;login:

SPRING 2018 VOL. 43, NO. 1

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

UPCOMING EVENTS

SREcon18 Americas
March 27–29, 2018, Santa Clara, CA, USA
www.usenix.org/srecon18americas

NSDI '18: 15th USENIX Symposium on Networked Systems Design and Implementation
April 9–11, 2018, Renton, WA, USA
www.usenix.org/nsdi18

ScAINet '18: 2018 USENIX Security and AI Networking Conference
May 11, 2018, Atlanta, GA, USA
www.usenix.org/scainet18

SREcon18 Asia/Australia
June 6–8, 2018, Singapore
www.usenix.org/srecon18asia

2018 USENIX Annual Technical Conference
July 11–13, 2018, Boston, MA, USA
www.usenix.org/atc18

Co-located with USENIX ATC '18

HotStorage '18: 10th USENIX Workshop on Hot Topics in Storage and File Systems
July 9–10, 2018
Submissions due March 15, 2018
www.usenix.org/hotstorage18

HotCloud '18: 10th USENIX Workshop on Hot Topics in Cloud Computing
July 9, 2018
Submissions due March 13, 2018
www.usenix.org/hotcloud18

HotEdge '18: USENIX Workshop on Hot Topics in Edge Computing
July 10, 2018
Submissions due March 20, 2018
www.usenix.org/hotedge18

27th USENIX Security Symposium
August 15–17, 2018, Baltimore, MD, USA
www.usenix.org/sec18

Co-located with USENIX Security '18

SOUPS 2018: Fourteenth Symposium on Usable Privacy and Security
August 12–14, 2018
www.usenix.org/soups2018

WOOT '18: 12th USENIX Workshop on Offensive Technologies
August 13–14, 2018
www.usenix.org/woot18
Submissions due May 30, 2018

ASE '18: 2018 USENIX Workshop on Advances in Security Education
August 13, 2018
www.usenix.org/ase18
Submissions due May 8, 2018

CSET '18: 11th USENIX Workshop on Cyber Security Experimentation and Test
August 13, 2018
www.usenix.org/cset18
Submissions due May 10, 2018

FOCI '18: 8th USENIX Workshop on Free and Open Communications on the Internet
August 14, 2018
www.usenix.org/foci18
Submissions due May 24, 2018

HotSec '18: 2018 USENIX Summit on Hot Topics in Security
August 14, 2018
Lightning Talk submissions due June 11, 2018
www.usenix.org/hotsec18

SREcon18 Europe/Middle East/Africa
August 29–31, 2018, Dusseldorf, Germany
Submissions due April 3, 2018
www.usenix.org/srecon18europe

OSDI '18: 13th USENIX Symposium on Operating Systems Design and Implementation
October 8–10, 2018, Carlsbad, CA, USA
Abstract registration due April 26, 2018
www.usenix.org/osdi18

LISA18
October 29–31, 2018, Nashville, TN, USA
www.usenix.org/lisa18
Submissions due May 24, 2018

Enigma 2019
January 28–30, 2019, Burlingame, CA, USA

FAST '19: 17th USENIX Conference on File and Storage Technologies
February 25–28, 2019, Boston, MA, USA
www.usenix.org/fast19
Submissions due September 26, 2018

;login:
SPRING 2018 VOL. 43, NO. 1

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

I've long been fascinated by hardware. That fascination was re-awakened by Tom Van Vleck's letter to the editor in this issue. Tom is a Multician (multicians.org), and he wrote to us with comments about my interview with Peter G. Neumann in the Winter 2017 issue of *;login:*.

When I was fortunate enough to assemble my first (nearly UNIX) system [1], it included a 34 MB Seagate drive with the ST-506 interface. The disk controller was not part of the hard drive, as it is today. Instead, the disk controller sent commands—such as seek-inward, switch to head two, read sector headers until sector 10 is reached, then write to sector 10, over a 34 pin control cable—and read or wrote data over a separate 20 pin cable. Device driver writers had to consider issues like how fast to seek, how many blocks to skip between reading or writing to allow the CPU to finish with the previous operation, and handling the bad block map. The latter was truly a PITA, as it appeared as a printed label on the hard drive case and had to been entered, as block numbers, when formatting the drive. Even worse, the controller actually was responsible for sending or receiving analog signals for writing or reading, and that meant that you could only read a hard drive with the controller that had originally done the writing.

By the time ATA [2] became popular, hard drives included their own controllers, and instead of two cables, we only needed a single 40 pin cable that could be used to attach two drives. Each drive had a jumper to determine whether it was a master or not (device 0 or 1), and getting this wrong meant your system wouldn't boot. But having the controller built into the drive was a huge leap forward, as you could now move drives between host adapters. The host adapter was just a 16-bit ISA bus relay for commands and data between the bus and the drive's onboard drive controller. As the ATA standards evolved, the drive controllers became more sophisticated, able to understand SCSI commands.

The Small Computer System Interface [3] (SCSI, pronounced "scuzzy") required an even smarter drive controller. Up to seven devices, plus a host adapter, could be connected to a SCSI cable, and each drive had to be capable of bus arbitration. The SCSI standard also allowed the drives to queue up multiple commands.

SATA, which means serial ATA, uses a four pin cable, with commands and data being sent serially rather than in parallel. Just as PC busses have moved from the parallel ISA bus to the PCI busses that support many simultaneous serial channels, SATA achieves higher data rates by moving away from parallel busses.

And somewhere along the way, disk vendors quietly changed how sectors were accessed. Except with really old interfaces, like the ST-506, disks presented an array of blocks. The operating system was responsible for writing blocks to the most appropriate free block, and the OS did its best to write sectors that would be read in sequence together later, or at least be located in nearby tracks, for better performance.

Since around 1999, hard disks accept Logical Block Addresses (LBAs) instead of block numbers. The hard disk then maps the LBA to a physical block address, a bit like flash translation layers (FTL) work. This change had two effects: the disk controller needed to become smarter, and the operating system no longer had control over disk layout. File systems like

ext4 and BSD's fast file system (FFS) create cylinder groups, and associate directories and free block lists within these cylinder groups, to speed up both reading and writing. But the disk controller is unaware of these distinctions and just stores blocks using its own algorithms. These algorithms can even move blocks later, for example, if a group of blocks is often read in sequence.

I certainly thought that the operating system should have control over block placement, but the disk vendors saw things differently. They wanted to create the cheapest, most efficient drives in order to remain competitive, and for them, that meant taking control away from the operating system developers.

Disk vendors continue to leverage the CPUs and memory on disk drive controllers, and that's led to some interesting developments.

The Lineup

Timothy Feldman wrote a *;login:* article [4] in 2013 about Shingled Magnetic Recording (SMR), a new technique for increasing drive capacity. But SMR introduces its own set of issues, including highly variable write latencies. In this article, Tim explains a huge change to SMR drives: hybrid drives that include both conventional and SMR partitions. A new API means that the operating system can control how much of a drive appears as a conventional drive and how much as SMR, and it even allows changing the proportion of these two formats on the fly, with the disk controller moving blocks between the two regions as required. Tim explains the plans for the new APIs, changes that will be implemented in the drive's on-disk controller, but also need to be included in device drivers.

Carl Waldspurger et al. in their FAST '17 paper explain how to use very small samples of cache misses to determine how best to configure full-size caches. Cache miss rates are crucial when determining the optimal cache size. The authors create hashes of block numbers and select cache misses to record by using a portion of the hash space. Their creative use of hashes for getting a random collection of samples caught my attention.

I searched through the CCS [5] program for papers that match my criteria for articles that will have broad appeal and, out of a huge selection of security research, found two.

Luca Allodi narrates how he infiltrated a Russian exploit market. But his real point is how examining what is bought and sold tells us about which exploits are likely to be used in untargeted attacks. I liked Luca's exploit, managing to gain access to the market, and also how he explains what the going prices for exploits can reveal about which exploits are likely to be widely used.

Frank Li and Vern Paxson describe how they determined that it often takes a very long time for open source software to be patched. It's likely that commercial software is similar, but with open source, they could trace the time a patch appeared in the code, the time it was announced or distributed, and compare that with the time the vulnerability first appeared in the Common Vulnerability Exposure (CVE) ratings. Much of their work involved crafting the means of trawling online Git repositories as well as info about vulnerabilities, a task that would have been much more difficult without techniques for winnowing the data.

AbdelRahman Abdou and Paul C. van Oorschot volunteered to share their work on secure geolocation. While geolocation is commonly used, often for secure applications, geolocation is just as commonly spoofed. Abdou and van Oorschot lay out their proposal along with examples of how well it worked using sensors in PlanetLab.

Diptanu Choudhury offered an article about using eBPF. The extended Berkeley Packet Filter has been around for a few years and provides a secure method for injecting code within a live kernel. Choudhury's particular example involves the Express Data Path (XDP), which can be used for moving network methods, like a firewall or packet forwarder, into code that can access a network device's ring buffer, avoiding slow memory copies. Diptanu explains enough about eBPF to be helpful to anyone interested in beginning to use eBPF, as its programs can use triggers throughout the Linux and BSD kernels.

Tapasweni Pathak discusses her research into flaws in Linux 3.x kernels. Extending prior work, and using some of the same tools for searching through source code, Pathak explains her process and shows us, via graphs, just how well the kernel source is doing when it comes to bugs that can cause crashes or be exploited. For the most part, things have gotten better.

I interviewed Laura Nolan, a Google SRE and a past co-chair of SREcon Europe. Laura had helped me find authors for SRE articles, and I hoped to learn more about what it's like to be an SRE. Laura was definitely forthcoming, as well as providing a humorous example, before she went to Google, of what it's like to be a woman in this field. Laura also answers questions about why Google chose to use Paxos.

Bob Solomon interviewed David Rowe, the developer of the Open Source Line Echo Canceller (Oslec). Bob asks David about the difficulties involved in building something as difficult as an echo canceller and the tricks David used for testing and debugging, while allowing him to tell us a bit about what it's like to be a successful open source developer.

Chris McEniry takes us on a journey through using gRPC and protocol buffers in Golang. Borrowing the certificate generation portion of his Winter 2017 column, Mac explains how to use protobuf, a non-language-specific library, with Golang, as well as how to use gRPC, Golang's version of Remote Procedure Calls. A lot of work for one column.

EDITORIAL

Musings

David Blank-Edelman wants to prove to you that Perl is alive and well. He takes us to a site that keeps track of the hottest, or currently most interesting, Perl modules. He also demonstrates a few of these modules.

Dave Josephsen returns from KubeCon and CloudNativeCon fired up about the Open Tracing API. Dave explains just why tracing requests traveling between microservices is a crucial part of monitoring these systems, tells us how Open Tracing works, and suggests several frameworks for getting started.

Dan Geer and Dan Conway examine the security risks involved with crypto-currencies. At the time their column was written, Bitcoin had exceeded $18,000/BTC, 100,000 times its value just five years ago. Geer and Conway discuss the failings not just of Bitcoin but of other crypto-currencies. The amount of "value" that's already been lost or stolen is enough to give any sane person pause.

Robert Ferrell muses about the past, and the dark future, of the Internet. A much more serious column than his usual, but totally fitting the times.

Mark Lamourine has reviewed two books. He has high praise for *Fluent Python*, by Luciano Ramalho, a book that will sit beside his copy of Dave Beazley's *Python Essential Reference*. Mark also reviewed *Once Upon an Algorithm* by Martin Erwig.

You might find yourself wondering whether disk vendors really have taken control over block placement on modern drives. I heard Dave Anderson of Seagate mention this during FAST '07, questioned him about it, and wrote about this in a Musings column later in 2007. I've since been asked to prove this a number of times, and the best I've been able to come up with involves the documentation for a Seagate Enterprise SAS drive in 2004 [6]. I'm sure there are better examples, and even a standards doc that explains this change. If you know about this, please let me know, because people still find this hard to believe.

In the meantime, enjoy your hard drives, which are gaining not only in capacity over time, but also in intelligence.

References

[1] Rik Farrow, the long version, Morrow Micronix: https://www.rikfarrow.com/about/.

[2] Parallel ATA: https://en.wikipedia.org/wiki/Parallel_ATA.

[3] Small Computer System Interface: https://en.wikipedia.org/wiki/SCSI.

[4] T. Feldman and G. Gibson, "Shingled Magnetic Recording: Areal Density Increase Requires New Data Management," *;login:*, vol. 38, no. 3 (June 2013): https://goo.gl/wj5Doi.

[5] ACM CCS 2017 Agenda: https://ccs2017.sigsac.org/agenda.html.

[6] Seagate documentation: https://www.seagate.com/www-content/product-content/enterprise-hdd-fam/enterprise-capacity-3-5-hdd-10tb/_shared/docs/100791103d.pdf.

Letter to the Editor

Great interview of Peter in the Winter 2017 *;login:*. I had the pleasure of knowing and learning from Peter for many years.

Rik asked, "What happened with Multics?" It was a moderate commercial success, until its hardware became obsolete and was not replaced. The operating system design and features, and the people who helped build them, influenced many subsequent systems, including CHERI.

I can amplify Peter's remarks on Multics in a few areas.

Peter said, "The 645 was pretty much frozen early"—in fact, Multics had a major hardware re-design in 1973 (after Bell Labs left Multics development) when the GE-645 was replaced by the Honeywell 6180. The 6180 architecture extended the Multics hardware-software co-design, providing support for eight rings in hardware (instead of the 645's 64 rings simulated in software), as well as better security. A later I/O controller ran in paged mode and supported Multics device drivers that ran unprivileged in the user ring.

The transition from discrete transistor implementation to integrated circuits gave us 1 MIPS per 6180 CPU rather than the 645's 435 KIPS. The later DPS8/70 was rated at 1.7 MIPS.

Another minor clarification: Peter said, "The buffer overflow problem was solved by making everything outside of the active stack frame not executable, and enforcing that in hardware." Actually, there were several features preventing buffer overflows in Multics:

- The PL/I language has bounded strings and arrays, not just pointers.
- CPU string instructions enforced bounds at no runtime cost.
- "Execute" permission is limited to code segments.
- The stack grows from low addresses to high.
- ITS format prevents use of random data as pointers.
- The segment numbers are randomized.

See http://multicians.org/exec-env.html#buffer_overflow for more on this topic.

Another clarification: Peter said, "In the early 1970s there was even an effort that retrofitted multilevel security into Multics, which required a little jiggling of ring 0 and ring 1. I was a distant advisor to that (from SRI), although the heavy lifting was done by Jerry Saltzer, Mike Schroeder, and Rich Feiertag, with help from Roger Schell and Paul Karger." There were several projects to enhance Multics security so it could be sold to the US Air Force. The MLS controls were done by a project called Project Guardian, led by Earl Boebert. A more ambitious project to restructure the Multics kernel, led by Schell, Saltzer, Schroeder, and Feiertag, was canceled before its results were included in Multics (http://multicians.org/b2.html#guardian).

In the mid-'80s, the NCSC B2 security level was awarded to Multics, after a thorough examination of the OS architecture, implementation, and assurance. The evaluation process found a few implementation bugs; much of the effort in attaining the digraph was documenting the existing product.

There are over 2000 names on the list of Multicians. I am mildly uncomfortable at being the only person mentioned by Peter as "heavily involved" in Multics—we all were. I did my part, but there were many others who made contributions more important than mine, and some who worked on Multics longer. I look back on those times and those colleagues with affection and awe.

Jeffrey Yost's interview with Roger Schell, a key person in the design of security features and TCSEC ("the Orange Book"), is also fascinating: https://conservancy.umn.edu/handle/11299/133439.

Regards,
Tom Van Vleck
thvv@multicians.org

Underground Economics for Vulnerability Risk

LUCA ALLODI

Luca Allodi is an Assistant Professor at the Technical University of Eindhoven, the Netherlands. His research interests lie around the quantitative characterization of IT risk and attacker decisions. l.allodi@tue.nl

The estimation of vulnerability risk is at the core of any IT security management strategy. Among technical and infrastructural metrics of risk, attacker economics represent an emerging new aspect that several risk assessment methodologies propose to consider (e.g., based on game theory). Yet the factors over which attackers make their (economic) decisions remain unclear and, importantly, unquantified. To address this, I infiltrated a prominent Russian cybercrime market where the most prominent attack technology is traded. Supported by direct observations of market activity, I investigate in this work the economic factors that drive the adoption of new attacks *at scale* and their effect on risk of attack in the wild. As a market participant, I have access to the full spectrum of attack services offered to all members and, in particular, look at the market economics of vulnerability exploitation [1].

Software vulnerabilities are one of the main vectors of attack used to infect systems worldwide. As such, an effective management of vulnerability fixes is desirable on any system. Unfortunately, due to technical and budgeting restrictions, applying *all* fixes as soon as they are available is oftentimes not possible. For this reason, prioritizing patching work is a key aspect of any vulnerability management policy. The goal is clear: identify which vulnerabilities carry the highest risk and need immediate treatment.

Several methodologies to estimate this "potential risk" of vulnerability exploit exist, including technical measures of vulnerability severity (e.g., the Common Vulnerability Scoring System, CVSS), attack graphs, attack surfaces, and game-theoretic approaches that, for example, assign probabilities to specific attacker strategies in response to a certain set of defender decisions.

Importantly, and across all current approaches, the probability assigned to the materialization of an exploit mainly depends on vulnerability characteristics or specific "contextual" aspects such as network topology, deployed security controls, and vulnerability chaining. This, in turn, implicitly assumes that, all other factors being the same, attackers will be indifferent to which vulnerability to exploit.

An implication of this model is that all "high severity" vulnerabilities on a certain system or software will be equally likely to be exploited. Oftentimes, due to the high prevalence of severe vulnerabilities, exploit estimations will not be dramatically different across systems and vulnerabilities. This ultimately leads to inefficient vulnerability patching strategies [4], as most vulnerabilities are "indistinguishable" in terms of posed risk, and therefore all need immediate treatment.

All Vulnerabilities Are Not Equally Important

On the other hand, recent research developments reveal that the vast majority of attacks seem to be driven by a handful of vulnerabilities only. In [2], across most software types, the top 10% of vulnerabilities are reported to carry 90% of attacks across 1M Internet users

worldwide, approximating a power law distribution. Other research has shown that this huge skew in attack distribution is present also for zero-day vulnerabilities. In this analysis [6], of 20 zero-day vulnerabilities, two were reportedly responsible for millions of attacks worldwide, one for twenty thousand, and the remaining 18 for a few dozen only. These results are confirmed in follow-up empirical studies estimating that approximately 15% of disclosed vulnerabilities are exploited in the wild, and that this fraction is decreasing for recent vulnerabilities [10]. Similarly, recent work showed that the *refresh time* of exploits is very slow, with exploits being actively deployed in the wild up to two or three years before being substituted at scale by a different exploit [5].

These observations are in sharp contrast with the current narrative in the information security community, where every new severe vulnerability loosely resembles *Doomsday*. Industry studies recently started to acknowledge this effect as well: for example, in the last few editions of Verizon's Data Breach Investigations Report. Overall, empirical data clearly shows that a handful of vulnerabilities carry disproportionately more risk (by several orders of magnitude) than most vulnerabilities. It seems therefore that factors other than the characteristics of the vulnerability should be considered to explain this phenomenon.

Vulnerability Risk and Attacker Types

It is important to clarify the nature of the data leading to the observations above and its relation to different attacker types. In general, field data concerns attacks of an "untargeted" nature, where attackers in possession of a "fixed" set of exploits deliver attacks in the wild against the population of Internet users as a whole. These attacks are the most common and involve high attack automation, exploitation as-a-service [8], and delivery infrastructures based on spam or redirection of Internet traffic.

Attacks of a more "targeted" nature are radically different from the previous scenario: in such cases attackers adapt their exploit portfolio to the desired target system (as opposed to relying on a fixed set of exploits). Targeted attacks affect a very limited set of Internet systems and entail high levels of variability as attackers are (un)bounded by resource constraints, technical capabilities, and access rights to the network. Hence, in the case of targeted attacks, assigning probabilities to compute risk levels may not be a meaningful approach [7] as the notion itself of *probabilistic* risk does not apply anymore. In this article, I specifically refer to *risk of untargeted attacks at scale*.

A Dive into Exploit Economics

This distinction between "untargeted" and "targeted" attacks has become more and more relevant with the establishment of an underground economy driving the commodification of attacks at scale [8]. By outsourcing the complexity of attack engineering to the technically proficient sections of the underground, the technical difficulty of engineering and deploying an attack significantly decreased for those who participate in this economy. The acquisition of "off the shelf" attack tools represents a "multiplier factor" whereby a single attack technology (e.g., malware or vulnerability exploit) is shared among a multitude of attackers.

For example, exploit kits are known to be responsible for a significant share of the overall attack scenario by providing a ready-to-use, easy-to-configure attack framework that covers all steps of the attack process, from selection and redirection of vulnerable traffic, to vulnerability exploitation and malware delivery. Hence, buyers of these attack technologies may, potentially, jointly deliver a large fraction of attacks in the wild by sharing the same attack vectors and infrastructure.

