the right people looking at the incident faster?
- Can you reduce dysfunctions by improving your team's relationship with teams that are both ahead and behind you in the value chain?

These are all actions that can be taken that do not require root cause investigation or permanent solutions. And they should be acceptable outcomes for teams with limited resources.

Human Error

Can you mitigate the risk of human error? Human error is a common trigger or contributing factor to incidents. Human error is an inevitability, just like hardware failure. Systems that are not designed and built for resilience in the face of inevitabilities are incidents waiting to happen. There is no value in assigning the root cause of an incident to human error, as there will never be a permanent fix. We can't eliminate human error, but we should anticipate it and work to mitigate its impact.

Conclusion

How we respond to human error and to problems says a lot about our culture. We should expect both and have effective, established, and well-understood processes in place that enable good decisions and positive outcomes. The ideal outcome of any problem investigation or postmortem is finding and solving root causes. But don't let the perfect become the enemy of the good. Improving the experience of your users and customers and providing cost-effective solutions that benefit your company should be the drivers of any IT process, including problem management.

For Good Measure
To Burn or Not to Burn

DAN GEER AND JON CALLAS

Dan Geer is the CISO for In-Q-Tel and a security researcher with a quantitative bent. He has a long history with the USENIX Association, including officer positions, program committees, etc. dan@geer.org

Jon Callas is a cryptographer, software engineer, UX designer, and entrepreneur. He is the co-author of many crypto and security systems, including OpenPGP, DKIM, ZRTP, Skein, and Threefish. He has co-founded several startups, including PGP, Silent Circle, and Blackphone. He has worked on security, UX, and crypto for Tesla, Kroll-O'Gara, Counterpane, and Entrust. He is fond of Leica cameras, Morgan sports cars, and Birman cats. His photographs have been used by Wired, CBS News, and The Guggenheim Museum. jon@callas.org

There is no question that vulnerabilities are important. There is a rich history of vulnerabilities and of their use, yet if that history is a signal, then it is a noisy one. Inferences drawn from agreed upon history of vulnerabilities are still the source of quite conflicting interpretations—proof that it is hard to reduce the question of vulnerabilities to a simple set of inferences. Experts are less likely to agree on a simple set of inferences about vulnerabilities than the non-experts. Often as not, experts claim that all other expert opinions besides theirs are simplistic rather than simple (and that "I have discovered a truly remarkable proof which this [Tweet] is too small to contain").

At the time of writing this column, a new report from RAND had just appeared. The RAND report [1] (which you *must* read) is the best look yet at the question of vulnerabilities as seen through the lens of vulnerabilities not yet known. As should be expected, a part of its conclusions were immediately dismissed as simplistic by Those Who Tweet.

Three terms from the field of epidemiology may help us think about the life cycle of vulnerabilities. First is *incidence* (I), which is the number of new cases of disease which appear per unit time. Second is *prevalence* (P), which is the number of infections at a given time. Third is *duration* (D), which is the time interval between when infection appears and when that infection is cured. In a stable population, those three are mutually redundant—knowing any two of them allows you to determine the third: $I*D=P$. For the defender whose job is to treat disease, prevalence is the measure of workload. For the defender whose job is to prevent disease, changes in incidence are the measure of whether their work has or has not been successful. For the defender whose job is to judge the societal cost of disease, duration is likely the focus. Analysts studying risk factors for the disease must use incident cases within a given time interval rather than prevalent cases at a given time, i.e., near real-time detection matters: cf., "Neyman" bias.

But each of I, P, and D are for the single disease case. Thinking of the offender as an opportunistic infection, that is to say a secondary (intentional) infection that exploits an already infected patient, a patient whose existing infection(s) make that patient susceptible (vulnerable) to additional infections, the defender might come to think in terms of global immune system failure more than the lack of some specific antibody. Just as there is no human immune to all human diseases, RAND notes that

> [a]ny serious attacker can always get an affordable zero-day for almost any target. The majority of the cost of a zero-day exploit does not come from labor, but rather the value inherent in them and the lack of supply.

Which reminds us that we are talking about sentient opponents, not stray alpha particles or metal fatigue. We are inside a natural experiment, not some controlled laboratory work.