I propose that the adoption of attack techniques traded in the cybercrime markets may explain the disproportionate concentration of attacks over a small set of vulnerabilities discussed above. Hence, under this hypothesis, it becomes central to understand the relation between deployment of an attack at scale and attackers' economic activities [1]. For example, pricier exploits may be adopted less widely by attackers, and vulnerabilities that are seldom substituted in the markets may remain exploited at scale for longer periods of time.

Market Identification and Infiltration

One of the difficulties associated with studying the underground economy is to identify active, well-functioning underground markets where prominent attack tools are traded. The underground economy is indeed fragmented in a multitude of markets, both in the so-called "deep web" as "onion services" and in the "open Internet." Whereas finding these markets is not a challenge per se, finding *credible* markets is: one should expect most markets to be places where gullible "wanna-be" criminals get scammed and no real technological innovation happens; Herley and Florencio provide an excellent coverage of the foundational economic reasons why this is the case [9].

Following Herley and Florencio's guidelines, and jointly with Professor Fabio Massacci at the University of Trento (Italy) and Professor Julian Williams at the Durham Business School (UK), I started evaluating different underground markets in the English and Russian hacking communities in 2011. One (Russian) community, above all, emerged as a prominent market where we find convincing evidence of severe trade regulation enforcement, credible trade activities, and the most prominent attack tools reported by the security industry, including exploit kits such as *RIG* and *Blackhole*, malware platforms, malware packers, and so on. We refer to this market under the fictitious name of RuMarket. All other markets in our analysis have been discarded for not meeting at least one of these criteria; [3] reports an example comparison.

We first gained access to RuMarket in 2011 and carried out "under-the-radar" observations of the activity therein, without performing any interaction with the market members. At the time, access to the market was only as difficult as registering to the corresponding forum platform under a fictitious identity.

This changed rather abruptly in 2013 when a prominent member of the market was arrested by the Russian authorities. The market reacted by ejecting all non-active participants and by significantly increasing the entry barrier to the market. Uncontrolled access to the market was replaced by a more strict process supervised by the market administration whereby access was granted only if either:

1. A trusted member of the market vouched for the entry request, effectively implementing a *pull-in* mechanism.
2. The request for market entry was backed up by evidence that the requestor was a reputable member of the Russian hacking community.

As we had no contacts inside the market to regain access, we chose to follow (2). This required extensive research to identify communities affiliated with RuMarket with more loose access barriers and build our identity from there. This, in turn, called for some proficiency in Russian in the discussion boards but did not involve the execution or support of criminal activities.

We gained new access to RuMarket in 2014 after more than six months of activity in the affiliated communities. We have been observing the market ever since. In this article, we look at the economics of vulnerability exploit trading [1].

Market Activity and Exploit Packages

In RuMarket, vulnerability exploits are traded in *packages*, or bundles. These can be classified using three categories: EKIT (exploit kit), Malware, and Standalone exploits. Figure 1 reports on the introduction of new exploit packages per year. Standalone packages are clearly on the rise, whereas Malware and EKIT packages are introduced or updated at a steady rate each year. This difference can be explained by looking at the different business models behind the bundles: Malware and EKIT are typically service-oriented products that require a prolonged contractual agreement between the buyer and the seller and are very popular in the market (in particular, the average EKIT advertisement receives approximately 10 times more replies from the community than the average Standalone or Malware package). As such, vendors tend to regularly update their products (e.g., with new or more reliable exploits) as opposed to substituting the whole package with a new one. This creates a perhaps slightly counterintuitive effect in which only a few players sell EKITs (despite these being very attractive products in the market): the prolonged contractual form requires high levels of trust between market participants, a condition only well-established vendors

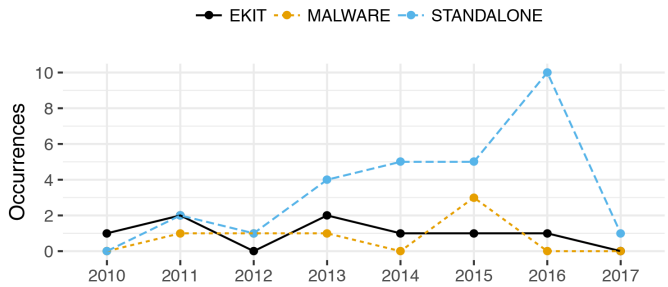


Figure 1: Release of exploit packages by type per year

Type	No.	Min	Mean	Median	Max
EKIT	6	150	693.89	400	2000
Malware	6	420	1735	1250	4000
Standalone	26	100	2972.69	3000	8000
ALL	38	100	2417.46	1500	8000

Table 1: Package prices (in USD)

can meet, and hence the low rate of new kits each year. As most malware in RuMarket is not advertised to exploit any specific vulnerability, Malware products have low introduction rates in Figure 1.

Table 1 reports descriptive statistics of package prices. Prices for rented EKITs are averaged over a period of three weeks, following the duration of typical malware delivery campaigns. We can observe that EKIT products are by far the cheapest, with a mean price of 700 USD, whereas Malware and Standalone products are significantly more expensive at 2000–3000 USD on average. This difference is stressed at the right-end tail of the distributions, where Standalone packages peak at 8000 USD, Malware at 4000, whereas EKITs stop at 2000 USD. Prices do not show a significant correlation with the number of embedded exploits, suggesting that other aspects, such as the business model behind the trade, or the age of the embedded exploits, may play a factor. An evaluation of the trend in pricing for each package type reveals that prices are clearly inflating for Standalone and Malware products, whereas EKIT prices are decreasing over time. This reflects the "consumer" nature of EKIT products, which are becoming more and more available to a larger pool of buyers, whereas the prices for Standalone exploits reflect a "niche" part of the market and are inflating.

Vulnerability Exploits

With the aim of evaluating the effect of exploit economics on vulnerability risk, it is useful to look at a breakdown of exploits bundled in a package, as opposed to the bundle "as a whole." Figure 2 reports the rate of introduction of single exploits in the market aggregated by vendor of the vulnerable software. Unsur-

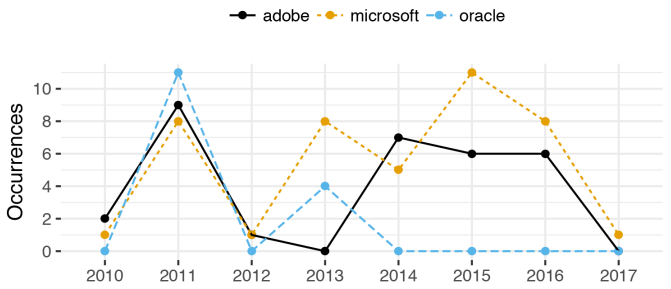


Figure 2: Occurrences of exploit publication by year

Type	No.	Min	Mean	Median	Max
EKIT	25	1	372.48	294	1745
Malware	1	185	185	185	185
Standalone	29	1	147.34	75	934
ALL	55	1	250.36	93	1745

Table 2: Exploit age (days) at time of first appearance in RuMarket

prisingly, in RuMarket we find exploits for Microsoft, Oracle, and Adobe software, which can be expected to cover the vast majority of user systems in the wild. The first observation we make is that the first "burst" of exploits appears in 2011, which corresponds to the appearance of "exploitation-as-a-service" as a new attack model [8]. After 2011 the market experienced a relative drop in number of introduced exploits to then stabilize around an average level of 6–8 new exploits per software vendor per year. This trend loosely resembles the *Gartner Hype Cycle* describing the introduction of new technologies in a market: a first inflation in the expectations associated with that technology causes a burst of interest in the market, followed by a "disillusionment" phase and, finally, by what Gartner calls the *plateau of productivity*, where the technology reaches maturity and its true value.

Table 2 reports the age, in days, of the exploits first introduced in RuMarket relative to the date of their publication in the National Vulnerability Database (NVD). As all collected exploits are associated with a Common Vulnerabilities and Exposures (CVE) identifier, no vulnerability is published in RuMarket before its publication on NVD. Interestingly, reporting the vulnerability's CVE is also the de facto standard for exploit advertisement in RuMarket (see sec. 3.2 in [1] for a discussion of why this is the case). All Malware samples included an exploit for the same vulnerability, which allows the malware to escalate to a higher privilege group on the victim system.

EKIT and Standalone exploits account for most of the variability in the market. EKIT exploits are by far the older ones at time of publication; 50% of Standalone exploits arrive two

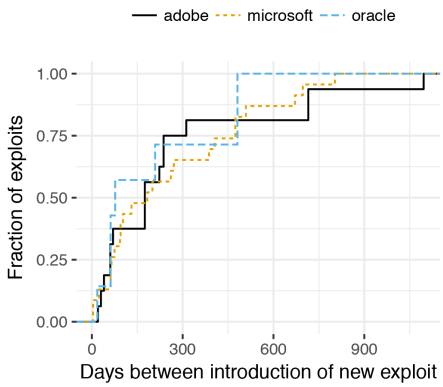


Figure 3: Distribution of days between exploit introduction

months after disclosure, whereas the faster 50% of EKIT's make it to the market after more than nine months. This has a clear correspondence with the package prices reported in Table 2, in which Standalone exploits are the most expensive in the market and EKITs the cheapest. A more formal analysis indeed reveals a strong correlation between exploit price and exploit age, with significantly different rates associated to different vulnerable software platforms: for example, exploits for Microsoft and Adobe products appear to better retain their value as they age than exploits for Oracle products.

Another important aspect in the overall threat scenario is *how often* exploits for a software platform are updated in the market. Figure 3 reports the cumulative distribution function of the time that passes between new exploits for a specific software, grouped by vendor. Irrespective of software vendor, we observe that in the median case, exploits are substituted six months after first introduction. The slowest update rate of exploits is around two years. This figure is well in line with previous findings on measurements of exploit appearance in the wild [5, 10] and underlines the importance of considering attacker activity in estimating vulnerability risk.

Economic Factors of Vulnerability Exploitation

To evaluate the relation between market activity and risk of exploit, we rely on data from Symantec on the presence of an exploit at scale [4]. Note that whereas an exploit for a vulnerability might well exist even if not reported by Symantec, it is unlikely for an exploit that delivers on the order of hundreds of thousands or millions of attacks to remain unnoticed and unreported.

We consider exploit package price, market activity around an exploit (measured in terms of the number of RuMarket responses to the ad reporting the exploit), and vulnerability severity as factors that may affect the probability of finding an exploit at scale. A formal analysis reveals that all effects significantly affect the change in odds of exploitation in the wild

for the respective vulnerability. Whereas a full description of the technical analysis is given in [1], as a rule-of-thumb the following emerges:

1. As market activity around an exploit doubles, so do the odds of finding an exploit at scale for the corresponding vulnerability.
2. As price of exploit acquisition doubles, the odds of exploit at scale halve.
3. Once we consider exploits traded in the markets, vulnerability severity becomes a significant predictor for exploitation in the wild.

Whereas the figures above are only indicative, a fully quantitative model can be obtained by plugging the coefficients reported in [1] in any vulnerability risk model. Importantly, a first approximation can be obtained without any direct insight from the markets. For example, exploit price can be estimated by considering the software vendor and the age of the vulnerability at the time of the estimate; this price can then be used, in conjunction with the vulnerability's severity, to estimate the change in the risk profile of the vulnerability if introduced in the market and how this evolves as time passes.

Although these conclusions are necessarily limited to RuMarket, and therefore the specific quantitative estimations may vary by considering other markets (e.g., trading vulnerabilities affecting different software vendors, or aiming at a larger English-speaking community), the qualitative conclusion remains: attacker economics are clearly correlated with risk of attack. Further research is needed in this direction: what is the attacker's process in deciding on which exploit to introduce and when? What determines whether an exploit can be expected to be traded in a market, as opposed to being used privately, or not being used at all? I believe that a characterization of these aspects can fundamentally change our perspective on cyber-risk and can provide an important building block for the division of workable and effective security practices.

References

[1] L. Allodi, "Economic Factors of Vulnerability Trade and Exploitation," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, pp. 1483–1499: https://acmccs.github.io/papers/p1483-allodiA.pdf.

[2] L. Allodi, "The Heavy Tails of Vulnerability Exploitation," in *Proceedings of 7th International Symposium on Engineering Secure Software and Systems (ESSoS '15)*, pp. 133–148.

[3] L. Allodi, M. Corradin, and F. Massacci, "Then and Now: On the Maturity of the Cybercrime Markets," *IEEE Transactions on Emerging Topics in Computing*, vol. 4, no. 1 (Jan.–March 2016): http://ieeexplore.ieee.org/document/7044581/.

[4] L. Allodi and F. Massacci, "Comparing Vulnerability Severity and Exploits Using Case-Control Studies," *ACM Transaction on Information and System Security (TISSEC)*, vol. 17, no. 1 (August 2014): http://disi.unitn.it/~allodi/allodi-tissec-14.pdf.

[5] L. Allodi, F. Massacci, and J. Williams, "The Work-Averse Cyber Attacker Model: Evidence from Two Million Attack Signatures," in *Workshop on the Economics of Information Security (WEIS '17)*: https://ssrn.com/abstract=2862299.2017.

[6] L. Bilge and T. Dumitras, "Before We Knew It: An Empirical Study of Zero-Day Attacks in the Real World," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (*CCS '12*), pp. 833–844: http://users.ece.cmu.edu/~tdumitra/public_documents/bilge12_zero_day.pdf.

[7] B. C. Ezell, S. P. Bennett, D. von Winterfeldt, J. Sokolowski, and A. J. Collins, "Probabilistic Risk Analysis and Terrorism Risk," *Risk Analysis*, vol. 30, no. 4 (2010), pp. 575–589: https://www.dhs.gov/xlibrary/assets/rma-risk-assessment-technical-publication.pdf.

[8] C. Grier, L. Ballard, J. Caballero, N. Chachra, C. J. Dietrich, K. Levchenko, P. Mavrommatis, D. McCoy, A. Nappa, A. Pitsillidis, N. Provos, M. Z. Rafique, M. A. Rajab, C. Rossow, K. Thomas, V. Paxson, S. Savage, G. M. Voelker, "Manufacturing Compromise: The Emergence of Exploit-as-a-Service," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (*CCS '12*), pp. 821–832: http://cseweb.ucsd.edu/~voelker/pubs/eaas-ccs12.pdf.

[9] C. Herley and D. Florencio, "Nobody Sells Gold for the Price of Silver: Dishonesty, Uncertainty and the Underground Economy" in *Economics of Information Security and Privacy* (Springer, 2010).

[10] K. Nayak, D. Marino, P. Efstathopoulos, T. Dumitras, "Some Vulnerabilities Are Different Than Others," in *Proceedings of the 17th International Symposium on Research into Attacks, Intrusions, and Defenses (RAID '14)*, pp. 426–446: http://www.umiacs.umd.edu/~tdumitra/papers/RAID-2014.pdf.

27TH USENIX SECURITY SYMPOSIUM

BALTIMORE, MD, USA

Co-located Workshops

WOOT '18

12th USENIX Workshop on Offensive Technologies
August 13–14, 2018
Submissions due May 30, 2018
www.usenix.org/woot18

WOOT '18 aims to present a broad picture of offense and its contributions, bringing together researchers and practitioners in all areas of computer security. Offensive security has changed from a hobby to an industry. No longer an exercise for isolated enthusiasts, offensive security is today a large-scale operation managed by organized, capitalized actors. Meanwhile, the landscape has shifted: software used by millions is built by start-ups less than a year old, delivered on mobile phones and surveilled by national signals intelligence agencies. In the field's infancy, offensive security research was conducted separately by industry, independent hackers, or in academia. Collaboration between these groups could be difficult. Since 2007, the USENIX Workshop on Offensive Technologies (WOOT) has aimed to bring those communities together.

ASE '18

2018 USENIX Workshop on Advances in Security Education
August 13, 2018
Submissions due May 8, 2018
www.usenix.org/ase18

ASE '18 is intended to be a venue for cutting-edge research, best practices, and experimental curricula in computer security education. The workshop welcomes a broad range of paper and demo submissions on the subject of computer security education in any setting (K–12, undergraduate, graduate, non-traditional students, professional development, and the general public) with a diversity of goals, including developing or maturing specific knowledge, skills and abilities (KSAs), or improving awareness of issues in the cyber domain (e.g., cyber literacy, online citizenship). ASE is intended to be a venue for educators, designers, and evaluators to collaborate, share knowledge, improve existing practices, critically review state-of-the-art, and validate or refute widely held beliefs.

CSET '18

11th USENIX Workshop on Cyber Security Experimentation and Test
August 13, 2018
Submissions due May 10, 2018
www.usenix.org/cset18

CSET '18 invites submissions on cyber security evaluation, experimentation, measurement, metrics, data, simulations, and testbeds. The science of cyber security poses significant challenges. For example, experiments must recreate relevant, realistic features in order to be meaningful, yet identifying those features and modeling them is very difficult. Repeatability and measurement accuracy are essential in any scientific experiment, yet hard to achieve in practice. Few security-relevant datasets are publicly available for research use and little is understood about what "good datasets" look like. Finally, cyber security experiments and performance evaluations carry significant risks if not properly contained and controlled, yet often require some degree of interaction with the larger world in order to be useful.

FOCI '18

8th USENIX Workshop on Free and Open Communications on the Internet
August 14, 2018
Submissions due May 24, 2018
www.usenix.org/foci18

FOCI '18 will bring together researchers and practitioners from technology, law, and policy who are working on means to study, detect, or circumvent practices that inhibit free and open communications on the Internet. Internet communications drive political and social change around the world. Governments and other actors seek to control, monitor, and block Internet communications for a variety of reasons, ranging from extending copyright law to suppressing free speech and assembly. Methods for controlling what content people post and view online are also multifarious. Whether it's traffic throttling by ISPs or man-in-the-middle attacks by countries seeking to identify those who are organizing protests, threats to free and open communications on the Internet raise a wide range of research and interdisciplinary challenges.

HotSec '18

2018 USENIX Summit on Hot Topics in Security
August 14, 2018
Lightning talk submissions due June 11, 2018
www.usenix.org/hotsec18

HotSec '18 aims to bring together researchers across computer security disciplines to discuss the state of the art, with emphasis on future directions and emerging areas. HotSec is not your traditional security workshop! The day will consist of sessions of lightning talks on emerging work and positions in security, followed by discussion among attendees. Lightning talks are 5 MINUTES in duration—time limit strictly enforced with a gong! The format provides a way for lots of individuals to share ideas with others in a quick and more informal way, which will inspire breakout discussion for the remainder of the day.

Registration will open in May 2018.

A Large-Scale Empirical Study of Security Patches

FRANK LI AND VERN PAXSON

Frank Li is a PhD student at the University of California, Berkeley. His research mainly focuses on improving the remediation process for security issues such as vulnerabilities and misconfigurations. More broadly, he is interested in large-scale network measurements and empirical studies in a computer security context. frankli@cs.berkeley.edu

Vern Paxson is a Professor of Electrical Engineering and Computer Sciences at the University of California, Berkeley, and leads the Networking and Security Group at the International Computer Science Institute in Berkeley. His research focuses heavily on measurement-based analysis of network activity and Internet attacks. He works extensively on high performance network monitoring, detection algorithms, cybercrime, and countering censorship and abusive surveillance. vern@berkeley.edu

Miscreants seeking to exploit computer systems incessantly discover and weaponize new security vulnerabilities. As a result, system administrators and end users must constantly run on the "patch treadmill," where they apply security patch after security patch to fix newly discovered software vulnerabilities, relying on many of the same processes practiced for decades to update their software against the latest threats. Given the vital role that security patches play in our management of vulnerabilities, it behooves us to better understand the patch development process and characteristics of the resulting fixes.

Prior studies [2, 4, 5, 7, 8] investigated aspects of the vulnerability and patching life cycles but typically at a restricted scale in terms of software diversity, focusing on only a few projects or even just one. While these studies provide insights into the patch development process, there remains a question of how generally their findings apply, and how the nature of security patches may differ from that of other types of bug fixes. Security patches are of particular importance given their critical role in securing software and the time sensitivity of their development.

In this work, we conduct a large-scale empirical study of security patches, investigating 4,000+ bug fixes for 3,000+ vulnerabilities that affected a diverse set of 682 open-source software projects. We build our analysis on a data set that merges vulnerability entries from the National Vulnerability Database [6], information scraped from relevant external references, affected software repositories, and their associated security fixes. Tying together these disparate data sources allows us to perform a deep analysis of the patch development life cycle, including investigation of the code base life span of vulnerabilities, the timeliness of security fixes, and the degree to which developers can produce safe and reliable security patches. We also extensively characterize the security fixes themselves in comparison to general bug patches, exploring the complexity of different types of patches and their impact on code bases.

Data Collection Methodology

To explore vulnerabilities and their fixes, we must collect security patches and information pertaining to them and the remedied security issues. Given this goal, we restricted our investigation to open-source software for which we could access source code repositories and associated metadata. Our data collection centered around the National Vulnerability Database (NVD) [6], a database provided by the US National Institute of Standards and Technology (NIST) with information pertaining to publicly disclosed software vulnerabilities. These vulnerabilities are identified by CVE (Common Vulnerabilities and Exposures) IDs.