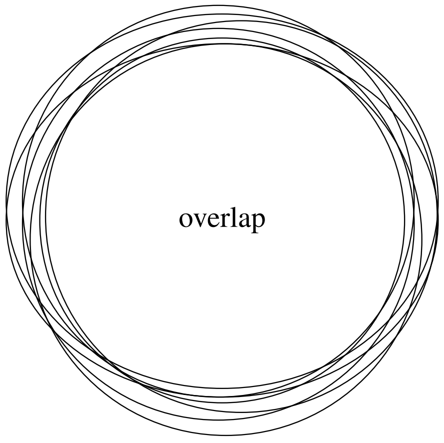


Figure 1: The threat landscape includes the overlap of different actors' toolsets.

So be it, but it is that existence of multiple possible infections—due to multiple possible infectious agents—that we find worth comment. Consider the set of tools (vulnerabilities, exploits, software, etc.) that *A* has, and the set that *B* has, and *C*, and *D*, and *E*,…Make a Venn diagram of these; their union is the threat landscape (Figure 1). What is especially noteworthy is not the area common to all the closed curves in the diagram but the areas outside that joint intersection.

If any actor holds their tools secret, then what is in that actor's subset that is not in the union of everyone else? If that subset is small compared to the whole set union, and then the marginal cost to you of any actor, state or otherwise, holding such a cache of tools is small. Yet to any given attacker, they view their whole tool set as an asset and are loathe to give it up. Thus, perhaps counterintuitively, they tend to view their tool set as being worth a lot because it is their whole set, but the defender would view it as not being worth much because the marginal cost to the defender of that set being held secret is only the cost that can be imposed by the tools unique to it.

If you capture someone else's tool set, then the cost to you of burning (exposing) all the other entity's tools is the intersection of that tool set with yours. The smaller the intersection, the smaller the cost. When you burn someone else's tool, you signal to them that they lost control of it. Beyond that effect, burning the other entity's tools also burns them for anyone else in the intersection set, which alerts all the actors with intersections with the set that you burned that there is some other-party intelligence that they didn't know about.

Earlier, we spoke of the defender whose job is to think about prevention. If the number of tools that are common to many actors' tool sets is a large fraction of all such tools, that is to say that the joint intersection in the Venn diagram contains the greater share of all known tools, then burning them would greatly reduce the firepower available to all actors in the aggregate, including you.

As with all modeling exercises (which is what we are doing), there are assumptions. Assumptions are not bad, but the better grade of analyst will make them, get an analysis done, and then test whether the result of the analysis was or was not critically dependent on its assumptions.

Aitel and Tait's superb article [2] on the conditions under which a free-world nation-state should reveal vulnerabilities to their authors of record has especially telling conclusions in this regard, which follow from two axioms (assumptions). The first axiom is that the free world's most dangerous opponents do not have the constraints on their discovery, use, retention, and disclosure of vulnerabilities that free world nations do. The second axiom is that the vulnerabilities that are a crucial threat to the software base of one nation are different from the vulnerabilities that are a crucial threat to another. The Venn diagram for "How much of my code base is also your code base?" is not something we have in hand, but we strongly suspect that the parts that are country-unique are the greater half, and if that is the case, then it biases the equation away from disclosing vulnerabilities to vendors of code you don't use. If opponent countries are investing in home-grown software as a strategic defense, then the bias away from disclosure to their vendors only grows stronger.

It must be acknowledged that part of our problem is the rapid rate of change which "we" otherwise praise. Beginning with Ozment and Schecter's 2006 paper [3], we have known that stable code bases under stable oversight do cleanse themselves of vulnerabilities. Clark et al. have since shown in measurable ways [4] that while the goals and compromises necessary to compete in a global market have made software reuse almost compulsory, "familiarity with a codebase is a useful heuristic for determining how quickly vulnerabilities will be discovered and, consequently, that software reuse (exactly because it is already familiar to attackers) can be more harmful to software security than beneficial." The language theoretic security group [5] has indirectly shown that the closer the code is to Turing-complete, the more likely it is to be reused, i.e., the very code that has the greatest probability of being reused is the code that has the greatest probability of being rich enough in complexity to enhance exploitability. In medical care, this is called "adverse selection" (the better the care you provide, the sicker are the people who throw themselves on your mercy).

Which leads us to the—repeat, the—fundamental question with respect to vulnerabilities: are they sparse or are they dense [6]? RAND's conjecture is that "the overlap between what is found and disclosed publicly and what is found and kept privately appears to be relatively small… [which] implies that vulnerabilities may either be dense or very hard to find," to which we might add a third option, that vulnerabilities are essentially sparse but aggregate code volume is increasing so fast that there are many more vulnerabilities than there are researchers to find them. Meanwhile, the fallout from the DARPA Cyber Grand Challenge [7] may well answer the question of sparse vs. dense and thus tell us whether or not to look for vulnerabilities (they are sparse so each killing brings them closer to extinction vs. they are dense so killings have negative return on investment).

End-of-life code bases are a special case; because they remain "unimproved" (stable), every vulnerability killed decreases the number of vulnerabilities extant. As such, it would be useful to patch zero-day vulnerabilities in no-longer-maintained software, especially for code that remains in widespread use [8].

But knowing what we know now, as underscored by Aitel and Tait, the—repeat, the—central policy question is this: are the vulnerabilities that will take you down the same vulnerabilities that will take down your opponent? If they are different, then disclosing to vendors vulnerabilities in code upon which you do not rely is an act of unilateral disarmament. Releasing a vulnerability is an aggressive act if you know someone else has it—and an intelligent move.

References

[1] L. Ablon and T. Bogart, "Zero Days, Thousands of Nights: The Life and Times of Zero-Day Vulnerabilities and Their Exploits": www.rand.org/content/dam/rand/pubs/research_reports/RR1700/RR1751/RAND_RR1751.pdf.

[2] D. Aitel and M. Tait, "Everything You Know About the Vulnerability Equities Process Is Wrong," Lawfare, August 18, 2016: www.lawfareblog.com/everything-you-know-about-vulnerability-equities-process-wrong.

[3] A. Ozment and S. Schechter, "Milk or Wine: Does Software Security Improve with Age?" in *Proceedings of the 15th USENIX Security Symposium (USENIX Security '06)*, pp. 93–104: www.usenix.org/legacy/events/sec06/tech/full_papers/ozment/ozment.pdf.

[4] S. Clark, M. Collis, M. Blaze, J. M. Smith, "Moving Targets: Security and Rapid-Release in Firefox": seclab.upenn.edu/sandy/movingtargets_acmccs14_340.pdf.

[5] Language-theoretic security: http://langsec.org.

[6] B. Schneier, "Should U.S. Hackers Fix Cybersecurity Holes or Exploit Them?": www.theatlantic.com/technology/archive/2014/05/should-hackers-fix-cybersecurity-holes-or-exploit-them/371197.

[7] Defense Advanced Research Projects Agency, Cyber Grand Challenge, August 4, 2016: archive.darpa.mil/cybergrandchallenge.

[8] D. Geer, "On Abandonment," *IEEE Security and Privacy* (July/August 2013): geer.tinho.net/ieee/ieee.sp.geer.1307.pdf.

/dev/random

ROBERT G. FERRELL

Robert G. Ferrell is an award-winning author of humor, fantasy, and science fiction, most recently *The Tol Chronicles* (www.thetolchronicles.com).
rgferrell@gmail.com

Let us now consider backups: boon or bane? "Boon obviously," I hear you respond with more than a soupçon of righteous indignation. Fair enough. Having an additional copy of your critical data is objectively better than the alternative; I agree. Provided, of course, that said copy is truly a copy. If, on the contrary, it is no more than a hollow shell filled with empty zeroes, that "backup" may prove less salubrious. Allow me to elucidate.

In 1997 I was a UNIX system administrator and email/DNS monkey at USGS headquarters in Reston, VA. Not long after I started that job a scientist came in and reported that he'd experienced a catastrophic laptop drive failure while in the field and needed to restore from one of the backups conducted every few months during brief visits to headquarters. I pulled the appropriate DAT tape, properly labeled and stored, and began the restore procedure. Much to my chagrin and horror, while the tape headers and log entry for the backup looked perfectly normal, there was no actual data therein residing. Frantically, I tried everything I could think of to recover at least a partial image, but it was no use. There simply wasn't anything there to recover. That volcanologist lost three full years of field research because someone (not me, thankfully) didn't bother to check the integrity of the backup process.