We mined the NVD and crawled external references to extract relevant information, including the affected software repositories, associated security patches, public disclosure dates, and vulnerability classifications. Figure 1 depicts an overview of this process. In the remainder

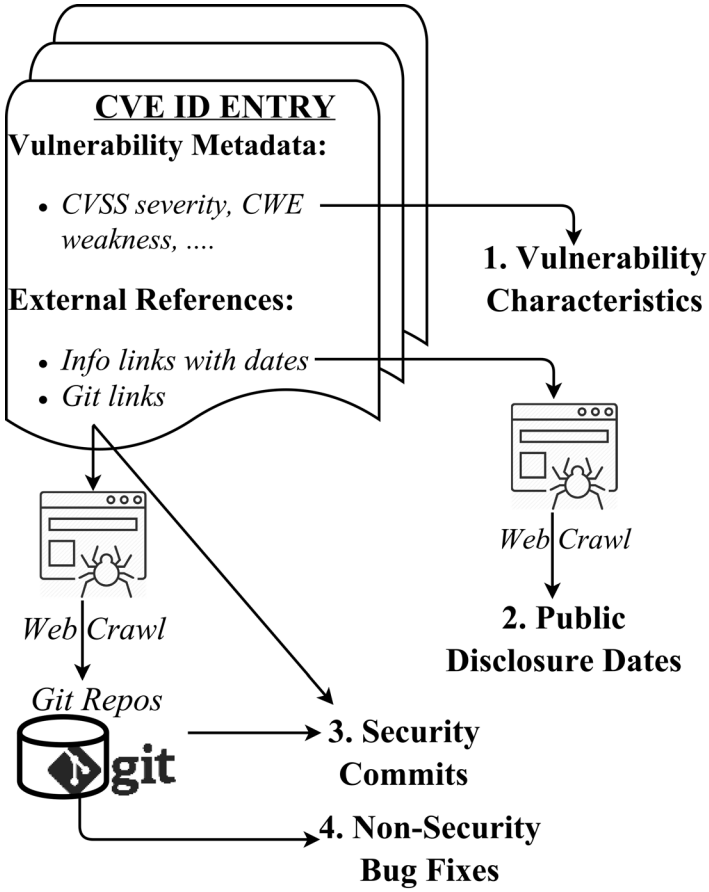


Figure 1: An overview of our data collection methodology. (1) We extracted vulnerability characteristics from CVE entries in the NVD with external references to Git commit links. (2) We crawled other references and extracted page publication dates to estimate public disclosure dates. (3) We crawled the Git commit links to identify and clone the corresponding Git source code repositories, and collected security fixes using the commit hashes in the links. (4) We also used the Git repositories to select general bug fixes.

of this section, we briefly describe these various data sources and our collection methodology (see [3] for details).

Note that throughout our methodology, we frequently manually inspected random samples of populations to confirm that the population distributions accorded with our assumptions or expectations.

Finding Public Vulnerabilities with the NVD

The NVD contains entries for all publicly released vulnerabilities assigned a CVE identifier, and rich annotations about the vulnerabilities. In particular, it summarizes the vulnerability, links to relevant external references (such as security advisories and reports), specifies the affected software, identifies the class of security weakness under the Common Weakness Enumeration (CWE) classifications, and evaluates the vulnerability severity using the Common Vulnerability Scoring System (CVSS).

We focused on the NVD as it is public, expansive, manually curated, and detailed. For this study, we analyzed a snapshot of the NVD taken on December 25, 2016. Its 80,741 CVE vulnerabilities served as our starting point for further data collection.

Identifying Software Repositories and Security Patches

Many open-source version-controlled software repositories provide web interfaces to navigate project development (such as git.kernel.org). We frequently observed URLs to these web interfaces among the external references for CVE entries, linking to particular repository commits that addressed the security vulnerability. We focused on popular Git web interfaces as they were the most commonly occurring (and Git overall is popular). Crawling these links afforded us the ability to collect security patches and access the source code repositories.

In total, we retrieved 4,080 commits across 682 unique Git repositories, tied to 3,094 CVEs. Note that these repositories are distinct, as we de-duplicated mirrored versions. By manually investigating 100 randomly sampled commits, we found that all commits reflect fixes for the corresponding vulnerabilities, indicating the vast majority, if not all, of our commits are security patches. This data set corresponds to a variety of vulnerability types and severities, affecting an expansive range of products (from OS distributions to applications to libraries), detailed in [3].

Identifying General Bug Fixes

We can gain insights into any especially distinct characteristics of security patches by comparing them to bug fixes *in general*. However, to do so at scale we must automatically identify bug fixes. We tackled this problem using a logistic regression that models the character n-grams in Git commit messages to identify likely bug fix commits. We discuss the details of our classifier training and evaluation in [3].

With our classifier, we collected a data set of bug fixes by randomly selecting per repository up to 10 commits classified as bug fixes. This provided us with a large set of over 6,000 bug fixes (similar to our number of security fixes) balanced across repositories.

Processing Commits

In a patch, it can be useful to consider only changes to functional source code, rather than documentation files or source code comments. For each commit that we collected (both security and general bug fixes), we processed the commit data to produce an alternative "cleaned" version that filtered non-source code files and removed comments.

Estimating Vulnerability Public Disclosure Dates

Determining the public disclosure date of a vulnerability is vital to understanding the timeline of its life cycle. The CVE publication date indicates when the CVE entry was published, not necessarily when the vulnerability was publicly disclosed. To estimate the public disclosure date, we analyzed the external references associated with CVEs. These web pages frequently contain publication dates for information pertaining to vulnerabilities. Example pages include security advisories, public mailing list archives, other vulnerability database entries, and bug reports. We chose the earliest date among the extracted dates and the CVE publication date as our estimate.

Analysis Results

Our collected data set provides us with a unique perspective on the development life cycle of security fixes, as well as on the characteristics of the security patches themselves in comparison to general bug fixes. In this section, we discuss our more salient analyses and findings (see [3] for additional analyses).

We first consider the patch development process by connecting the vulnerability information available in the NVD with the historical logs available in Git repositories. We follow that by analyzing our collection of security and general bug fixes to help illuminate their differences, considering facets such as the complexity of fixes and the locality of changes. In general, to assess whether differences observed have statistical significance, we use permutation tests with a significance threshold of $\alpha = 0.05$ (discussed in detail in [3]).

Vulnerability Life Spans in Code Bases

Upon a vulnerability's discovery, we might naturally ask how long it plagued a code base before a developer rectified the issue, a duration we call the *code base life span*. Automatically and reliably determining this life span is difficult, requiring semantic understanding of the source code and the vulnerability. However, we can approximate a *lower bound* on age by determining when the source code affected by a security fix was previously last modified. We note that this heuristic does assume that security fixes modify the same lines that contained insecure code. We assessed that this is a robust approximation through manual inspection of a random sample of security patches.

We analyzed the cleaned versions of security commit data to focus on source code changes. For all lines of code deleted or modified by a security commit, we retrieved the last time each line was previously updated. We conservatively designate the most recent change date across all of the lines as the estimated vulnerability birth. The duration between this date and the patch commit date provides a lower bound on the vulnerability's code base life span. We observe that vulnerabilities exist in code bases for extensive durations, with a median life span of 438 days (14.4 months). Furthermore, a third of all CVEs had life spans beyond three years. The longest surviving vulnerability was a *21-year-old* information disclosure vulnerability in Kerberos.

Security Fix Timeliness

The timeliness of a security fix relative to the vulnerability's public disclosure affects the remediation process and the potential impact of the security issue. On the one hand, developers who learn of insecurities in their code bases through unanticipated public announcements have to quickly react before attackers leverage the information for exploitation. On the other hand, developers who learn of a security bug through private channels can address the issue before public disclosure, but may not release the available patch for some time due to a project's release cycle, expanding the vulnerability's window of exposure.

We explore this facet of remediation by comparing the patch commit date for CVEs in our data set with public disclosure dates (estimated as described in "Data Collection Methodology," above).

How frequently are vulnerabilities unpatched when disclosed? We observe that 21% of all vulnerabilities were not fixed at the time of public disclosure. We cannot determine whether these vulnerabilities were privately reported to project developers but with no prior action taken, or disclosed without any prior notice. However, a quarter (26%) of these unpatched security issues remained unaddressed 30 days after disclosure, leaving a window wide open for attacker exploitation.

For the remaining 79% of all CVEs, project developers committed the security fixes by public disclosure time. This suggests that the majority of vulnerabilities were either internally discovered or disclosed to project developers using private channels, the expected best practice.

Are vulnerability patches publicly visible long before disclosure? The degree to which security commits precede disclosures varies widely. This behavior highlights the security impact of an interesting aspect of the open-source ecosystem. Given the public nature of open-source projects and their development, an attacker targeting a specific software project can feasibly track security patches and the vulnerabilities they address.

While the vulnerability is remedied in the project repository, it is unlikely to be widely fixed in the wild before public disclosure and update distribution. We note that over 50% of CVEs were patched more than two weeks before public disclosure, giving attackers ample time to develop and deploy exploits.

Patch Reliability

The patch that a developer creates to address a vulnerability may unfortunately disrupt existing code functionality or introduce new errors. Beyond the direct problems that arise from such patches, end-user trust in generally applying patches (or in the software itself) can erode. To assess how successful developers are at producing reliable and safe security fixes, we identified instances of multiple commits for the same CVE, and classified the causes.

To locate CVEs associated with multiple commits where a subsequent commit may fix a previous one, we found CVEs listed in the NVD with multiple commits. Additionally, we attempted to identify further commits potentially associated with a CVE using repository Git logs, looking for commit messages that explicitly reference the original patch's commit hash or the CVE ID. Note that with this approach, we could only identify multiple patches when commit messages contained this explicit linkage, so our analysis provides a lower bound.

Filtering out duplicate commits (e.g., merges, rebases, and cherry-picks) as well as CVEs where all commits were within a 24-hour time window (thus even if there was a problem, it was quickly resolved), we found 440 CVEs with multiple commits.

CVE Commits Label	Num. CVEs	Median Num. Follow-on Commits	Median Fix Interarrival Time (days)
Incomplete	26 (52%)	1.0	181.5
Regressive	17 (34%)	1.0	33.0
Benign	14 (28%)	1.5	118.5

Table 1: Summary of our manual investigation into 50 randomly sampled CVEs with multiple commits. Note that a CVE may have commits in multiple categories.

We randomly sampled 50 of the remaining 440 CVEs and manually investigated whether the fixes were problematic. Table 1 summarizes our results. We identified 26 (52%) as having incomplete fixes, requiring a later patch to complete the job. We labeled 17 (34%) as regressive, as they introduced new errors that required a later commit to address. Other follow-on commits were benign, such as commits for documentation, testing, or refactoring. Note that some CVEs had multiple commits in multiple categories, resulting in the sum of CVEs in each category exceeding 100%. Problematic initial patches were followed by a median of one additional commit, with a median of 181.5 days and 33 days between commits for incomplete and regressive patches, respectively.

This random sample is representative of the 440 CVEs with multiple commits accounting for 14.2% of all CVEs. Extrapolating from the sample to all CVEs, we estimate that about 7% of *all* security fixes may be incomplete, and about 5% regressive. These findings indicate that broken patches occur with unfortunate frequency, and applying security patches comes with non-negligible risks. In addition, these numbers have a skew towards underestimation: we may not have identified all existing problematic patches, and recent patches in our data set might not have had enough time yet to manifest as ultimately requiring multiple commits.

Patch Complexity

How complex are security patches compared to bug fixes in general? Given the number and diversity of software projects we consider, we chose lines of code (LOC) as a simple-albeit-rudimentary metric.

Are security patches smaller than general bug fixes? Under the LOC metric, security commits overall are statistically significantly smaller than bug patches in general ($p \approx 0$). The median security commit diff involved 7 LOC compared to 16 LOC for general bug fixes. Approximately 20% of general bug patches had diffs with over 100 lines changed, while this occurred in only 6% of security commits.

Do security patches make fewer "logical" changes than general bug fixes? As an alternative to our raw LOC metric, we can group consecutive lines changed by a commit as a single "logical" change. Under this definition, we consider several lines updated as a single logical update, and a chunk of deleted code counts as a single logical delete. Across all logical actions, we observe that security commits involve significantly fewer changes (all $p < 0.01$). Nearly 78% of security commits did not delete any code, compared to 66% of general bug-fix commits. Between 30% and 40% of all commits for both security and general bug-fix commits also did not add any new code portions, indicating the majority of logical changes were updates to existing code.

Do security patches change code base sizes less than general bug fixes? Another metric for a patch's complexity is its impact on the code base size. The net number of lines changed by a commit reflects the growth or decline in the associated code base's size. We observe that significantly more general bug patches result in a net reduction in project LOC, compared to security fixes: 18% of general bug fixes reduced code base sizes compared to 9% of security patches. For all commits, approximately a quarter resulted in no net change in project LOC, which commonly occurs when lines are only updated. Overall, projects are more likely to grow in size with commits, since the majority of all commits added to the code base. However, security commits tend to contribute less growth compared to general bug fixes, an observation that accords with our earlier results.

Patch Locality

Finally, we can quantify the impact of a patch by its *locality*. We consider two metrics: the number of files affected and the number of functions affected.

Do security patches affect fewer source code files than general bug fixes? We observe that security patches modify fewer files compared to bug fixes in general, a statistically significant observation ($p \approx 0$). In aggregate, 70% of security patches affected one file, while 55% of general bug patches were equivalently localized. Fixes typically updated, rather than created or deleted, files (mirroring code changes, which were typically updates). Only 4% of security fixes created new files vs. 13% of general bug fixes, and only 0.5% of security patches deleted files vs. 4% of general bug fixes.

Do security patches affect fewer functions than general bug fixes? We find that 5% of general bug fixes affected only global code outside of function boundaries, compared to 1% of security patches. Overall, we observe a similar trend as with the number of affected files. Security patches are significantly ($p \approx 0$) more localized across functions: 59% of security changes resided in a single function compared to 42% of other bug fixes.

Aspect of Security Patches	Summary of Results
Vulnerability Life Spans	Vulnerabilities often lived for years, with a third for more than three years.
Security Fix Timeliness	A fifth of vulnerabilities were not fixed at public disclosure time. When fixed before disclosure, the patches were visible in repositories weeks to months in advance.
Patch Reliability	We conservatively estimate that about 7% of security patches were incomplete and 5% regressive.
Patch Complexity	Security patches were significantly smaller than bug fixes in general.
Patch Locality	Security patches were more localized in their changes than general bug fixes.

Table 2: Summary of main analysis results.

Moving Forward

In this study, we have conducted a large-scale empirical analysis of security patches across over 650 projects. Here we discuss the main takeaways, highlighting the primary results developed (summarized in Table 2) and their implications for the security community moving forward.

Need for more extensive or effective code testing and auditing processes for open-source projects. Our results show that vulnerabilities live for years and their patches are sometimes problematic. These findings indicate that the software development and testing process, at least for open-source projects, is not adequate at quickly detecting and properly addressing security issues. A natural avenue for future work is to develop more effective testing processes, particularly considering usability, as developers are unlikely to leverage methods that prove difficult to deploy or challenging to interpret. In addition, software developers can already make strides in improving their testing processes by using existing tools such as sanitizers or fuzzers more extensively.

The transparency of open-source projects makes them ripe for such testing not only by the developers, but by external researchers and auditors as well. Community-driven efforts, such as those supported by the Core Infrastructure Initiative [1], have already demonstrated that they can significantly improve the security of open-source software. Further support of such efforts, and more engagement between various project contributors and external researchers, can help better secure the open-source ecosystem.

Need for refined bug reporting and public disclosure processes for open-source projects. Our analysis of the timeliness of security fixes revealed that they are poorly timed with vulnerability public disclosures. Over 20% of CVEs were unpatched when they were first announced, perhaps sometimes to the surprise of project developers.

In the opposite direction, we discovered that when security issues are reported or discovered privately and fixed, the remedy is not immediately distributed and divulged, likely due to software release cycles. Over a third of fixed vulnerabilities were not publicly disclosed for more than a month. While operating in silence may help limit to a small degree the dissemination of information about the vulnerability, it also forestalls informing affected parties and spurring them to remediate. Given the transparency of open-source projects, attackers may be able to leverage this behavior by tracking the security commits of target software projects. From the public visibility into these commits, attackers can identify and weaponize the underlying vulnerabilities. The issue of vulnerability disclosure and embargoing of information is a complex debate, but the visibility of the patch itself should be part of that discussion.

Opportunities for leveraging characteristics of security patches. Our comparison of security patches with general bug fixes revealed that security fixes have a smaller impact on code bases, across various metrics. They involve fewer lines of code, fewer logical changes, and are more localized in their changes. This has implications along various patch analysis dimensions, such as patch safety analysis. Tying back to broken patches, the lower complexity of security patches can perhaps be leveraged for safety analysis customized for evaluating just security fixes. Also, as these remedies involve fewer changes, automatic patching systems may operate more successfully if targeting security bugs. Zhong and Su [8] observed that general patches are frequently too complex or too delocalized to be amenable to automatic generation. However, security patches may be small and localized enough. From a usability angle, we may additionally be able to better inform end users of the potential impact of a security update, given its smaller and more localized changes. The need for more exploration into the verification and automated generation of security patches is quite salient as our ability to respond to security vulnerabilities still heavily depends on patching, while the attack landscape has grown ever more dangerous.

Acknowledgements
This work was supported in part by the National Science Foundation awards CNS-1237265 and CNS-1518921.

References

[1] Core Infrastructure Initiative: https://www.coreinfrastructure.org.

[2] Z. Huang, M. D'Angelo, D. Miyani, and D. Lie, "Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response," in *Proceedings of the 37th I-EEE Symposium on Security and Privacy (S&P '16)*: https://www.eecg.toronto.edu/~lie/papers/zhuang_talos_oakland2016.pdf.

[3] F. Li and V. Paxson, "A Large-Scale Empirical Study of Security Patches," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS '17)*: https://www.icir.org/vern/papers/patch-study.ccs17.pdf.

[4] A. Ozment and S. E. Schechter, "Milk or Wine: Does Software Security Improve with Age?" in *Proceedings of the 15th USENIX Security Symposium (Security '06)*: https://www.usenix.org/legacy/event/sec06/tech/full_papers/ozment/ozment.pdf.

[5] J. Park, M. Kim, B. Ray, and D. Bae, "An Empirical Study of Supplementary Bug Fixes," in *Proceedings of the 9th Working Conference on Mining Software Repositories (MSR '12)*: https://web.cs.ucla.edu/~miryung/Publications/msr2012-supplementarypatch.pdf.

[6] US National Institute of Standards and Technology, National Vulnerability Database: https://nvd.nist.gov/home.cfm.

[7] S. Zaman, B. Adams, and A. E. Hassan, "Security Versus Performance Bugs: A Case Study on Firefox," in *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11)*: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.740.4377&rep=rep1&type=pdf.

[8] H. Zhong and Z. Su, "An Empirical Study on Real Bug Fixes," in *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*: https://web.cs.ucdavis.edu/~su/publications/icse15-bugstudy.pdf.

Secure Client and Server Geolocation over the Internet

ABDELRAHMAN ABDOU, PAUL C. VAN OORSCHOT

AbdelRahman Abdou is a Postdoctoral Researcher in the Department of Computer Science at ETH Zurich. He received his PhD (2015) in systems and computer engineering from Carleton University. His research interests include location-aware security, SDN security, and using Internet measurements to solve problems related to Internet security. abdoua@inf.ethz.ch

Paul C. van Oorschot is a Professor of Computer Science at Carleton University, and the Canada Research Chair in Authentication and Computer Security. He was the program chair of USENIX Security 2008, NDSS 2001–2002, and NSPW 2014–2015; a co-author of the *Handbook of Applied Cryptography*; and a past Associate Editor of *IEEE TDSC*, *IEEE TIFS*, and *ACM TISSEC*. He is an ACM Fellow and Fellow of the Royal Society of Canada. His research interests include authentication and Internet security. paulv@scs.carleton.ca

We provide a summary of recent efforts towards achieving Internet geolocation securely, that is, without allowing the entity being geolocated to cheat about its own geographic location. Cheating motivations arise from many factors, including impersonation (if locations are used to reinforce authentication) and gaining location-dependent benefits. In particular, we provide a technical overview of Client Presence Verification (CPV) and Server Location Verification (SLV)—two recently proposed techniques designed to verify the geographic locations of clients and servers in real time over the Internet. Each technique addresses a wide range of adversarial tactics to manipulate geolocation, including the use of IP-hiding technologies like VPNs and anonymizers, as we now explain.

Internet geolocation is the process of determining the geographic location of an Internet-connected device. Secure geolocating of a web client (a client visiting a website) is useful for location-aware authentication, location-aware access control, location-based online voting, location-based social networking, and fraud reduction. From the client's perspective, geolocating the remote server can provide higher assurance to the server's identity, and justify conducting certain sensitive transactions—for example, those requiring certain privacy measures or requiring data sovereignty [1]. Independent of server and client geolocation, geolocating network intermediate systems (e.g., routers) can also be useful for monitoring [2] and network mapping [3].