The point of this sad story is to show that backups aren't always what they promise to be. If you trust them without verification, sooner or later you will regret it. This cautionary principle can of course be applied across a swath of IT-related activities; at its most fundamental it warns against complacency and making presumptions. While backups themselves are, overall, Good Things To Have Around, betting the farm on those backups existing simply because they appear to exist is skating on exceptionally thin metaphorical ice.

Even properly executed backups aren't a universal panacea. There are times when you simply don't want everything recorded accurately for posterity. One might reasonably presume that the sorts of activities best forgotten are not likely to be found in a routine corporate disk image, true, but even this is not a foregone conclusion. Mistakes, indiscretions, bad ideas, erroneous data, miscommunications, poorly conceived notions, fumbling, hemming, hawing, tangents, memos you wish you hadn't written, memos you wish you hadn't read—all of these and more might be better off consigned to oblivion.

Where am I leading this parade of the obvious? Right past my flea market of invention, naturally. The idea that backing up data is a simple binary decision is outdated and probably runs contrary to all sorts of sound business practices, I guess. If not currently, then I hereby instantiate said practices such that they can be run contrary to for the purposes of furthering this diverting narrative. It feels good to take charge of my own twisted destiny.

You know how in some operating systems each file has various flags that can be set? "Archive," "Read Only," "Certified Organic," and so on? I propose we add one for "Backup Suitability." It'll have to be a metadata field rather than a simple binary flag, though. It would need at least four or five possible values, to denote Retention Desirability Quotient.

This value would range from 5 (Totally Keep This Data For All Eternity, Possibly Longer) to 0 (Civilization As We Know It Will Be Irrevocably Harmed If This File Is Not Deleted Immediately And With Extreme Prejudice). When the backup program encounters these flags, it acts accordingly.

You might at this juncture feel compelled to point out that there are already backup products available that feature very similar, if less sarcastic, functionality. To this I can only reply, "bah." My proposed program goes further, much further. There's also a predictive component that will scan each file, no matter the format, for potentially embarrassing content and—here's the best bit—report when and to what extent it will interfere with your future life. It can even modify or extract those damaging areas based on a wide range of user-configurable filters. Think of it as a sort of personal *Minority Report*. In "Forensic Avoidance" mode the program copies the dodgy file bit by bit into memory, makes the appropriate changes, and writes it to the backup image without modifying any of the administrative metadata: instant file integrity without all that messy undesirable content. The program download, incidentally, is free. The client is charged only when a file is actually backed up, on a sliding scale depending on degree of "posterity assurance" undertaken. It's all very scientific and stuff.

The whole "subscription" model for software bothers me, now that we've brought it up. It's like rent-to-own, except you never get to the "own" part. As if online privacy hasn't taken enough of a beating with adware, trackers, consumer profiling, constant account compromises, draconian digital "rights" management, and shadowy government data slurping on a beyond-massive scale, now software companies want us to borrow their products temporarily in exchange for radio tracking collars on our most intimate computer use habits.

Since we seem to have slopped our way over into targeted marketing now, let me state without fear of being regarded in any way as an original thinker that it cannot, statistically, be long before even the prime real estate of our sleep periods is being developed for advertising purposes. Do you have liberating flying dreams? Airlines and exotic travel destinations will pay handsomely for ads plopped down into those. Leave home without your pants? Clothing manufacturers have you covered.

If you're thinking that the technology to beam these ads directly into your neural landscape from some advertising studio does not exist, you're (probably) correct. However, they don't need said technology to achieve oneiric product placement nirvana. All they require is a series of carefully constructed subliminal implants: essentially, a buffer overflow for the brain. They inject the right code via TV or streaming video and it runs in batch mode in the heap of your subconscious mind. Corporations will now control your nighttime data dumps even more stringently than before. Nowhere is safe; nothing is sacred.

> To sleep: perchance to dream: ay, there's the market;
> For in that consumer's sleep what dreams may come
> When they are no longer able to change channels or
>   surf away,
> Must give us profit...

Once again is the immortal bard shamelessly and tastelessly paraphrased for petty purpose. Try haddock, and let's see what slips the dogs wore.

NOTES

USENIX Member Benefits

Members of the USENIX Association receive the following benefits:

Free subscription to *;login:*, the Association's quarterly magazine, featuring technical articles, system administration articles, tips and techniques, practical columns on such topics as security, Perl, networks and operating systems, and book reviews

Access to *;login:* online from December 1997 to the current issue: www.usenix.org /publications/login/

Discounts on registration fees for all USENIX conferences

Special discounts on a variety of products, books, software, and periodicals: www .usenix.org/member-services/discount -instructions

The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers

For more information regarding membership or benefits, please see www.usenix.org /membership or contact office@usenix.org. Phone: 510-528-8649.

The State of the USENIX

by Casey Henderson, USENIX Executive Director

When I assumed the role as Executive Director, I had dreams of scaling USENIX globally to better serve our constituents. It is exciting that just over three years later, thanks to collaboration by staff and volunteers, USENIX is actively engaged in hosting events at international venues. In fact, as I write this, I am en route to Europe to visit prospective locations for SREcon18 Europe/Middle East/Africa, and will soon leave for SREcon17 Asia/Australia in Singapore.

I also had dreams of creating events that would more directly address the challenges of gender diversity in our field. Now, among the SREcon events and the security-focused Enigma conference, we have succeeded in engineering environments with diverse leadership matched by attendees who are engaged in diversifying their respective fields as they strive to scale and secure systems. We plan to transfer the lessons learned from these events to our systems research-focused conferences.

These initiatives are rooted in the Board and staff's twin goals of securing USENIX's sustainability and maintaining its relevance through new ways of fulfilling our mission. We face financial limitations, which challenge our small but mighty staff to be creative and flexible. I often receive questions from our constituents about why our Web site can't do *this* and why a conference can't be *there*. While we appreciate and keep track of your ideas and input, the answers usually tie back to a limited budget with ambitious plans for serving myriad communities. This makes it all the more rewarding to report on the progress we've made in developing new events, nurturing emerging communities worldwide, and engaging in new processes.

We currently serve 40% more attendees annually than we did five years ago, and are exploring new approaches to provide the best possible service with our existing staff. For example, you may have experienced self check-in at a recent USENIX event; we are revamping the arrival process to improve the attendee experience, reduce the amount of staff time needed to prepare for events, and ship fewer materials to events. By strategically spending to meet a variety of goals, we are attempting to meet the needs of constituents while watching the budget.

Soon you'll see more changes to the USENIX Web site, beyond the recent revamping of our conference pages. You'll notice discussion about *;login:* as we consider the most effective ways to serve its audience. You'll see how our events evolve as we continue to engage in experiments: HotSec has a new model in 2017; FAST and NSDI will co-locate in Boston in 2019; and SOUPS will continue to find its home at USENIX, but will shift from ATC to Security in 2018.

I hope you'll join the Board and staff teams for the Annual Meeting in Santa Clara on July 13 to find out more about our plans and provide us with feedback.

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710

POSTMASTER
Send Address Changes to *;login:*
2560 Ninth Street, Suite 215
Berkeley, CA 94710

PERIODICALS POSTAGE
PAID
AT BERKELEY, CALIFORNIA
AND ADDITIONAL OFFICES

26TH USENIX SECURITY SYMPOSIUM

AUGUST 14–16, 2017 • VANCOUVER, BC, CANADA

The USENIX Security Symposium brings together researchers, practitioners, system administrators, system programmers, and others interested in the latest advances in the security and privacy of computer systems and networks. The Symposium will span three days, with a technical program including refereed papers, invited talks, a poster session, a Work-in-Progress session, a Doctoral Colloquium, and Birds-of-a-Feather sessions (BoFs).

The following co-located events will occur before the Symposium:

WOOT '17: 11th USENIX Workshop on Offensive Technologies
August 14–15

CSET '17: 10th USENIX Workshop on Cyber Security Experimentation and Test
August 14

FOCI '17: 7th USENIX Workshop on Free and Open Communications on the Internet
August 14

ASE '17: 2017 USENIX Workshop on Advances in Security Education
August 15

HotSec '17: 2017 USENIX Summit on Hot Topics in Security
August 15

Register by July 24 and Save!
www.usenix.org/sec17