Both CPV and SLV are based on network measurements, where delays are measured from trusted network nodes dubbed *verifiers* and are analyzed in real time to verify physical presence inside a prescribed geographic region. We explain the threat model of both techniques, how they militate against known adversarial tactics, how they adapt to various network dynamics, and what distinguishes them from other geolocation approaches.

Geolocation Background

Many academic geolocation methods have been proposed, but there has been very limited deployment in practice. As of this writing, most of the geolocation conducted in practice relies on the clients' IP address or GPS coordinates of hand-held devices, as explained below.

Geolocation in Practice

There are several methods for device geolocation over the Internet. If the device belongs to a user that is acting as a web client (i.e., visiting a website), the Geolocation API is a W3C standard that enables browsers to obtain location information of the device they are running on and communicate it to a webserver. Servers request location coordinates using JavaScript as follows:

Figure 1 [a-c]: Snapshots of the Flagfox browser extension

```
if(navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(success, error,
geoOptions);
  } else {
    console.log("Geolocation is not supported on this
browser.");
}
```

The geolocation methods a browser uses are left to the browser vendor's discretion. Most major browsers rely on the following in varying orders (that is, when one fails, the next is tried): GPS, WiFi Positioning System (WPS), IP address-based location lookups, or cell-tower triangulation of mobile devices. The location of an IP address can be obtained from publicly available routing information or public registries, such as whois. Many IP location service providers (commercial and free) maintain lookup tables to instantly map IP addresses to locations. Such static *tabulation* methods may take long times to reflect changes or IP address reassignments, which occur quite often for client geolocation to be up-to-date (studies were conducted to confirm this [4]). IP address-based geolocation can, however, be reliable for benign server geolocation. *Flagfox* is an example Firefox extension that visually indicates a flag of the country corresponding to the IP address resolution of the URL (Figure 1).

From a security perspective, none of the above techniques is resilient to adversarial manipulation. When the geolocation API is in use, the server normally makes no effort in geolocating the client device; it rather trusts the browser-communicated coordinates, which can easily be forged on the fly before being sent to the server. Firefox extensions that enable forgery include *Fake Location* (Figure 2) and *Location Guard*; both enable a user to specify where in the world they would like to appear to be. If the server relies on tabulation methods to geolocate the client (instead of asking the browser for its coordinates), the common practice of clients hiding their own IP addresses behind proxies and anonymizers comes into play.

Geolocation in the Literature

A wide set of techniques can be used, mostly for a server to geolocate clients [5]. These enable a server to infer a client's geographic location from hints obtained from browser-generated HTTP headers such as preferred language or time zone. Locations can also be obtained through crowd-sourcing by interpolating a device's location from its proximity to nearby devices, like phones or WiFi access points (APs), with known GPS locations.

Figure 2: Snapshots of the Fake Location extension—an example browser extension allowing users to fake their locations

Another class of Internet geolocation approaches is based on network measurements. Similar to GPS triangulations that are based on the delays between the receiver and satellites, measurement-based techniques also aim to locate devices (clients or server) by estimating their distance from landmarks in the network with known locations. These landmarks measure network delays from themselves to the device, typically identified by its IP address, and map these delays to geographic distances. The accuracy of such mapping, however, is not anywhere near that of mapping satellite delays to distances, and is thus the primary source of inaccuracies in such techniques. Still, measurement-based geolocation is generally considered more accurate than methods like tabulation-based geolocation.

From the security point of view, although most of the above methods are positioned as resilient to evasion, examination has shown otherwise. Delay-increasing attacks can allow an adversary to distort its perceived location [6]. Delay-decreasing was also studied, for example, by manipulating ICMP "ping" and "traceroute" as they fail to preserve the integrity of timing measurements.

Combining both attacks, an adversary can forge the calculated location to an accuracy of a few tens of kilometers relative to a target desired location [7].

Client Presence Verification—CPV

CPV [8] is a measurement-based technique designed to verify the geographic locations of web users (clients) over the Internet. The client is assumed to be motivated to misrepresent its location to gain location-dependent benefits. CPV's design takes into consideration various adversarial location-forging tactics, including delay manipulations and IP-hiding technologies like VPNs and anonymizers. CPV does not rely fundamentally on the clients' IP addresses, nor does it determine geographic locations. Rather, it *verifies* an asserted (unverified) location, typically made by a client. The client's location could be asserted using the client's GPS coordinates, the client's IP address, or even explicitly asking the user to fill-in their street address in an online form during login.

To verify location assertions, CPV relies on an infrastructure of geographically scattered nodes, dubbed *verifiers*. The technique works as follows. When a client visits a website and asserts the geographic location from which he/she is currently browsing, three verifiers surrounding the asserted location are selected. The verifiers measure (in real time) network delays between themselves and the client's browser, and analyze these delays to corroborate that the client is present somewhere inside the triangle determined by their (the verifiers') geographic locations. Because the verifiers cannot pinpoint where exactly the client is within the triangle, the size of the triangle is the verification granularity.

Secure One-Way Delay Estimation

The verifiers do not measure round-trip times (RTT) between themselves and the client. Rather, they estimate the smaller of the forward and reverse one-way delays (OWDs) between each of them and the client. The larger OWD is discarded because propagation delays between two network nodes are bounded by the physical distance between them, so a smaller OWD measurement is a better representation to the geographic distance between both nodes than the larger—the larger must have been affected by other factors such as network congestion or circuitous routing.

To measure the OWD between a verifier and the client, CPV does not rely on standard OWD-estimation protocols like OWAMP (RFC 4656), as those require honest client cooperation: for example, client clock synchronization and honest reporting of delays. As such, CPV relies on the *minimum-pairs* (MP) protocol [9]. MP requires the three verifiers, A, B, and C, to first synchronize their clocks and pre-share cryptographic keys to ensure operational integrity.

Through JavaScript, the client's browser is first directed to establish a WebSocket (RFC 6455) connection to the three verifiers, which are chosen based on the client's asserted location. Verifier A begins by sending a cryptographically protected timestamp (in millisecond precision) to the client, which the browser forwards to the other two verifiers. On receiving this, verifier B calculates the propagation time from A → client → B, and likewise when the timestamp is received by C. Verifiers B and C then follow suit, taking turns in sending timestamps. When all three verifiers are done exchanging timestamp messages, they will have six delay values as follows:

- A→ client → B
- A→ client → C
- B→ client → A
- B→ client → C
- C→ client → A
- C→ client → B

Between each pair of verifiers, e.g., between {A → client → B} and {B → client → A}, the verifiers exclude the larger OWD and solve a system of three equations simultaneously for an estimate to the smaller OWD between the client and each verifier. That is, if the smaller of the forward and reverse OWD between the client and A, B, and C, respectively, is *a*, *b*, *c*, then (note: = sign here is used to indicate mathematical equality rather than an assignment operator):

- $a + b = min(AtB, BtA)$
- $a + c = min(AtC, CtA)$
- $b + c = min(BtC, CtB)$

where *AtB* is the delay A → client → B, and so on. Analysis of MP's accuracy showed that the protocol is likely to provide more accurate estimates to the smaller OWD than simply using half the RTT [9].

Corroborating Presence Inside the Triangle

In order to avoid potential inaccuracies from delay-to-distance mapping, the calculated OWDs are not mapped to distances. Rather, they are compared to the smaller OWDs between the verifiers themselves, which are measured and updated periodically in a background process, independent of whether or not a client is currently being verified. Assuming x = min(AB, BA) is the smaller of the forward and reverse OWDs between verifiers A and B *directly* (not to be confused with min(AtB, BtA) from the previous section), and likewise y = min(BC, CB) and z = min(AC, CA), then the client's asserted location is accepted as inside the triangle if:

$$area(\Delta xab)+area(\Delta ybc)+area(\Delta zca) \leq area(\Delta xyz) + \epsilon$$

such that area (Δxab) is the area of that triangle calculated from its side lengths x, a, and b. The value of ϵ is used to account for

the two extra access network traversals occurring at the client when the timestamps propagate from a verifier to the client to another verifier.

Iterative Delay Measurement

To account for abrupt delay spikes or network irregularities, the above process of OWD calculations and comparison with those between the verifiers is iteratively repeated n times. If the condition is met for the majority of the conducted iterations, the location assertion is accepted.

CPV Calibration

There are several parameters that tune CPV's reaction to events. The most important three are ϵ, n, which is the number of delay measurement iterations, and τ, which is the fraction of those iterations that must *pass* if the condition is met for the client's asserted location to be accepted. This calibration should take place before the location verification process begins. To do that, the three verifiers may use network nodes that they know as a ground truth to be inside the triangle. From the network delays of these nodes, the verifiers compute values for the above-mentioned three parameters and then run CPV to verify a client's location.

Hindering Illicit Traffic Relaying

In an attempt to defeat geolocation, a middlebox (like a proxy server or a VPN gateway) that is physically inside the triangle can be specifically customized to filter out the verifiers' timestamps from the client's traffic and forward them to the verifiers on behalf of the client. This threat against CPV is exacerbated by the presence of numerous cheap public VPN providers whose primary service is to enable subscribers to evade geolocation technologies.

Techniques like CPV can mitigate this by adapting known proof-of-work techniques [10]. The verifiers generate a cryptographic client puzzle with each timestamp message, which the client's browser must solve before forwarding the message (puzzle solution and timestamp) to the other two verifiers. The puzzles must be easy to solve so that they do not (1) overwhelm the client with high processing costs and (2) overshadow the network propagation delays. In the case of a middlebox connected to many simultaneously cheating clients, the middlebox will choose to either solve these puzzles on behalf of the clients or forward them to the clients. In the latter case, the network delay between the middlebox and the client will get added to the time the verifiers observe for location verification, which results in CPV correctly detecting the client's absence from the respective triangle. It is thus in the middlebox's interest to choose the former case—solving the puzzles on behalf of the clients. However, this means that as more clients are connected, the middlebox will have to solve more puzzles. When these puzzles begin to accumulate, they will increase queueing delays, which contribute to the delays observed by the verifiers, eventually causing CPV to reject the location assertions of all middlebox-connected clients.

In this model, there are two main parameters contributing to the puzzle queueing rate at the middlebox: the puzzle difficulty and the middlebox's computational resources. Queueing analysis [10] shows that the puzzle difficulty has a higher impact on the rate of puzzle queueing than the middlebox's computational power. This analysis suggests that this puzzle mechanism will effectively hinder illicit middlebox relaying.

Evaluation Results

CPV was evaluated using PlanetLab—a distributed testbed for Internet measurement research and network experiments—using 80 PlanetLab nodes in North America. Three of the nodes were selected to act as verifiers, and the remaining 77 acted as clients. Some of the 77 nodes were inside the triangle and others were outside. All 77 nodes carried out the protocol with the verifiers simultaneously to get their locations verified. Knowing the ground-truth of which nodes were inside and which were outside (the geographic locations of PlanetLab nodes are publicly disclosed on PlanetLab's website), we could count the number of false rejects, nodes inside the triangle identified by CPV as outside, and false accepts. The process is repeated after choosing a different triangle, a different set of three nodes to act as verifiers, again counting false rejects and false accepts. In total, 34 triangles where chosen. Triangles were chosen to be nearly equilateral (physically), with inside angles ranging from 50–70 degrees (0.87–1.22 radians). The smallest triangle had an area equivalent to a circle of radius 100 km, and the largest of 400 km.

When the inside nodes were not too close to the triangle's sides, that is, away from the closest side by at least 10% of its length, CPV resulted in a total of 1.0% false accepts and 2.0% false rejects [8]. These results were obtained when $n = 600$ CPV iterations were performed with each client. The results were not much different when only 100 iterations were performed, where the false accept rate increased only to 1.1% and the false reject rate remained unchanged. However, when only 10 iterations were performed, false accepts and false rejects were at 2.1% and 4.1%, respectively.

Testing was later repeated to assess the effect of WiFi access networks on CPV's efficacy [11]. WiFi access networks often have higher delays and delay jitters. A different evaluation technique was used, as the PlanetLab infrastructure used above involved nodes connected using wired access networks. To model WiFi clients, 802.11 delay models from the literature were used to generate the last-mile delays, which were added to the delay traces collected from PlanetLab. Since higher network delays for nodes inside the triangles may result in higher false rejects, the generated 802.11 delays were only added to the delays

of inside nodes to create the most stressful testing situation. 802.11 networks employ slotted retransmissions. The delays were generated such that each slot was 20 μsec, the propagation delay from the device to the wireless gateway was 1 μsec, and the four other wireless devices were continuously competing for the wireless media along with each wireless CPV client. With these parameters, CPV's false accepts were at 2% and false rejects at 4%. Although CPV's efficacy was affected by the WiFi access network, increasing the number of iterations can improve the results (see [11]).

Live Demo

A live demo of CPV is currently running on http://cpv.ccsl.carleton.ca. This link hits a webserver in Ottawa, Canada, which enables clients to verify whether they are present inside a US-based triangle determined by verifiers in San Francisco, Las Vegas, and San Diego. The verifiers are provided by hosting services DigitalOcean, ServerPoint, and M5 Hosting. Each VM has a 500 MB RAM and runs Ubuntu 16.04. NTP is used to synchronize their clocks. Additionally, each verifier issues an NTP query every 30 minutes using the "ntpq" utility to calculate the clock offset with the other two verifiers, which is added to the calculated OWDs between the verifiers for more accurate OWD estimates. Each verifier issues a timestamp to the other two verifiers every six seconds for direct OWD measurements between the verifiers.

A Java implementation of a CPV verifier runs on top of a lightweight custom-written WebSocket server, which is also implemented in Java. When a location verification request is initiated, the verifiers first check that it was issued from the authentic server (the one based in Ottawa in that demo implementation), because this server digitally signs connection IDs when they are issued. Additionally, each exchanged timestamp message between the verifiers through the client is corroborated using an MD5-based HMAC (a stronger HMAC is recommended to be used in practice). For the currently running demo, eight delay-measuring iterations are performed, once every 300 ms. When all iterations are performed, the verifiers send the measured delays back to the Ottawa server, which processes the result and returns it to the browser as a jQuery response.

No client puzzles are implemented yet in this demo as of this writing, nor is any automatic calibration of CPV's parameters. Instead, the main server has manually set parameters of $\epsilon = 10$ ms and $\tau = 0.7$, which are static and used across all clients.

Server Location Verification (SLV)

Analogous to CPV but on the server side, SLV [12] works by finding evidence of a server's physical presence inside a geographic region by measuring the server's network delays. A browser typically communicates with an SLV Manager, which orchestrates a network of server location verifiers. The challenges faced in doing so are quite different from verifying clients: (1) clients do not normally have the ability to write and run code on the server, whereas that was easily achievable by the server on the client, typically using JavaScript; (2) the common physical distribution of web content using content distribution networks (CDNs) and replication technologies begs the questions: *Of the multiple physical servers that may serve client content, which such servers should be selected to geographically locate (verify) in order to provide a useful server-authentication service? How should that machine be identified?*

The answers to these questions depend on the threat model and the application for which geolocation is to be used. Since the goal of SLV is to reinforce server authentication, the implementation of SLV takes the view that the first machine that terminates the client's TCP (and TLS) handshake is the most critical one. The protection provided from verifying that first machine would be comparable to that provided by TLS in the cases where the browser fetches content from multiple machines, some of which are not TLS-protected: for example, a page with mixed content.

For deciding on the mechanism used to identify machines, it is important to dissect man-in-the-middle (MITM) and server impersonation attacks. In MITM attacks, an adversary hijacks network traffic intended for the authentic server and relays it to the authentic server with or without modification. Hijacking could occur on several layers of the network stack as follows. (Note that using uncompromised TLS protects against the following hijacking cases; the value of using server location to reinforce server authentication is more profound for non-TLS-enabled websites or to catch attacks against the TLS system.)

- **Case 1: Attacker's machine has a different IP address than the authentic server**. In upper layers, phishing and pharming attacks are prominent traffic hijacking examples; the outbound traffic from the client has a different IP address from that of the authentic server. If the browser submits the domain name of the visited website to the SLV Manager, the Manager may resolve it to a different IP address from that seen by the browser (which could also occur benignly in the cases of CDNs). Verifying the geographic location of that IP address then becomes useless to the browser because a MITM adversary would go undetected. It is thus important to have the browser resolve a domain and submit the IP address to the SLV Manager.
- **Case 2: Attacker's machine has same IP address as authentic server**. In a lower layer hijacking, such as MAC table poisoning, ARP spoofing, and BGP spoofing, outbound traffic from the client has the same destination IP address as that of the authentic server. Such tactics are based on routing manipulation, so that traffic intended to the authentic server's IP address reaches a different network location (versus geographic location), which corresponds to the attacker's machine.

In comparison to upper layers, lower layer hijacking attacks tend to be more scalable, affecting a larger proportion of clients. For MAC table poisoning and ARP spoofing, the closer the attacker's machine is to the authentic server's network, the more the affected clients. Likewise, BGP spoofing can cause traffic hijacking at a global scale [13]. This implies that as with higher layer traffic hijacking attacks (discussed above), identifying the server by its IP address will likely allow the SLV Manager to detect whether the browser-intended machine is at a different geographic location from that asserted through a static location mapping previously obtained for that IP address.

Revisiting the above questions, if SLV targets the IP address as resolved by the browser of the first machine that the client initially handshakes, regardless of whether the browser will be instructed to fetch other content from different places later in the session, it can detect most of the above server impersonation attacks.

Verification Mechanism

After obtaining an unverified server location assertion, three verifiers surrounding that location are selected. The verifiers measure network RTTs to the server over several layers, including an application using HTTP request-response times and transport using TCP handshake responses. By means of comparing these delays with the delays between the verifiers, each pair of verifiers then verify whether the server is physically present inside the circle whose diameter is the physical distance between the verifier pair, and whose center is the midpoint between them (Figure 3).

Evaluation Results

Pilot testing of ~200 experiments was conducted on SLV using PlanetLab, half of which were true location assertions made by servers and the other half were false assertions. As with CPV, the rates of false rejects and false accepts were the fundamental evaluation parameters. SLV resulted in 0% false accepts and 2.4% false rejects [12]. Although the false reject rate may seem high for some applications, it can be improved by proper selection of verifiers, those with sufficient network bandwidth and processing resources.

SLV Browser Extension

We have built a Firefox browser extension to reinforce TLS by integrating the webserver's verified physical location, as described above, into the server authentication process. The extension sends the IP address of the server to the SLV Manager and receives the location verification result. The extension uses FlagFox to obtain an unverified assertion for the server's location. It also displays a flag in the URL (Figure 1) and a green tick mark or a red cross indicating whether the location asserted by FlagFox is true (according to SLV's verification) or not. This process takes a few seconds to execute, during which a throbber is displayed by the flag instead. Note that such visual cues are only meant as visual feedback in prototypes and are not an indication that we would expect end-users to base decisions upon. See below for how policies could be implemented to automatically make decisions on behalf of users.

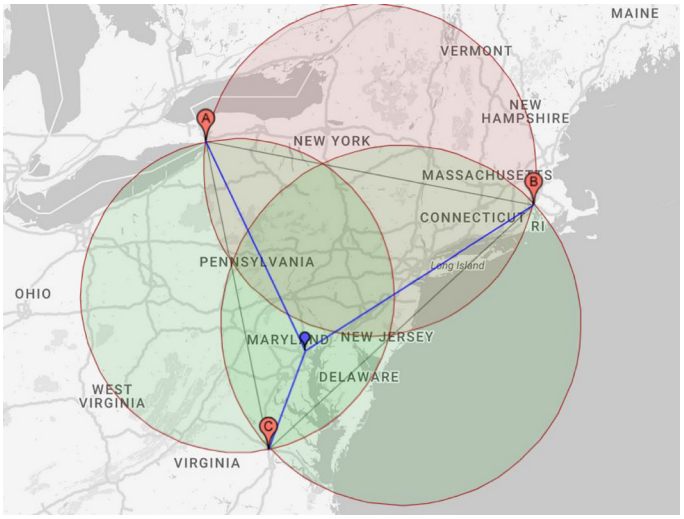


Figure 3: Server Location Verification (SLV) using network measurements from three verifiers (A, B, and C) to a server. Please view the online version of this article to see the figure in color. Map data: Google, INEGI.

Server Location Pinning in the Browser.

To avoid having the user interpret visual icons, the SLV extension is supported with a *location pinning feature*, whereby a browser saves the fact that a website identified by its URL was previously verified to host content from a particular geographic location, analogous to key pinning [14]. Although location verification is performed based on the IP address, the SLV Manager only receives an IP address from the browser, with location pinning in the browser based on the domain name. Upon receiving the verification result for a website, its location gets pinned only if the result is positive. This operation follows a trust on first use (TOFU) concept.

In general, for interpreting a received verification result, the SLV extension checks whether that location to some degree of geographic precision was pinned before for that website. The result of the operation falls into one of three categories: Critical, Suspicious, or Unsuspicious. Critical means the verification result for a previously pinned location was negative. A Suspicious outcome occurs when the location verification result is negative, but no location was previously pinned for that website . Finally, an Unsuspicious outcome is when location verification passes for a domain that was not previously pinned. Note that these are only meant to illustrate how a client might utilize SLV, but we expect different applications would make different choices.

Such outcomes could result in the browser automatically taking decisions through a policy-based engine. An example would be to instruct the browser to block/terminate the connection for all Critical outcomes of the user's personal banking website. Such terminology is subject to more research scrutiny and is not yet part of the above-described SLV extension.

Conclusion

This article provides a technical overview of recent advancements in the field of secure geolocation over the Internet. Two technologies, CPV and SLV, were explained to address client and server geolocation, respectively. Both rely on network timing measurements for secure location verification, taking into consideration safety measures to limit adversarial manipulations. Of the wide variety of applications that may benefit from secure location information of clients and servers, reinforcing authentication (location-aware authentication) for both ends remains an important example. Future research on CPV and SLV includes further enhancing their accuracy in terms of the false reject and accept rates and their efficiency for large-scale deployments in practice.

References

[1] Z. N. J. Peterson, M. Gondree, and R. Beverly, "A Position Paper on Data Sovereignty: The Importance of Geolocating Data in the Cloud," in *Proceedings of the 3rd USENIX Conference on Hot topics in Cloud Computing (HotCloud '11)*: https://www.usenix.org/legacy/event/hotcloud11/tech/final_files/Peterson.pdf.

[2] B. Huffaker, M. Fomenkov, and k. claffy, "DRoP: DNS-Based Router Positioning," *SIGCOMM Computer Communication Review*, vol. 44, no. 3 (2014), pp. 5–13: http://www.caida.org/publications/papers/2014/drop/drop.pdf.

[3] A. Csoma, A. Gulyás, and L. Toka, "On Measuring the Geographic Diversity of Internet Routes," *IEEE Communications Magazine*, vol. 55, no. 5 (2017), pp. 192–197: https://arxiv.org/pdf/1601.01116.pdf.

[4] I. Poese, S. Uhlig, M. A. Kaafar, B. Donnet, and B. Gueye, "IP Geolocation Databases: Unreliable?" *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 2 (2011), pp. 53–56: https://inl.info.ucl.ac.be/system/files/paper_2.pdf.

[5] J. A. Muir and P. C. van Oorschot, "Internet Geolocation: Evasion and Counterevasion," *ACM Computing Surveys*, vol. 42, no. 1 (2009): https://www.ccsl.carleton.ca/~jamuir/papers/TR-06-05.pdf.

[6] P. Gill, Y. Ganjali, B. Wong, and D. Lie, "Dude, Where's That IP?: Circumventing Measurement-Based IP Geolocation," in *Proceedings of the 19th USENIX Conference on Security (Security '10)*, pp. 241–256: https://people.cs.umass.edu/~phillipa/papers/UsenixSec2010.pdf.

[7] A. Abdou, A. Matrawy, and P. C. van Oorschot, "Accurate Manipulation of Delay-Based Internet Geolocation," in *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS '17)*, pp. 887–898: http://people.scs.carleton.ca/~paulv/papers/asiaccs-2017.pdf.

[8] A. Abdou, A. Matrawy, and P. C. van Oorschot, "CPV: Delay-Based Location Verification for the Internet," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 14, no. 2 (2017), pp. 130–144: https://sce.carleton.ca/~abdou/CPV_TDSC.pdf.

[9] A. Abdou, A. Matrawy, and P. C. van Oorschot, "Accurate One-Way Delay Estimation with Reduced Client-Trustworthiness," *IEEE Communications Letter*, vol. 19, no. 5 (2015): http://people.scs.carleton.ca/~paulv/papers/OWD.pdf.

[10] A. Abdou, A. Matrawy, and P. C. van Oorschot, "Taxing the Queue: Hindering Middleboxes from Unauthorized Large-Scale Traffic Relaying," *IEEE Communications Letter*, vol. 19, no. 1 (2015): http://people.scs.carleton.ca/~paulv/papers/taxing-the-queue.pdf.

[11] A. Abdou, A. Matrawy, and P. C. van Oorschot, "Location Verification of Wireless Internet Clients: Evaluation and Improvements," *IEEE Transactions on Emerging Topics in Computing (TETC)*, vol. 5, no. 4 (2017), pp. 563–575.

[12] A. Abdou and P. C. van Oorschot, "Server Location Verification (SLV) and Server Location Pinning: Augmenting TLS Authentication," *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 1, (2017), pp. 1–26.

[13] R. Hiran, N. Carlsson, and P. Gill, "Characterizing Large-Scale Routing Anomalies: A Case Study of the China Telecom Incident," in *International Conference on Passive and Active Network Measurement* (Springer, 2013), pp. 229–238: https://people.cs.umass.edu/~phillipa/papers/Hiran_Pam2013_full.pdf.

[14] M. Kranch and J. Bonneau, "Upgrading HTTPS in Mid-Air: An Empirical Study of Strict Transport Security and Key Pinning," Network and Distributed System Security Symposium, Internet Society, 2015: http://www.jbonneau.com/doc/KB15-NDSS-hsts_pinning_survey.pdf.

XDP-Programmable Data Path in the Linux Kernel

DIPTANU GON CHOUDHURY

Diptanu Gon Choudhury works on large-scale distributed systems. His interests lie in designing large-scale cluster schedulers, highly available control plane systems, and high-performance network services. He is the author of the upcoming O'Reilly book *Scaling Microservices*. diptanuc@gmail.com

Berkeley Packet Filter was introduced almost two decades ago and has been an important component in the networking subsystem of the kernel for assisting with packet filtering. Extended BPF can do much more than that and is gradually finding its way into more kernel subsystems as a generic event-processing infrastructure. In this article, I provide enough background to help you understand how eBPF works, then describe a simple and fast firewall using Express Data Path (XDP) and eBPF.

Berkeley Packet Filter (BPF) has been around for more than two decades, born out of the requirement for fast and flexible packet filtering machinery to replace the early '90s implementations, which were no longer suitable for emerging processors. BPF has since made its way into Linux and BSDs via libpcap, which is the foundation of tcpdump.

Instead of writing the packet filtering subsystem as a kernel module, which can be unsafe and fragile, McCanne and Jacobson designed an efficient yet minimal virtual machine in the kernel, which allows execution of bytecode in the data path of the networking stack.

The virtual machine was very simple in design, providing a minimalistic RISC-based instruction set, with two 32-bit registers, but it was very effective in allowing developers to express logic around packet filtering. BPF owes its relative longevity to two factors—flexibility and performance. The design goal was to design the subsystem in a protocol-agnostic manner and the instruction set to be able to handle unforeseen use cases.

A sample BPF program that filters every IP packet:

```
(000) ldh      [12]
(001) jeq      #0x800          jt 2      jf 3
(002) ret      #262144
(003) ret      #0
```

This program loads a half-word from offset 12, checks if the value is #0x800, and returns true if it matches and false if it doesn't.

The flexible instruction set allowed programmers to use BPF for all sorts of use cases such as implementing packet filtering logic for iptables, which performs very well under high load and allows for more complex filtering logic. Having a protocol-independent instruction set allowed developers to update these filters without writing kernel modules; having a virtual machine run the instructions provided a secure environment for execution of the filters. A significant milestone was reached in 2011 when a just in time (JIT) compiler was added to the kernel, which allowed translating BPF bytecode into the host system's assembly instruction set. However, it was limited to only x86_64 architecture because every instruction was mapped one on one to an x86 instruction or register.

Things took an interesting turn when the BPF subsystem was "extended" in the Linux operating system in 2013, and since then BPF is used in a lot more places, including tracing and security subsystems, besides networking.

Extended BPF

Linux 3.18 had the first implementation of extended BPF (eBPF), which made significant improvements from its precursor. While the original BPF virtual machine had two 32-bit registers, eBPF had 10 64-bit registers, added more instructions that were close to the hardware, and made it possible to call a subset of the kernel functions. All the BPF registers matched with the actual hardware registers, and BPF's calling conventions were similar to the Linux kernel's ABI in most architectures. One of the important outcomes of this was that it was now possible to use a compiler like LLVM to emit BPF bytecode from a subset of the C programming language. Another important addition was the BPF_CALL instruction, which allows BPF programs to call helper functions from the kernel allowing reuse of certain existing kernel infrastructure.

The important point to keep in mind is that eBPF today can be used as a general-purpose event-processing system by various subsystems in the kernel. These events can come from various different sources such as a kprobe tracepoint or an arrival of a packet in the receive queue of the network driver. Support for BPF has gradually been added to various strategic points in the kernel such that when code in those kernel subsystems execute, the BPF programs are triggered. The kernel subsystems that trigger a BPF program dictate the capability of a BPF program, and usually every BPF program type is connected to a kernel subsystem. For example, the traffic control subsystem supports the BPF_PROG_TYPE_SCHED_CLS and BPF_PROG_TYPE_SCHED_ACT program types that allow developers to write BPF to classify traffic and control behavior of the traffic classifier actions, respectively. Similarly, the `seccomp` subsystem can invoke a BPF program to determine whether a userspace process can make a particular syscall.

Writing the BPF bytecode for anything nontrivial can be challenging, but things have become a lot simpler since BPF has been added as a target in LLVM and users can now generate BPF in a subset of the C programming language.

In today's Linux kernel, the old BPF instruction set, commonly known as cBPF, is transparently translated to eBPF instructions. *I will use eBPF and BPF interchangeably from here on.*

BPF Maps

An introduction to BPF is incomplete without discussing BPF maps. BPF programs by themselves are stateless, and so maps allow programs to maintain state between invocations. For example, we could write a BPF program that prints a trace message whenever the `inet_listen` function is called in the kernel. However, if we wanted to expose that information as a counter to some monitoring tool, we would need the program to maintain state somewhere and increment a counter every time the method is called. This is where BPF maps come in. BPF maps are generic data structures implemented in the kernel where eBPF programs can store arbitrary data. These data structures, commonly referred to as maps, treat the data as opaque, and hence programs can store arbitrary bytes as key-value as appropriate. Maps can only be created or deleted from the userspace; BPF programs access the maps by using helper functions such as `bpf_map_lookup_elem`.

As of this writing, there are 11 different types of maps implemented in the kernel today, some of them generic and others used specifically with helper functions. The generic maps are:

```
BPF_MAP_TYPE_HASH
BPF_MAP_TYPE_ARRAY
BPF_MAP_TYPE_PERCPU_HASH
BPF_MAP_TYPE_PERCPU_ARRAY
BPF_MAP_TYPE_LRU_HASH
BPF_MAP_TYPE_LRU_PERCPU_HASH
BPF_MAP_TYPE_LPM_TRIE
```

Each of them is designed for a specific use case, so it's useful to understand the performance characteristics and their heuristics before starting to use them in BPF programs. For example, if we were designing a filter that increments a counter for every UDP packet that is being dropped, it would be best to use a per-CPU hash map so that the counters can be incremented without any synchronization to prevent multiple instances of the BPF program being triggered on different CPUs simultaneously. The non-generic maps are best described in the context of the documentation for the operations with which they can be used.

The BPF Syscall

The BPF syscall introduced in kernel 3.18 is the main workhorse for userspace programs to interact with the BPF infrastructure. The syscall multiplexes almost all the operations that userspace processes need to perform when handling BPF programs and maps. The syscall's usage includes, but is not limited to, loading BPF filters into the kernel, creating new maps, or retrieving data from existing ones.

```
int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

- `cmd`—It could be one of the many operations that the syscall can perform. There are 10 such commands in total, six of which are documented in the man page.
- `attr`—A union that provides context to the command. For example, when used with the BPF_PROG_LOAD command, it allows the bytecode to be passed to the kernel, and with the BPF_MAP_CREATE, it lets the user define the size of the key and values of the map.
- `size`—The size of the attr union.

The syscall returns 0 when it succeeds in most cases, except for BPF_PROG_LOAD and BPF_MAP_CREATE, which return the file descriptor of the BPF object created. For any failures, it returns -1 and sets the appropriate errno.

However, most userspace programs don't use the raw syscalls; the BPF Compiler Collection (BCC) provides the libbpf library, which has some wrapper functions that make working with BPF objects easier.

```
int bpf_prog_load(enum bpf_prog_type prog_type, const char
*name, const struct bpf_insn *insns, int prog_len, const char
*license, unsigned kern_version, int log_level, char *log_buf,
unsigned log_buf_size)
```

For example, the above wrapper function creates the attribute union for the `bpf()` syscall and wires in appropriate parameters. The kernel samples include good examples of usage of the libbpf library and other userspace helpers.

BPF Verifier

Safety is a very important concern for BPF programs, especially because they tend to run in the performance-critical sections of the kernel. The BPF infrastructure includes an in-kernel verifier that uses CFG (control flow graph) to determine that the BPF program terminates within the limit of maximum number of instructions. The verifier, for example, forbids loops and makes sure that maps are not destroyed until a program that uses it doesn't terminate. The verifier also statically ensures the safety of the calls to the helper functions by checking that the types of the data in the BPF VM register matches with the types of the helper function arguments.

In addition to ensuring type safety, the verifier also ensures safety of the program by prohibiting out of bounds jumps and out-of-range data access. The verifier also restricts what kernel functions and which data structures can be accessed based on the BPF program type.

BPF File System

BPF maps and filters are effectively kernel resources exposed to userspace via file descriptors backed by anonymous inodes; this comes with some benefits but also interesting challenges. Once a userspace program exits, the BPF program would get destroyed, and so would the maps related to that program. The lifetime of a BPF program and maps are tied to that of the userspace process that loaded the program, which prevents maps from persisting between filter invocations. As a result, maintaining state between program invocations becomes impossible. To overcome these limitations, the BPF infrastructure comes with a file system where BPF objects like maps and programs can be pinned to a path in the file system. This process is commonly known as Object Pinning, and two new commands, BPF_OBJ_PIN and BPF_OBJ_GET, facilitate pinning and retrieving an existing pinned object. The command simply needs file descriptors and the path to which the object is going to be pinned.

An interesting aspect of BPF objects being exposed as file system objects is that processes with higher privileges could create the objects and pin them to the file system and then drop their permission. For example, this allows lower-privileged userspace tools, like monitoring tools, to read telemetry data from maps.

BPF Tail Calls

BPF programs are limited to 4096 instructions, but it's possible to chain multiple programs together via *tail calls*. This technique allows a BPF program to call another BPF program when it finishes. Tail calls are implemented by long jumps inside the VM, which reuse the same stack frame. BPF tail calls are different from normal functions in the sense that once the new function is invoked when the current function ends, the previous program ends. Data could be shared between stages by using per-CPU maps as temporary buffers. Tail calls are used to modularize BPF programs. For example, a program could parse the headers of a network packet, and the following program could implement some other logic like tracing or running classifier actions based on the headers.

There are certain limitations to tail calls:

1. Only similar programs can be chained together.
2. The maximum number of tail calls allowed is 32.
3. Programs which are JITed can't be mixed with the ones that are not JITed.

As stated earlier, various kernel subsystems now have support for BPF. I will cover one such area that is part of the networking subsystem.

Express Data Path (XDP)

The networking subsystem of the kernel is one of the more performance-sensitive areas—there is always ongoing work to improve performance! Over the years userspace networking frameworks like DPDK have attracted users with the promise of faster packet processing by bypassing the kernel network stack. While it's lucrative for userspace programs to get access to network devices and improve on some data copies by bypassing the kernel, there are some problems with that approach as well. Most notably, in some cases packets have to be re-injected back to the kernel when they are destined for ssh or other system services. XDP provides an in-kernel mechanism for packet processing for certain use cases by providing access to the raw packets, so BPF filters can make decisions based on the headers or contents within the packets. XDP programs run in the network driver, which enables them to read an ethernet frame from the Rx ring of the NIC and take actions before any memory is allocated for the

packet in the kernel's socket buffers (skb). As a use case, XDP programs can drop packets in the event of denial-of-service attacks at the line rate without overwhelming the kernel's TCP stack or the userspace application. It is important to note that XDP is designed to cooperate with the existing networking subsystem of the kernel, and so developers can selectively use XDP to implement certain features that don't need to leave the kernel space.

XDP programs currently can make the following decisions:

1. XDP_DROP—Instructs the driver to simply drop the packet. It's essentially recycling a page in the Rx ring queue, since this happens at the earliest possible stage of the Rx flow.
2. XDP_TX—Retransmits a packet on the same NIC source. In most scenarios the eBPF program alters the headers or the contents of the packet before retransmitting. This allows for some very interesting use cases, such as load balancing where the load balancing decision is entirely done in the eBPF program. In this scenario, the networking stack or any userspace code doesn't need to participate in the decision making or packet retransmission flow, which allows for throughput close to line rate. One important point to keep in mind is that the XDP infrastructure doesn't have any sort of buffer, because the packets are processed at the driver layer, so when a packet is retransmitted and the TX device is slower, packets might simply get dropped.
3. XDP_PASS—The eBPF program has allowed the packet to move on to the networking stack of the kernel. It's also possible to rewrite the contents of the packets before the packet is passed on.
4. XDP_ABORT—This action is reserved for usage when the program encounters some form of an internal error; it essentially results in the packet getting dropped.

XDP depends on drivers to implement the Rx hook and plug into the eBPF infrastructure. Currently, there can be only one XDP program attached to a driver, but programs can call other programs using the tail calls infrastructure.

Case Study: XDP-Based Firewall

To demonstrate how XDP programs work, we can go through the design of a simple packet filtering service. Services like firewalls are usually divided into a distributed control plane and a data plane. The control plane provides APIs for operators to create filtering rules and introspects the filters to provide telemetry data. The data plane runs on every host in a cluster where packet filtering happens. XDP filters naturally constitute the data plane of such a system.

In general, software using BPF filters are divided into three parts:

1. BPF filter code and maps that are loaded into the kernel
2. Userspace program that loads the filter and provide APIs to update various maps
3. Optional processes like command line tools to access the maps

BPF Maps

The BPF maps form the most essential part of the firewall system. As stated above, they essentially allow the userspace processes to provide the rule set for performing packet filtering and the XDP program to emit telemetry data. We use the following maps in the data plane:

1. LPM trie map—The trie data structure allows doing prefix-based lookups efficiently, and BPF includes an implementation of LPM (longest prefix match) trie. We will use the LPM trie map to store the CIDR blocks of the source and the destination addresses which have to be either blacklisted or whitelisted.
2. Map array—For whitelisted or blacklisted destination ports.
3. Hash maps—Hold counters for packets dropped and passed based on the rule set.

XDP Filters

The BPF program that XDP invokes when a packet is received in the driver contains the logic for parsing incoming packets, reads the maps to look up the rules provided by the userspace process, and makes filtering decisions based on them. The BPF program also updates maps with telemetry data to provide observability into the actions taken.

There would be separate XDP filters for whitelisting and blacklisting flows, so we will have two different XDP filters:

1. The blacklisting filter would parse the ethernet frame and extract the source IP address and the destination port. If the source IP address has a match in the LPM trie, it would simply return the XDP_DROP action. From there on, it would look up the blacklist's array map and return the XDP_DROP action if there is a match. If none of the above checks has a positive outcome, the filter returns the XDP_PASS action, thereby passing on the packet to the kernel's networking stack.
2. The whitelisting filter behaves similarly except that it returns the XDP_PASS action and allows the packet to pass into the kernel only if the lookups within the LPM trie map and the array map have a successful match. In other cases it returns the XDP_DROP action, thereby dropping the packet.

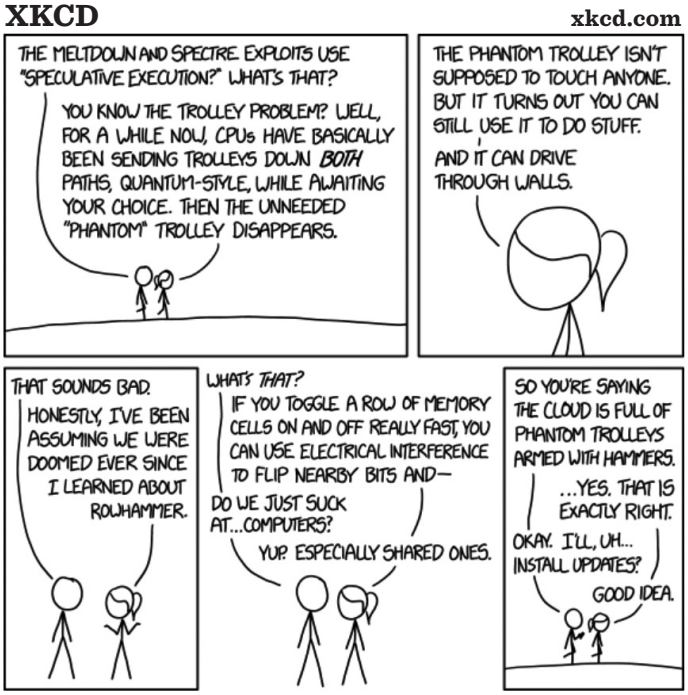
Userspace Program

The userspace program has all the necessary infrastructure to interact with the control plane service and also interacts with the BPF infrastructure. It retrieves the rules that need to be enforced, creates the necessary maps, and loads either of the whitelisting or the blacklisting filters based on the rules that need to be enforced. Any updates from the control plane would in turn update the maps containing the rules so that the filters can enforce the new rules. It can also provide APIs for other tools, such as monitoring system APIs that get telemetry data.

In addition to the XDP program and the userspace process that loads it, there could also be additional userspace tools that might interact with the pinned BPF objects. For example, third-party monitoring system tools could implement logic to read the maps and push telemetry data.

Conclusion

eBPF and XDP have been a major step towards achieving programmability in the kernel's data path, which provides safety without compromising on speed. Beyond networking, eBPF has made a significant improvement in the tracing capabilities in the kernel, which has enabled instrumentations in areas that were previously not possible. The future of eBPF in the kernel is strong, and we will see more tools using the power of the BPF infrastructure.



Save the Date!

usenix

LISA18

October 29–31, 2018
Nashville, TN, USA

LISA: Where systems engineering and operations professionals share real-world knowledge about designing, building, and maintaining the critical systems of our interconnected world.

The Call for Participation is now available.
Submissions are due May 24, 2018.

Program Co-Chairs

Rikki Endsley
Opensource.com

Brendan Gregg
Netflix

www.usenix.org/lisa18

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Faults in Linux 3.x

TAPASWENI PATHAK

Tapasweni Pathak has completed her Bachelor's in computer science from IGDTUW (previously IGIT), Delhi. She has worked with Qualcomm Inc., SAP, and now is with Mapbox. She contributes to open source projects like Linux Kernel, OWASP, Debian Sources, X.org and ABI's Syster Org's projects. She is interested in studying more about decentralization of networks and operating systems as her research interest.
tapaswenipathak@gmail.com

Prior studies have used tools to find bugs in the Linux kernel versions 1 and 2. In this article, I share the results for faults in 3.x versions. This study is a continuation of the work by Chou et al. [1] for versions 1.0 to 2.4.1 and Palix et al. [2a–c] for 2.6 versions. I explain the types of bugs studied, trends for these bugs over newer versions, and how the reports were generated across the different Linux kernel versions.

In 2001, Chou et al. used static analysis tools run over each kernel version to get the results, and the number of common faults found was very high. By 2011, Linux kernel was in its third decade. Palix et al. found that the number of common faults decreased from the previous study results, implying better code quality in 2.6.x, but it was still very high. On February 8, 2015, Linux kernel version 3.19 was released. Patches are regularly submitted for faults found using checkpatch [3], Sparse [4], Coccinelle [5] and Smatch [6]. The number of lines of code in the Linux kernel also crossed 15M at this time. I wanted to follow the path of the previous studies and research how many bugs were in 3.x versions.

Methodology

Palix et al. used the open source tools Coccinelle [5], to automatically find faults in source code, and Herodotos [7, 2c], to run Coccinelle for each fault type and to track the faults across multiple versions of the Linux kernel. Coccinelle and Herodotos are available on the open access archive HAL [7, 2b]. Coccinelle is a tool for pattern matching and text transformation. To study the bed of faults it is necessary to understand the history behind them. When were they first released? When did they die if they did? Did they move after they were first introduced? Following the methodology deployed in the 2011 study, I used Coccinelle to automatically find problematic programming patterns in Linux kernels, and Herodotos to correlate these fault reports between different versions of the Linux kernel. The data about faults in this article were compared with the last study performed and helped to improve the reports generated for the study done on 3.x versions. As an example, there were cases where false positives previously reported moved around in different places in the code file.

Emac's org mode (orgmode.org), a text file format, was used to categorize the reports as bug or false positive. With this it was easier to move between different versions of the Linux kernel for the same report and study the history and reason behind a given bug type classification. This manual process was performed to make sure that none of the false positives generated were marked as bugs. I cloned all Linux kernel versions from 3.0 to 3.19 and considered the function stack, calls, and all possible inputs, outputs, Linux kernel standards, stack size etc. to categorize these reports as bugs or false positives. I also submitted patches for the bugs that were present in the then-current Linux kernel version.

In a few cases, I was not able to categorize the reports as bugs or false positives. In these cases, I used UNKNOWN/IGNORED.

The tools and marked reports generated were publicly made available in a GitHub repository [8].

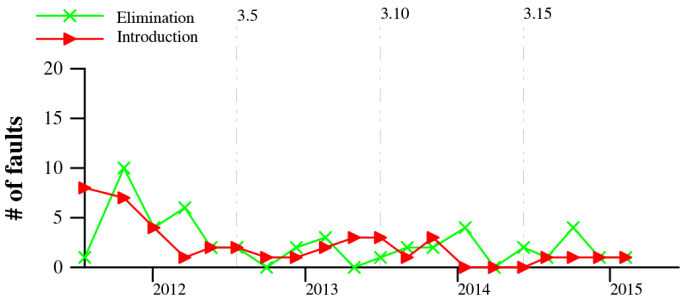


Figure 1: Birth and death of IsNull

After analyzing the reports, Nicolas Palix generated figures to highlight the rise and fall of the number of bugs in the different versions. I used org2sql to update the database of records for the Linux kernel versions 3.xx. org2sql tries to import all the faults and needs of at least two parameters: the prefix of files to drop (/fast_scratch/linuxes/) and the new.org file to import. The output is on stdout, which I then directed to an SQL file, which later was used with psql. All the figures were generated using these data and scripts [9].

Studied Fault Types and Their State

Inconsistent Assumptions about NULL

Dereferencing a pointer is undefined if the pointer is null. This fault type comes in two flavors: IsNull and NullRef. An IsNull fault is where a NULL test is done preceding a dereference, and a NullRef fault is where a NULL test is done following a dereference. The former is always an error, while the latter may be an error or may simply indicate overly cautious code, if the pointer can never be NULL.

Both fault types consistently decreased between versions 3.0 to 3.19. Figure 1 shows that the introduction of the IsNull bug type moved close to zero with Linux version 3.19 from the highest point with Linux version 3.0.

208 NullRef faults (Figure 2) were reported in total in Linux 3.19, and 112 of them were introduced in 3.0 or later.

As an example, a bug in Linux 3.15 occurred where a null check was done after referencing it inside the file drivers/staging/media/rtl2832u_sdr/rtl2832_sdr.c, line 992, in the function `rtl2832_sdr_start_streaming` for the *s* variable.

```
dev_dbg(&s->udev->dev, "%s:\n", __func__);
if (!s->udev)
```

An interesting false positive (FP) was found in Linux-3.11 inside the file net/nfc/llcp_core.c, lines 724 (null test) and 761 (nullref), in the function `nfc_llcp_tx_work()`, if `llcp_sock` is checked for null with one more condition (&&):

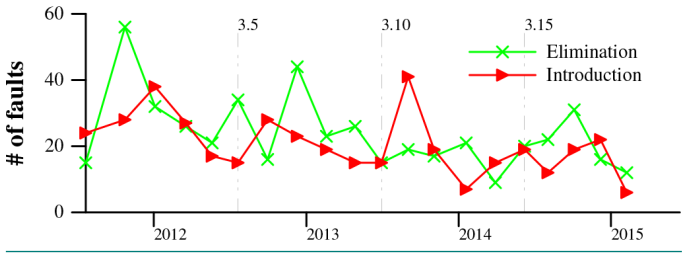


Figure 2: Birth and death of NullRef

```
if (llcp_sock == NULL && nfc_llcp_ptype(skb) == LLCP_PDU_I)
        .....
else if (llcp_sock && !llcp_sock->remote_ready)
        .....
```

Then inside the `else`, `llcp->sock` is dereferenced using

```
skb_queue_tail(&llcp_sock->tx_pending_queue, copy_skb);
```

The code is only a problem if `llcp_sock` is null and if `ptype == LLCP_PDU_I`. But `ptype` is defined as `u8 ptype = nfc_llcp_ptype(skb)`. And up at the top of the sequence of ifs there is another case for where `llcp_sock == NULL && nfc_llcp_ptype(skb) == LLCP_PDU_I`.

Disabling but Not Reenabling Interrupts

This includes interrupts that are turned off but not turned on again, using the function `spin_lock_irqsave`. `spin_lock_irqsave` is used to save the interrupt state before acquiring the spin lock. This is because spin lock disables the interrupt, when the lock is taken in interrupt context, and reenables it while unlocking or when using `local_irq_disable` and `local_irq_save`. The interrupt state is saved so that it can reinstate the interrupts again.

Locking but Not Unlocking and Double Locking

Double locking is a bug. This check looks for cases where a lock is taken but not released, that is, where an unlock is missing. In a few cases, interrupts are disabled at the same time that a lock is taken. Figure 3 shows that for the LockIntr bug type, the introduction rate reached its peak during 2013. With the introduction of Linux 3.9, the LockIntr rate fell to zero, implying there were no new LockIntr bugs that were produced with this release.

I even found a few interesting FPs where I plan to improve the semantic patch in Linux 3.5 inside the file kernel/workqueue.c at line 1013, in the function `__queue_work()`:

```
 spin_lock_irqsave(&last_gcwq->lock, flags);
 worker = find_worker_executing_work(last_gcwq, work);
if (worker && worker->current_cwq->wq == wq
gcwq = last_gcwq;
  else {
```

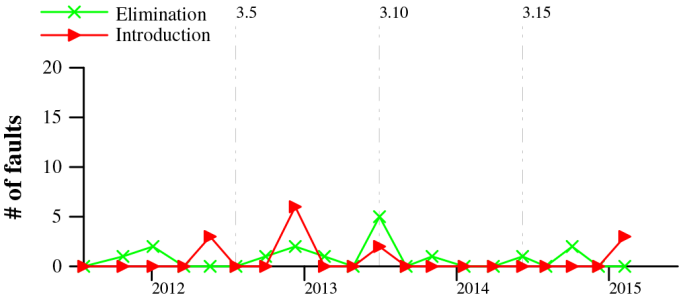


Figure 3: Birth and death of LockIntr

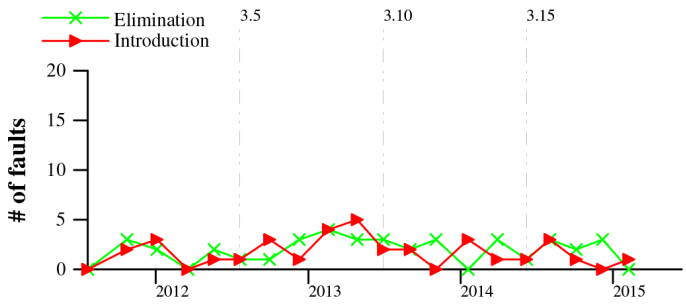


Figure 4: Birth and death of Lock

```
/* meh... not running there, queue here */
spin_unlock_irqrestore(&last_gcwq->lock, flags);
spin_lock_irqsave(&gcwq->lock, flags);
```

In the case where the unlock seems to be missing, there is the code `gcwq = last_gcwq`. In the case where the unlock is present, it is followed by the code `spin_lock_irqsave(&gcwq->lock, flags)`. That is, the whole set of nested `if`s terminates with the need to unlock `&gcwq->lock`. This lock is unlocked later. And in the case where `worker && worker->current_cwq->wq == wq`, it is the case that `gcwq = last_gcwq`, so the subsequent unlock of `&gcwq->lock` will unlock the `last_gcwq` lock because they are the same.

Calling Blocking Function with Interrupts Disabled or Spinlock Held

Blocking with interrupts disabled or a spinlock held can lead to deadlock. Basic memory allocation functions, such as the kernel function `kmalloc`, often take as their argument the constant `GFP_KERNEL` when `kmalloc` is allowed to block until a page becomes available. Thus, a function that contains a call with `GFP_KERNEL` as an argument may block.

However, blocking with interrupts turned off is not necessarily a fault, and indeed core Linux scheduling functions, such as `interruptible_sleep_on`, call "schedule" with their interrupts turned off. This issue was taken into account when checking for false positives.

This fault type checked for locks around possibly blocking functions.

Figure 4 shows that the birth and death of the Lock bug type had a fall. The slight increase in the introduction with Linux 3.19 is explained below.

In Linux 3.19, in the file drivers/staging/emxx_udc/emxx_udc.c at line 2797, inside the function `nbu2ss_ep_queue()`, `GFP_KERNEL` is used when calling `dma_alloc_coherent`. `GFP_KERNEL` was replaced with `GFP_ATOMIC` with a patch, as the latter will fail if the heap doesn't have enough free pages but will not sleep and hence avoids deadlock.

Wrong Use of `krealloc`

This fault type checked for a wrong use of `krealloc`. `krealloc` reallocates memory, while the contents of the memory remain unchanged. If `krealloc()`' returns NULL, it doesn't free the original pointer, which was pointing to the memory allocated. So any code of the form `foo = krealloc(foo, ...);` is certainly a bug. `krealloc` should use a temporary pointer for allocations and check the temporary pointer returned against NULL too.

For `krealloc` type reports, all reports were bugs and none were FPs in the case for 3.x versions. The most recent was in Linux-3.16 in the file drivers/pinctrl/sunxi/pinctrl-sunxi.c at line 740:

```
pctl->functions = krealloc(pctl->functions,
        pctl->nfunctions * sizeof(*pctl->functions),
        GFP_KERNEL);
```

If reallocation fails, `krealloc` will return NULL to `pctl->functions` without freeing the memory previously pointed to by `pctl->functions`.

Interrupts Turned Off but Not Turned On Again

Calling the `local_irq_save` function disables interrupts on the current processor and saves current interrupt state as flags (passed to this function). `local_irq_restore` function enables interrupts and restores state using the flags. In early versions of Linux, locks and interrupts were managed separately: typically interrupts were disabled and reenabled using `cli` and `sti`, respectively, while locks were managed using operations on spinlocks or semaphores. This fault type checked for the case where interrupts were turned off using the functions `local_irq_save` or `save_and_cli` but were not turned on again.

Figure 5 shows that in Linux kernel 3.17, this bug type had new introductions as well as eliminations. By Linux 3.19 both introduction and elimination reached zero.

I found a total of four bugs of this type. One was in Linux 3.17 in the file arch/mips/kvm/tlb.c at line number 206, inside method `kvm_mips_host_tlb_write()`:

```
local_irq_save(flags);
```

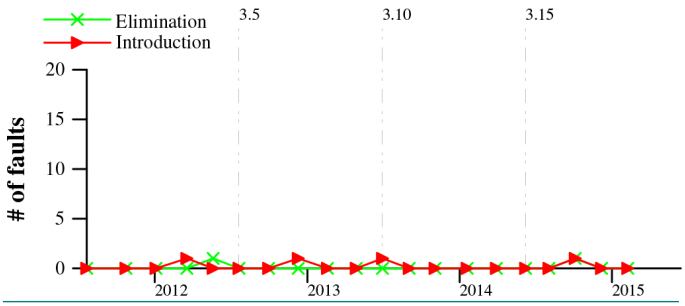


Figure 5: Birth and death of interrupt-related bugs

The interrupt state is saved so that it should reinstate the interrupts again, but, in this case, after the above call to `local_irq_save()`, there is no call to `local_irq_restore()`.

Using Freed Memory

`kfree` frees previously allocated memory. Using freed memory can cause the kernel to crash, can lead to a write-what-where condition, and can have consequences like corruption of valid data and the execution of arbitrary code. I checked for cases where there was a use after `kfree` and after a function that directly or indirectly calls `kfree`. The false positives were mostly when the variable freed was accessed only after a null check.

There were a lot of cases where `goto` was being used immediately after the `kfree`, which doesn't allow the statement to execute when using the freed memory. There were also many cases where an immediate return was done after `kfree`, and thus the statement where a variable accessed after `kfree` was not executed. There were cases where a check on the variable just freed (inside an `if`) was being done, hence avoiding a buggy situation.

Allocating Large Arrays on the Stack

All the local variables in the function are allocated on the stack. If too much memory is allocated on the stack, the kernel might run out of stack memory because the Linux kernel stack has a fixed size.

This fault type checked for instances where large arrays were allocated on the stack. I considered an array to be large if it contained more than 1023 bytes. Anything below that was marked as a false positive, and anything greater than that was considered a bug.

No new bugs of this bug type were introduced with Linux kernel version > 3.11. I found one FP of type var in Linux 3.11 in the file drivers/staging/lustre/lnet/klnds/socklnd/socklnd_cb.c at line 1034:

```
static char ksocknal_slop_buffer[4096];
```

In this case it was a global static variable, only visible in one function and not declared on the stack, so this was an FP.

Using Value Taken from User as Array Bounds and Loop Index without Check

Values taken from userspace should be checked for limits before using these values. A value could be huge, or it could be negative if the type of the field is not unsigned. `copy_from_user` is used to copy a block of data from userspace to kernel space. It then returns the number of bytes that could not be copied. On success, this will be zero. If some data could not be copied, this function will pad the copied data to the requested size using zero bytes. `get_user` is used to get a simple variable from userspace. This macro copies a single simple variable from userspace to kernel space.

This fault type checked for the case where unchecked values were obtained from the user level through `copy_from_user` and `copy_from_user` may be used as an array index or loop bound.

The Linux kernel from versions 3.0 to 3.19 had very few instances of introduction of this bug type. The Linux kernel 3.19 had no new user-value bug type introductions.

I found one `copy_from_user` type bug in Linux 3.12, in the file fs/btrfs/ioctl.c at line number 2736, inside the function `btrfs_ioctl_file_extent_same()`; `copy_from_user` is done using the `same` structure. `same->logical_offset` is then assigned to `off`, and `same->length` is assigned to `len`. The `len` variable is then checked for the maximum value it can have; if it exceeds that, it is assigned the maximum it can take. But later, the loop uses `same->dest_count` and not `len`.

I found one bug of type `get_user` in Linux 3.14 in fs/btrfs/ioctl.c at line number 2759, inside the function `btrfs_ioctl_file_extent_same()`'. No checks were done on count, and later it was used as an array index.

Wrong Assumption about Size of Object Being Allocated Memory

There were a total of 25 bugs relating to size type, all in <= 3.9 versions of the Linux kernel. A very simple way to identify this bug was in Linux 3.5 in the file drivers/net/wireless/mwifiex/ie.c at line 166. The two structures mwifiex_ie_list and mwifies_ie are different, which makes this usage buggy.

Using Floating Point Values

When a userspace process uses floating-point instructions, the kernel catches a trap for a floating-point instruction and then initiates the transition from integer to floating-point mode. This varies by architecture. In a kernel space process, the kernel cannot trap itself to support floating point. This is supported by manually saving and restoring the floating-point registers, among other chores. Saving and restoring floating point register state also makes floating-point operations slower than integer operations. People have always been advised not to use floating-point operations in the kernel.

Type	Reports	Bugs	Unknown
Kfree	304	138	1
	180 + 124	111 + 27	1 + 0
isNull	152	122	0
	108 + 44	80 + 42	
NullRef	1813	1578	24
	1313 + 500	1169 + 409	23 + 1
LockIntr	252	103	2
	186 + 66	89 + 14	2 + 0
Intr	35	23	0
	25 + 10	19 + 4	
Krealloc	25	21	0
	14 + 11	10 + 11	
Lock	674	230	4
	454 + 220	198 + 32	4 + 0
var	66	36	0
	51 + 15	35 + 1	
copy_from_user	5	5	0
	4 + 1	4 + 1	
get_user	25	19	1
	24 + 1	18 + 1	1 + 0
Float	549	46	0
	532 + 17	46 + 0	
size	214	52	0
	149 + 65	27 + 25	

Table 1: Number of bugs in Linux 2.6.x and 3.x versions. The first number in each row shows the total bugs for both 2.6.x and 3.x, and the pairs of numbers following are for 2.6.x first and 3.x second.

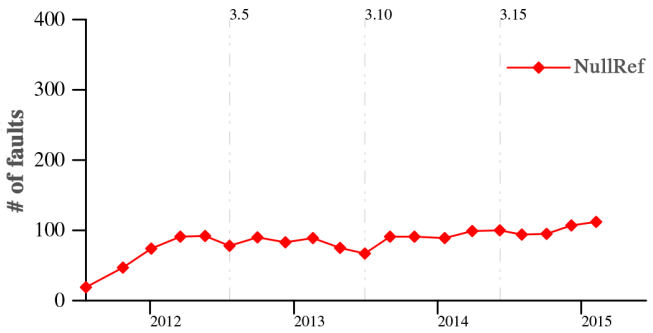


Figure 8: Count of bugs of NullRef type

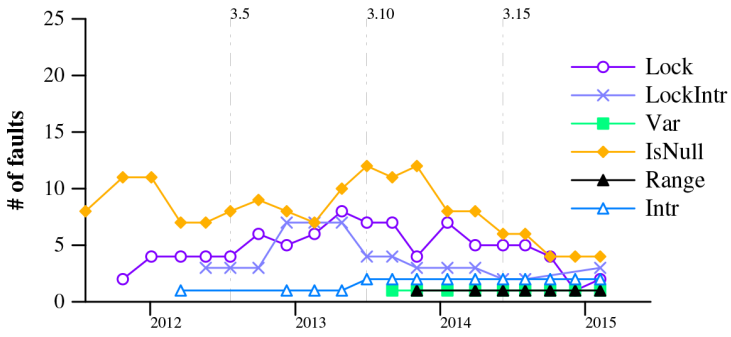


Figure 6: Overall birth and death of six fault types

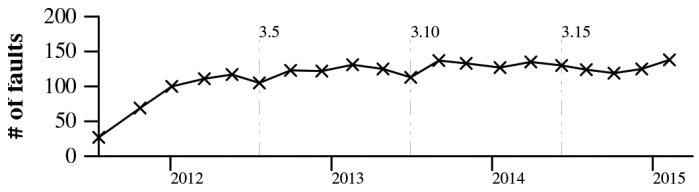


Figure 7 : Faults introduced with version 3.0 and after

This fault type checked for floating-point usages in kernel code. There was only one report for this case. Most false positives occur when the computation can be simplified at compile time to an integer. The checker only reports a floating-point constant that is not a subterm of an arithmetic operation involving another constant, and hence may end up in the compiled kernel code. Examples of false positive occurred where values like 1.6 * 1000 * 10 were being used.

The Overall Results

The total number of reports generated using the tools described in the Methodology section were 4114. This number constitutes the results generated by both 2.6.x and 3.x correlated reports. There were 1074 reports belonging only to 3.x, out of which 567 were bugs and 506 were false positives. We marked one as unknown. This table breaks down the numbers per each fault type. The second line in each cell breaks down the total into the numbers from 2.x and 3.x versions.

Overall Birth and Death of Faults

This graph indicates the number of each type of bug introduced in the 3.x versions and the number of bugs introduced and removed in each version.

All of the six fault types have decreased over the period of 2012 to 2015, with the greatest decrease being for the IsNull bug type. The Intr bug type, which was once zero, increased with the 3.10 version but has remained flat up to Linux 3.19. Figure 6 also suggests that these bug types did not reach zero until 2015.

Faults Introduced in 3.0 or After

Figure 7 shows the overall number of all the faults. The slope of the faults increases with newer Linux kernel versions, with NullRef being the highest (see Figure 8).

Future Work

Julia Lawall, Nicolas Palix, and I plan to study these fault types for Linux kernel 4.x.

Acknowledgements

Thanks to Julia Lawall, Inria Senior Research Scientist, and Nicolas Palix, Assistant Professor at University Grenoble Alpes, for working alongside me on this and for reviewing the article.

References

[1] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An Empirical Study of Operating Systems Errors," in *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP '01)*, pp. 73–88: http://www.stanford.edu/~engler/.

[2a] N. Palix, G. Thomas, S. Saha, C. Muller, J. Lawall, "Faults in Linux 2.6," *ACM Transactions on Computer Systems*, vol. 32, no. 2 (June 2014): https://arxiv.org/pdf/1407.4346v1.pdf.

[2b] N. Palix, S. Saha, G. Thomas, C. Calvès, J. Lawall, and G. Muller, *Faults in Linux: Ten Years Later* (Dec. 2010): http://faultlinux.lip6.fr/.

[2c] N. Palix, S. Saha, G. Thomas, C. Calvès, J. Lawall, and G. Muller, *Faults in Linux: Ten Years Later*, Research Report RR-7357, INRIA (July 2010): http://hal.inria.fr/inria-00509256.

[3] Checkpatch: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/scripts/checkpatch.pl.

[4] Sparse: https://sparse.wiki.kernel.org/.

[5] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and Automating Collateral Evolutions in Linux Device Drivers," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '08)*, pp. 247–260: https://www.cse.unsw.edu.au/~cs9242/08/exam/paper2.pdf.

[6] The Kernel Janitors, "Smatch, the Source Matcher," 2010: http://smatch.sourceforge.net.

[7] N. Palix, J. Lawall, and G. Muller, "Tracking Code Patterns over Multiple Software Versions with Herodotos," in *Proceedings of the ACM International Conference on Aspect-Oriented Software Development (AOSD '10)*, pp. 169–180: DOI:http://dx.doi.org/10.1145/1739230.1739250.

[8] GitHub, "Coccinelle/Faults in Linux: Experimental Bed to Study Linux Faults": https://github.com/coccinelle/faults-in-Linux.

[9] Scripts to generate graphs for Linux kernel fault study: https://github.com/npalix/linux-study-figures.

An Interview with Laura Nolan

RIK FARROW

Laura Nolan has been writing software since teaching herself to code in 1996. After earning her computer science degree at Trinity College Dublin, she worked as a Software Engineer for several years before joining Google as a Site Reliability Engineer in 2013. Currently, she is Tech Lead for a team of SREs working on Google's Edge Network in Dublin. Laura is also co-chair of the USENIX SREcon Europe/Middle East/Africa Conference as well as one of the (many) authors of the O'Reilly SRE book. When away from the keyboard, Laura can often be found traveling, hiking, scuba diving, or just enjoying a scotch and a good book (Longmorn and space opera are her particular weaknesses). lnolan@google.com

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

I first heard about Laura Nolan when she spoke at SREcon Europe in 2015. I watched the video of her talk [1], and that forms the basis of some of my questions.

Laura has since been a co-chair of SREcon, and we've had email discussions about potential *;login:* authors. Some of the people she suggested have written articles related to SRE for *;login:*.

I thought it was time to get to know Laura better, so I asked her if we could do this interview.

Rik Farrow: I watched your presentation at SREcon Europe about distributed consensus algorithms. I have enough systems background to know what that means, but perhaps you'd like to explain that in the context of SRE?

Laura Nolan: Well, distributed consensus is a really important building block of a lot of practical distributed systems. Any system where you need to get a group of processes to agree on something, as a whole, is solving distributed consensus. I think it's important for SRE to know something about it because it is a difficult problem, and trying to solve it in ad hoc ways can lead to some very surprising outcomes. So that talk, and the distributed systems chapter of the O'Reilly SRE book [2], is really trying to focus on showing what distributed consensus problems are and discussing the operational aspects of distributed consensus systems: their performance constraints, failure modes, monitoring, and so on.

Generally speaking, I think we need to raise our level here as a profession. SREs are working at the sharp end of distributed systems. Most of us don't have a solid background in them though; most engineering and computer science education is still pretty light on distributed systems content. We are figuring out these things on the fly all the time, and we're not being systematic enough about it.

Software engineers have design patterns...we need distributed systems reliability patterns!

So I'd love to see more distributed systems content at SREcon next year, building on some of the great content we had such as Theo Schlossnagle and John Looney's distributed systems workshop [3] and the reliable RPC talk and workshop presented by three Googlers (Grainne Sheerin, Lisa Carey, and Gabe Krabbe).

RF: In your talk, you mention using Paxos. I interviewed the primary author of Raft, Diego Ongaro, who said that one of the reasons for creating Raft was that Paxos was difficult to reason about. Yet Google seems to have settled on Paxos. Could you tell us the reasons for using Paxos instead of Raft (or ZAB)?

LN: Well, historical reasons is one part—Google was using Paxos heavily since before 2006, when the Chubby paper was published. The Raft paper [4] was published around 2014, and ZAB was also after Chubby. I'm not involved personally in any of this infrastructure at Google—I used to be on a team that ran a lot of Paxos-based data stores but have moved on—but as a software engineer, I will say that making that sort of change to any software system is expensive, and always more expensive than you think. Effort to rewrite a system is one thing, but then you have to test it, which is particularly onerous in the case of this sort of system, with many subtle failure modes. You'd need a really compelling reason to do it, and Raft and ZAB don't provide any immediate technical benefits over Paxos.

RF: How much opportunity have you had to learn new things and move into other areas? How does that compare with other places you have worked?

LN: Google certainly affords plenty of opportunity to work on different things and learn—this is one of the benefits of being a large organization. Mobility is encouraged reasonably strongly—not every few months, but if an SRE wants to move every few years that's seen as positive. It avoids teams becoming too siloed and set in their ways.

In the almost-five years I've been here, I've worked on three major areas that were all very different—big data stores and pipelines were the first, then an internal development project, and now I'm working on a team whose main focus is the reliability of our Edge Network and peering. So that's been a major change for someone without a network engineering background, and I've been working hard on "knowledge upload" related to both networking generally and our network specifically.

Even staying on one team, though, things do change—the major tools we work with evolve and are replaced, for instance; new projects are developed, and SRE teams will begin to support them. One big change in the last couple of years has been the introduction of Golang as the programming language of choice for most SRE automation projects, so everyone's learned Go.

RF: You mentioned that Golang has become the language of choice. What do you think of Go, and what were you using (or liked using) before Go, for contrast?

LN: I like Golang. It is relatively hard to shoot yourself in the foot with it compared to many other languages. It scales fairly well, has great concurrency constructs, and I am a fan of stronger typing. Generics would be nice, though, and I miss my ternary operator!

RF: John Looney has written about psychological safety on SRE teams. When I read his article, I felt like my entire work life would have been different if the teams I had worked with were more like the ideal John writes about. What has your experience been along the lines of psychological safety?

LN: I think that psychological safety is really important. My experiences as a member of teams have definitely been on a continuum from very psychologically unsafe to quite safe. It's certainly much more pleasant and way more productive to be in a team that is safer. Unsafe teams will burn you out faster than anything else, and burnout is the curse of Ops work.

Google SRE is pretty good, as these things go, in particular with respect to blame. We had an incident a few months ago that I think illustrates this nicely. We had an internal mishap that caused us to have a pager storm, and without going into the details, the trigger was a junior non-engineer who erroneously did something involving a mailing list. I know for a fact that multiple SREs specifically reached out to that person's manager afterwards to tell them that the incident was not that person's fault. None of them knew the individual personally, they just didn't want them to experience any negative consequences for the incident. I thought that was pretty great. And, of course, we've fixed the root cause of the pager storm too!

Tanya Reilly, an amazing SRE, gave a talk at LISA where she discussed what to do when someone breaks something in your system or finds a bug: you thank them for finding the gap and then you go fix it. That is for me an essence of psychological safety: no blame, no acrimony—just making the systems better.

On the flipside, I think the worst thing for psychological safety is the engineer who thinks they're smarter than everyone else and is constantly negative and critical about other people's ideas. That sucks the life right out of a team. Nobody, no matter how much of a rock star they are, is worth the kind of damage that causes. Don't be that person! Far better to be the engineer who makes everyone around them better than to be the one that makes everyone around them miserable.

RF: While at a WiAC meetup during NSDI '17 in Boston, one of the participants said that women have to work twice as hard as men do just to be noticed. You've worked in several organizations. What's your viewpoint on this?

LN: Different people have different experiences. I don't think I've ever been overlooked due to being a woman, but then I am assertive and outspoken, and it would be hard not to notice me. I did, however, once have a hilarious performance review (before I was at Google) with a male manager of mine where he spent about a solid hour telling me about how I talked too much. There was not an ounce of self-awareness there as I couldn't get a word in edgewise to actually discuss this issue properly. I didn't stay much longer in that organization.

I think there may be more truth in saying that there are some expectations for women that don't exist for men. There's a common antipattern where women end up doing a lot of the emotional labor in a team, even if it's not in their job description—things like organizing team events and recognizing occasions, and so on. Another huge thing is women engineers taking on things like taking minutes, organizing meetings, project management, building relationships with partner teams—some people call it being "the glue." Teams actually really need the glue to work well, but it can be under-recognized compared to coding because it's harder to measure. This also goes for men who are "glue" types. I also feel like it can be harder to get technical things done sometimes as women—I've seen things like excessively picky design and code reviews aimed at women more often than at men.

References

[1] L. Nolan, "Distributed Consensus Algorithms for Extreme Reliability," SREcon15 Europe: https://www.usenix.org/conference/srecon15europe/program/presentation/nolan.

[2] B. Beyer, C. Jones, J. Petoff, and N. Murphy, "Monitoring Distributed Systems," in *Site Reliability Engineering* (O'Reilly Media, 2016).

[3] J. Looney and T. Schlossnagle, "Distributed Systems-Reasoning," SREcon17 Europe: https://www.usenix.org/conference/srecon17europe/program/presentation/looney.

[4] D. Ongaro, J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *Proceedings of the 2014 USENIX Annual Technical Conference (ATC '14)*: https://web.stanford.edu/~ouster/cgi-bin/papers/raft-atc14.

Oslec, the Open Source Line Echo Canceller

An Interview with David Rowe

BOB SOLOMON

David Rowe has been working and playing with signal processing hardware and software for 30 years. In 2006 he left an executive position in the satellite communications industry to become a full-time open source developer. Since then David has worked on open hardware and software projects in VOIP, developing world communications, echo cancellation, speech compression, modems, and digital voice over HF radio. David writes a popular blog that is read by 70,000 people each month, drives a home-brew electric car, and also enjoys bike riding and sailing. www.rowetel.com

Phone System Administrator is one of too many hats Robert Solomon wears at a medium-sized nonprofit in New York City. bobsol@gmail.com

Echo is a pain point in open source telephony. The default echo cancellers provided with Asterisk are generally held to be unacceptable [1]. Hardware add-on modules, the common alternative, add $200 or more to the cost of a voice card [2] and are closed source.

In 2008, David Rowe completed development of an improved software echo canceller: Oslec, the Open Source Line Echo Canceller. Although Oslec has been well reviewed by those who have tested it [3], the code has not been incorporated into the Asterisk/DAHDI source. Nine years later, I became aware of Oslec while shopping for new voice hardware. David agreed to an email interview for *;login:*.

Robert Solomon: I am awed by the idea of software echo cancellation for Asterisk that actually works and that a lone developer (with help) could do this. Oslec has at least seven predecessors, none of which were usable IMO. How far along were you before you thought you might succeed?

David Rowe: I always knew it was possible. I mean an EC is just DSP software running on some sort of CPU. However, it took me nearly 20 years of failed attempts, first starting in the early 1990s as part of an early speech codec I was working on. So you might say determination played a part—an unscratched itch—something I had to "fix" at some point in my life.

There were a couple of key algorithms—a way to handle double-talk without diverging and the non-linear echo suppressor. When they dropped into place, I knew I was getting somewhere.

RS: When I first learned of Oslec, surprisingly on a vendor's page offering a hardware EC module [4], and then on your website [5], I thought that you must be really good at maths and algorithms or something. Reading your five-part Oslec blog [6], I learned that the magic of Oslec is more like framing the problem clearly, getting help from people who already know what they are doing, researching the literature, writing the code, and then reaching out to the community for testing. Would you comment? Applied this process in other areas?

DR: Yes, you have the process spot on. A couple of other points are:

1. Developing against the standards-based set of unit tests for EC, which was supported by a framework in Steve Underwood's spandsp library. This neatly isolated any issues and provided a binary pass/fail criteria to develop against.

2. Using open source and offline analysis to "crowd source" testing. Oslec was fitted with test points to capture the signals flowing to the EC. When a beta tester encountered a problem, they could run an application to capture some wave files, then email them to me for offline analysis. This quickly let me engineer solutions to corner cases: for example, low frequency audio from sound cards upsetting the analog hybrids (described in detail in one of the blog posts).

In contrast, everyone else was developing EC by saying "Hello 1,2,3..." down a telephone line in real time.

Oslec, the Open Source Line Echo Canceller: An Interview with David Rowe

I use a very similar process for other signal-processing projects, in particular the use of a high-level simulation that can run "offline" (not in real time) using data captured from the real-time version. For example, in my Codec 2 project, I have GNU Octave simulations that can single step through frames of speech, plotting various signals and statistics. Then, when I'm happy with performance, I run the same code in real time using a bit-exact C port of the same algorithm.

RS: So there is some science involved. "PhD in Electronic Engineering (topic: speech compression)." Want to say anything about that?

DR: Well, the software engineering process was just as important as the science. In addition to signal processing algorithm development, I have a parallel career as a project manager. I, and people working for me, have struggled at times with development of commercial signal-processing widgets. Turns out it's really hard to get the clever maths running effectively in real time in real-world products.

So after having screwed up and licked my wounds a few times, I worked out how to engineer complex signal-processing products effectively. I apply these ideas to my own projects and those I manage for others.

The PhD was on low bit rate speech compression and was completed in the late 1990s. About 10 years later, Bruce Perens approached me—there was a need for an open source low bit rate speech codec. Like echo cancellation, low bit rate speech compression is mired in closed source, license fees, and FUD. So I dusted off the PhD, and the Codec 2 project started [7].

RS: Although Oslec is the default echo canceller in Debian installations [8], echo.ko is not found in any Debian packages except user-mode-linux. I asked Tzafrir Cohen, a Debian DAHDI maintainer, about this in an email and he said that "Debian generally does not ship out-of-tree kernel binary modules." (He recommended building from Debian source with 'm-a'.) I am surprised that a module that was in staging in 2009 is not "in-tree" now. Could you comment, educate, explain?

DR: You know, I've lost track of the progress of Oslec through the staging process. I do recall there was some debate because it was a driver with no hardware. The kernel developer who was managing Oslec in the kernel, Greg KH, may have some comment. It's a good question, and I'd like to know the answer!

RS: Greg KH suggested that I "dig through the public email archives." With the help of Google, I found threads as new as 2012. This prompted me to actually browse the kernel, and I found that echo moved from staging to misc as of 3.15 [9]:

```
2014-02-28      staging: echo: move to drivers/misc/
Greg Kroah-Hartman      6       -0/+1181
```

DR: Yayyy, that means I'm officially a "kernel hacker." :-)

RS: I'm sure there is a good reason why Oslec was not implemented in userspace, but I don't know it. Would you explain?

DR: You need tight control of the delay in speech samples from the ADC/DAC signals flowing from the telephony hardware to the EC. Typically the kernel <-> user mode switch means buffering and timing uncertainty. For the Mesh Potato (village Telco project), I did the EC in user mode, as I built in careful control of the buffering in the kernel mode driver I wrote.

RS: I am noticing a decline of open source telephony in that vendors who once supported "Linux" now, in one case, support only CentOS 6 or worse and, in another case, their special variant of CentOS 7. Would this be due to mobile, to vendor business model, or to the decline of the white box phone system? Some other reason?

DR: I don't feel I have any useful knowledge on this one. I'm not involved in phone systems or Asterisk anymore myself. Given the rise of mobile phones I do wonder about the long-term viability of any sort of PBX; in my day job (a seven-person startup), we don't even have a landline.

RS: Your work on Oslec was complete in 2008 or so. What are you up to these days?

DR: Oslec was developed for the IP04, an open source embedded IP-PBX that I developed. I sold and supported the IP04 for many years in partnership with Atcom, but sales dropped off and, from a technical point of view, I lost interest.

Since 2009 my main project has been Codec 2, a low bit rate (3200 to 700 bits/s) open source speech codec. In the last few years I've been working on the problem of digital voice over HF radio. Making some progress but haven't beat legacy analog Single Sideband (SSB) yet. Along the way, I've developed several high quality modems for digital radio and discovered some more marketing-based FUD—not unlike that around hardware EC. My blog (rowetel.com) is also very popular.

I recently joined an Australian startup (solinnov.com.au) that does contract-based FPGA-based signal processing development for communications and defense applications. They are growing rapidly, and I'm helping out with some high-level signal processing, project management, and company process to help them grow. I'm working there a few days a week, plus keeping busy as a Dad, and I also enjoy sailing my little 16-foot "trailer sailer" and riding my bike.

RS: A little more about the trailer sailer? Picture?

DR: Sure—it's a Hartley TS16 16-foot sailing boat popular in Australia and New Zealand. It lives at my home, and once a week I tow it down to the sea and have a day out with friends and family.

References

[1] J. Van Meggelen, L. Madsen, and J. Smith, *Asterisk: The Future of Telephony*, 2nd edition (O'Reilly, 2007), p. 201: https://ftp.openbsd.org/pub/OpenBSD/distfiles/9780596510480.pdf.

[2] Quick price comparison: https://www.voipsupply.com/.

[3] Oslec Echo Canceller: http://www.rowetel.com/?page_id=454.

[4] https://xorcom.com/product/voip-hardware-echo-cancellation/.

[5] http://www.rowetel.com/?page_id=454.

[6] Open Source Echo Canceller: http://www.rowetel.com/?p=18.

[7] http://www.rowetel.com/?p=128.

[8] See /usr/share/doc/dahdi/README.Debian.

[9] https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git/log/drivers/misc/echo?h=linux-3.15.y.

Flex Dynamic Recording

TIMOTHY FELDMAN

Tim Feldman works on drive design at the Seagate Technology Design Center in Longmont, Colorado. His current work focuses on cloud storage. He also spends time randonneuring, Nordic skiing, and logging.
timothy.r.feldman@seagate.com

Hard disk drives are capable of various recording methods without changing the hardware. For example, shingled magnetic recording (SMR) is a technique in which higher track density is obtained, but with a tradeoff of requiring sequential writing through a band of tracks. I introduce a new way of letting the storage manager dynamically, in the field, specify different recording methods for different parts of the media in the device. This article prepares readers for this upcoming disk technology by describing the opportunities to lower the total cost of ownership (TCO) and exploring the new device interface that needs to be defined.

In February 2016, Google's "Disks for Data Centers" white paper [1] proposed options to improve the total cost of ownership, speed, tail latency, and capacity of hard disk drives. One particular capacity improvement was for an implementation in which conventional and shingled recording are mixed in a single, hybrid hard disk drive. The white paper noted that if the outer tracks are conventional magnetic recording (CMR) and the inner tracks are SMR as shown in Figure 1, the outer tracks could be used for short-lived data enjoying the random write performance of CMR, while the inner tracks could hold long-lived data using higher density SMR.

More recently, Google has presented an initial set of requirements [2], and both Seagate and Western Digital have signaled their support in blog posts [3, 4]. In this article, I will refer to the ability to dynamically mix recording methods in a single disk as Flex, Seagate's name for the technology, alternately called Hybrid SMR and Realms by Google and Western Digital, respectively.

Problems and Opportunities

The range of tracks accessed by a workload is known as the *stroke*, referring to the range of motion of the heads. An application that accesses the full logical block address (LBA) space uses 100% of the stroke. If a disk is partitioned into 10 volumes, then each volume uses about 10% of the stroke—more accurately, 6.7% for the partition at the lowest LBAs, 10% in the middle, and 13.3% at the highest LBAs due to the variation in the number of sectors per track, with outer tracks at about twice the capacity of inner tracks. Note that this correspondence of logical addresses to physical radius assumes the conventional logical-to-physical disk mapping in which lower LBAs are on the outer tracks. With this partitioning, if the disk workload is restricted to a single partition, then the workload uses about 10% of the stroke. And since access time is highly sensitive to seek distance, this constrained workload is much faster than a 100% stroke workload.

In practice, accesses to hot data can be sped up by constraining it to a limited range of LBAs, a technique generally known as *short stroking*. This not only increases the I/Os per second (IOPS), but greatly increases the performance density (IOPS per TB) since the denominator of that term gets smaller. Figure 2 shows the relationship between performance density and stroke based on a first-order model of disk performance. Note that performance can increase by 4x just by short-stroking to 33%.

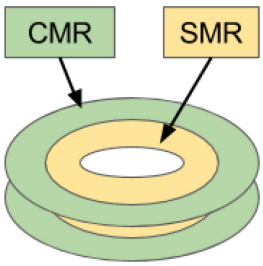


Figure 1: Depiction of a two-platter hybrid hard disk drive with CMR at the outer tracks and SMR at the inner tracks

But if short stroking is the only technique applied, and only 33% of the disk is used, there is unused media in the other 67% of the stroke. Finding a way to use this media while retaining the performance of the hot data is a TCO improvement opportunity.

Cold data, in contrast to hot data, can generally be written sequentially. Using SMR for cold data lowers the cost per byte. But after filling a disk with only cold data, the disk actuator arm will mostly sit idle. Finding a way to keep the disk mechanics busy serving useful I/Os is another TCO improvement opportunity. If a deployment has both hot and cold data, then a solution of segregated tiers not only leaves both TCO improvement opportunities unrealized, but also doubles the logistical complexity of managing two tiers and their unique drive types.

Flex, the ability to dynamically mix recording methods, allows the operating system to configure a single drive to a mix of CMR and SMR. And the mix can change to match a changing mix of hot and cold data. This means that hot data can enjoy the performance benefits of short stroking while cold data makes use of the rest of the media. The disk is then fully subscribed; all of its media and all of its mechanical capability are utilized, and the total cost is minimized.

Flex is not limited to just mixing CMR and SMR. There are other ways to improve TCO, speed, tail latency, and capacity. An idea as simple as using Flash in SLC or MLC mode provides one set of tradeoffs. Heat- or microwave-assisted magnetic recording may be able to record in different track widths by modulating the laser or microwave power and mixing track widths in an interlaced manner, as depicted in Figure 3, which increases the data density and, thus, disk capacity [5].

Interlaced magnetic recording (IMR) does not actually use different physical layers. Instead, "bottom" tracks are simply the wider tracks and "top" tracks are the narrower tracks. Since writing a bottom track can make two top tracks unreadable, IMR

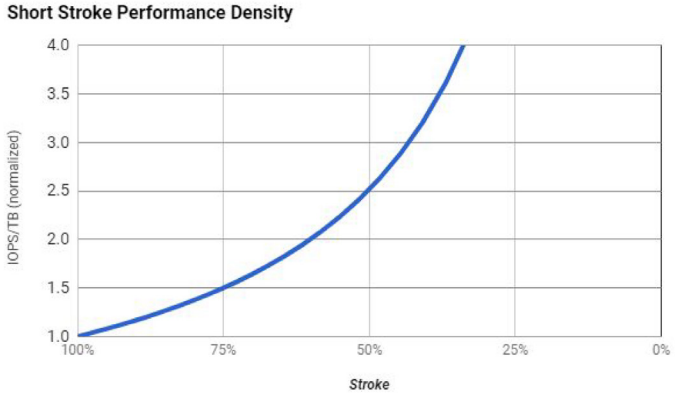


Figure 2: Performance density increase of small, random accesses from short stroking

presents a track write sequence problem similar to SMR, and the solutions invented for SMR can be applied [6]. For instance, 256 MiB worth of interlaced tracks can be mapped as a contiguous set of LBAs; this 256 MiB extent is then a logical zone, and zones can be managed as regions that must be sequentially rewritten. Or other innovative techniques might be used to manage top and bottom tracks. Beyond IMR, there are other ideas in the pipeline not yet in the public domain.

When various techniques can coexist on the same physical device, the fundamental Flex proposition of letting the OS select what recording method to use on a specified set of media is the most flexible solution.

Toward a Flex API

There is no existing API that allows an OS to change the configuration of a block device. A new interface needs to support conversions between the recording methods, and should include API improvements that kernel developers have been requesting for many years.

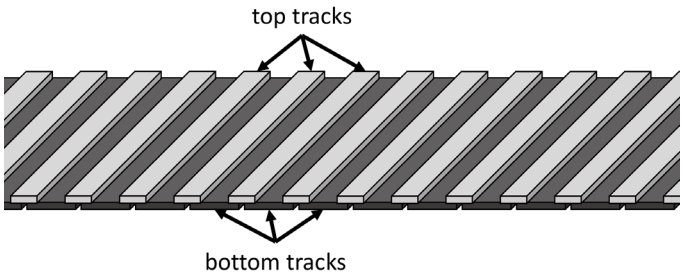


Figure 3: Depiction of interlaced track recording

Here are our goals for a new, superlative interface.

1. Provide backward compatibility
2. Leverage existing protocols
3. Support both ATA and SCSI to enable both SATA and SAS devices
4. Allow diverse configurations with fine-grained assignment of the recording method
5. Have completely discoverable capabilities
6. Enable on-the-fly conversions between recording methods
7. Protect against the loss of locked data or data that is still valid to some application
8. Be extensible to future recording methods

The Open Compute Project, T10 and T13 standards groups, will engage in the work of defining an API that meets these goals in 2018.

Providing Backward Compatibility

Flex devices should typically leave the factory in a configuration that allows existing software that is oblivious about the new capabilities to use the device. This implies that not only is a 100% CMR configuration supported but that this is the initial state. A Flex drive should be able to return to this configuration from any state.

Leveraging Existing Protocols

Cooperatively managed SMR, now codified as the zoned block device model, is the natural starting point. Zones are contiguous sets of LBAs and are always 256 MiB. Zones are either conventional zones without write pointers (for CMR space) or write pointer zones (for SMR space). The zoned block device model even already has an Offline zone state.

There are two observations about conversions that need to be addressed. First, conversions should be fast so that Flex does not introduce commands that take longer than any existing commands. It is important not to break the command timeout model that drivers use to detect dead drives.

Second, there may be valuable data that can be used to initialize space that has just come online as opposed to formatting with fill data only to immediately write the same media with valuable data. A conversion to SMR can finish with all of the space that just came online to be write pointer zones in the Empty state. This allows the device to skip initializing the SMR media with readable fill data, a process that takes about one second per 256-MiB zone. But we also want conversions to CMR space to be just as fast. The obvious extension to the zoned block device model is to define a new zone type for CMR space that also starts off as Empty, but unlike an SMR zone would have no performance penalty for random writes below the write pointer. Both CMR and SMR zones must either fail reads above the write pointer or return zeros, the former catching improper reads and the latter mimicking formatted media.

Supporting ATA and SCSI

Since ATA does not support logical unit numbers (LUNs), the Flex protocol should use separate LBA ranges for the CMR and SMR spaces. This can extend to more than two ranges when the device supports more than two recording methods when Flex is extended beyond just CMR and SMR.

For maximum flexibility, both queued and non-queued commands should be defined. And by co-developing ATA and SCSI, we can end up with a straight-wire SCSI to ATA translation (SAT) layer.

Allowing Diverse Configurations

To allow each storage stack to pursue its own optimal design point, conversions should be fine-grained. As part of embracing the zoned block device model, we want the zone to be the unit of conversion and the minimum allocation unit of the top-level allocator; that is, each zone is either online or offline, and a conversion can target any contiguous extent of zones.

For maximum short stroking benefit, all of the CMR space should be contiguous. But there may be other configurations needed. For instance, a 10-TB drive chopped into 10 one-TB pieces for 10 different tenants may want each tenant to have a CMR space and an SMR space. Thus, multiple "seams" between differing recording methods should be allowed, albeit with a small efficiency loss that averages one-half zone at each seam.

Discovering Capabilities

Device discovery includes detecting the device type through its signature and Identify Device data. Due to the backward-compatibility goal, Flex devices should identify as conventional disks. They also need to report the 100% CMR capacity in the existing capacity reporting fields.

Capabilities discovery then allows a host to learn what features a device supports. Simple additions to ATA logs and SCSI vital product data pages can serve to alert a stack that is cognizant of Flex to find out whether a Flex device is present. From there, existing zoned block device mechanisms, including Report Zones, can expose the SMR space in addition to the CMR space.

Enabling On-the-Fly Conversion

Availability is critical. Conversions need to be allowed as part of the normal workflow and not be restricted to system integration or an offline mode. Conversion commands need reordering constraints if they overlap reads or writes to the same LBAs, but

the rest of the LBA space that is not participating in a conversion needs not only to retain its data, but concurrent reads and writes must be allowed.

Protecting Valid and Locked Data

Before a conversion that takes space offline, any data in that space that is still valid for some application needs to be copied, either to space on this disk that will stay online or elsewhere. Since a conversion operation has a side effect of making previously written data unreadable, a conversion that gets ahead of the valid data copy process will lose data.

The existing SMR zone types support an operation, Reset Write Pointer, for the host to move a zone's write pointer back to the start of the zone. Since reads to LBAs above a write pointer either fail or return zeros, this also declares that the previously written data are discarded. Extending the Reset Write Pointer operation to the CMR zones allows a strong, firm handshake in the protocol: requiring that a zone's write pointer is reset before it is allowed to be converted to Offline, the conversion itself has no data retention side effects.

Similarly, enforcing that zones must be unlocked for a conversion allows a security management layer to know that locked data cannot be lost through execution of new Flex commands.

Being Extensible

New zone types can be defined as needed to support techniques like interlaced tracks. Other innovations might pack data more densely in other ways, but the tradeoffs often break legacy requirements. Simply getting all of the media provisioned to user-addressable space has been boxed in by ingrained assumptions about static configurations.

While the first generation of Flex will address the hybrid mix of CMR and SMR, the protocol needs to be extensible. This means that capabilities reporting and conversion commands need to be open to more than just two recording methods.

Adding Value

Flex Dynamic Recording recognizes that a single hardware configuration can be deployed in various ways, all the way down to physical recording methods on media. The philosophy of Flex is that allowing the owner of a device to configure what recording method is best for them adds value to the whole system. So rather than locking down the method at the factory, Flex moves the decision to the field.

References

[1] E. Brewer, L. Ying, L. Greenfield, R. Cypher, and T. Ts'o, "Disks for Data Centers": https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/44830.pdf.

[2] T. Ts'o, "Hybrid-SMR Product Requirements Proposal for OCP": http://files.opencompute.org/oc/public.php?service=files&t=50192ac3fff6f7d96c314dc39cd92f26.

[3] Seagate, "New Flex Dynamic Recording Method Redefines the Data Center Hard Drive": https://blog.seagate.com/intelligent/new-flex-dynamic-recording-method-redefines-data-center-hard-drive/.

[4] Western Digital, Dynamic Hybrid SMR: https://itblog.sandisk.com/dynamic-hybrid-smr/.

[5] E. Hwang, J. Park, R. Rauschmayer, and B. Wilson, "Interlaced Magnetic Recording (IMR)," *Journal of Transactions on Magnetics*, vol. 53, no. 4 (April 2017): http://ieeexplore.ieee.org/document/7781604/.

[6] T. Feldman and G. Gibson, "Shingled Magnetic Recording: Areal Density Increase Requires New Data Management," *;login:*, vol. 38, no. 3 (USENIX, June 2013): https://goo.gl/wj5Doi.

Miniature Cache Simulations for Modeling and Optimization

CARL A. WALDSPURGER, TRAUSTI SAEMUNDSSON, IRFAN AHMAD, AND NOHHYUN PARK

Carl Waldspurger is an Independent Consultant and Technical Advisor, collaborating on research and development projects with several companies. Carl has a PhD in computer science from MIT and has served as program chair for USENIX ATC, FAST, and VEE. His research interests include resource management, virtualization, caching, computer architecture, and security. carl@waldspurger.org

Trausti Saemundsson is a Software Engineer at Google working on datacenter software. He has an MSc in computer science from Reykjavik University in Iceland. His research interests are systems, analytics, caching, machine learning, security, and optimizations. He has papers published at SoCC, USENIX ATC, and GLBIO. trauzti@gmail.com

Irfan Ahmad is the founder of CachePhysics. Irfan works on interdisciplinary endeavors in memory, storage, CPU, and distributed resource management. He has published at ACM, USENIX, and IEEE. He has served as program chair for HotCloud, HotStorage, and HotEdge. mr.irfan@gmail.com

Nohhyun Park is a Software Engineer at DatosIO working on data protection for distributed databases. He has a PhD in electrical and computer engineering from the University of Minnesota and is interested in workload characterization and performance modeling for large-scale systems. nohhyun.park@datos.io

We present a surprisingly simple technique that accurately models the behavior of a cache with *any* policy by simulating a scaled-down *miniature cache* with a small, spatially hashed sample of requests. We also demonstrate how to leverage such models to optimize caches dynamically, using scaled-down simulations to explore multiple cache configurations simultaneously.

Caches are ubiquitous in modern computing systems, improving system performance by exploiting locality to reduce access latency and offload work from contended storage systems and interconnects. A wide variety of caches have been implemented in hardware and software, clients and servers, storage arrays, key-value stores, and other system infrastructure.

By definition, a cache is a small, fast memory backed by larger, slower storage. As a result, cache space is inherently scarce, and methods that can better utilize this space are extremely valuable. Techniques for accurate and efficient cache modeling are especially important for informing cache allocation and partitioning decisions, optimizing cache parameters, and supporting goals including performance, isolation, and quality of service.

However, caches are notoriously difficult to model. It is well known that performance is non-linear in cache size due to complex effects that vary enormously by workload. Although recent research has produced practical models for LRU caches, there has been no general, lightweight solution for more sophisticated policies, such as ARC [7], LIRS [4], and 2Q [5].

Modeling Caches with MRCs

Cache utility curves plot a performance metric as a function of cache size. Figure 1 shows an example miss-ratio curve (MRC), which plots the ratio of cache misses to total references for a workload (y-axis) as a function of cache size (x-axis). The miss ratio generally decreases as cache size increases, although complex algorithms such as ARC and LIRS can exhibit non-monotonic behavior due to imperfect dynamic adaptation.

MRCs are valuable for analyzing cache behavior. Assuming a workload exhibits reasonable stationarity at the time scale of interest, its MRC can also predict future performance. Thus, MRCs are powerful tools for optimizing cache allocations to improve performance and achieve service-level objectives.

Mattson et al. introduced a method for constructing MRCs for *stack algorithms*—for example, LRU, LFU, etc.—that yields the entire MRC for all cache sizes in a single pass over a trace [6]. Efficient modern implementations of this algorithm have an asymptotic cost of $O(N \log M)$ time and $O(M)$ space for a trace of length N containing M unique blocks. Recent approximation techniques can construct accurate MRCs with dramatically lower costs than exact methods. In particular, SHARDS [9] and AET [3] require only $O(N)$ time and $O(1)$ space, with a tiny footprint of approximately 1 MB. However, for more complex non-stack algorithms, such as ARC and LIRS, there are no known single-pass methods. As a result, separate runs are required for each cache size, similar to pre-Mattson modeling of LRU caches.

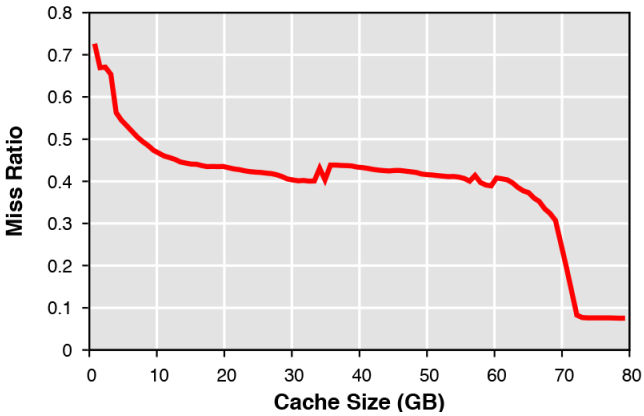


Figure 1: Example MRC. Miss-ratio curve for a production disk block trace using the ARC cache algorithm. The ratio of cache misses to total references is plotted as a function of cache size.

Miniature Simulation

The main idea behind miniature simulation is to approximate the behavior of a large cache by simulating a tiny one that processes only a tiny sample of its requests. Typically, the cache size and the input reference stream are both scaled down by several orders of magnitude.

A mini-simulation runs the full, unmodified cache replacement algorithm, making it possible to model *any* caching algorithm, including even ad hoc modifications often found in production systems. The miss ratio and other metrics are determined by simply extracting the usual statistics from the mini-cache, such as counts of misses and references. An adjustment that instead uses the *expected* number of references reduces bias due to sampling error significantly [8].

The reference stream is scaled down by using hashing to randomly sample the key space. A reference is sampled only when the hash value of its associated key is smaller than a threshold T that defines the sampling rate R. This approach is similar to our earlier work on SHARDS and is also related to sharding in distributed databases. Depending on the cache, a *key* may be a memory address, a logical block number for disk storage, or a string, as in a key-value store. The effectiveness of scaling depends on statistical self-similarity—that a randomized sample is fairly representative of the whole. As we will see, this is a good assumption that holds well in practice.

Figure 2 depicts a full-size cache and its input references, along with two scaled-down versions. To randomly sample the input, simple *temporal* sampling, such as flipping a coin for each reference, doesn't work. We must ensure that all references to the same key are always sampled or we will be blind to reuses that are central to caching behavior. Instead, randomized *spatial* sampling is implemented by selecting references based on deterministic hashes of their keys. In the figure, hash values are represented visually with shading. Scaling down by a factor of two results in a cache with half the size, and an input stream from half the key space, for example, by sampling a key only when the high-order bit of its hash is zero, shown as yielding half of the original shades. Similarly, scaling down by a larger factor of 128 shrinks both the cache size and the key space more dramatically.

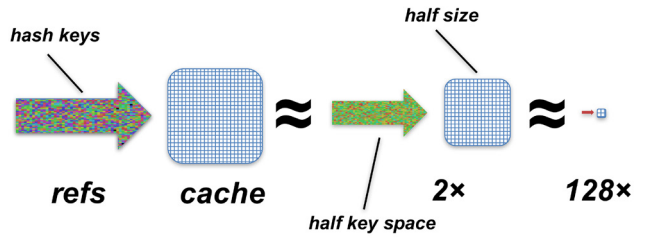


Figure 2: Scaling Down. Both the cache size and input reference stream are scaled down by factors of 2 and 128; each exhibits similar behavior. Only keys that fall within a subset of the hash space are sampled.

Scaling the key space and the cache size by the same amount maintains the same pressure on a mini-cache as the full-size cache, so it should exhibit approximately the same behavior. A cache of size S can be emulated by scaling down the cache size to $R \cdot S$ and scaling down the reference stream using a hash-based spatial filter with sampling rate R. In practice, sampling rates on the order of $R = 0.01$ or $R = 0.001$ yield very accurate results, achieving huge reductions in space and time compared to a conventional full-size simulation.

More generally, scaled-down simulation need not use the same scaling factor for both the miniature cache size and its reference stream. The emulated cache size S_e, mini-cache size S_m, and input sampling rate R are related by $S_e = S_m / R$. Thus, S_e may be emulated by specifying a fixed rate R, and using a mini-cache with size $S_m = R \cdot S_e$, or by specifying a fixed mini-cache size S_m and sampling its input with rate $R = S_m / S_e$. In practice, it is useful to enforce reasonable constraints on the minimum mini-cache size (e.g., $S_m \geq 100$) and sampling rate (e.g., $R \geq 0.001$) to ensure sufficient cache space and enough sampled references to simulate meaningful behavior.

Scaled-Down MRCs

For non-stack algorithms, there are no known methods capable of constructing an entire MRC in a single pass over a trace. Instead, MRC construction requires a separate run for each point on the MRC, corresponding to multiple discrete cache sizes. Fortunately, we can leverage miniature caches to emulate each size efficiently.

We evaluate the accuracy and performance of our approach with three diverse non-LRU cache replacement policies: ARC [7], LIRS [4], and the theoretically optimal OPT [2]. We use a collection of 137 real-world storage block trace files, similar to the

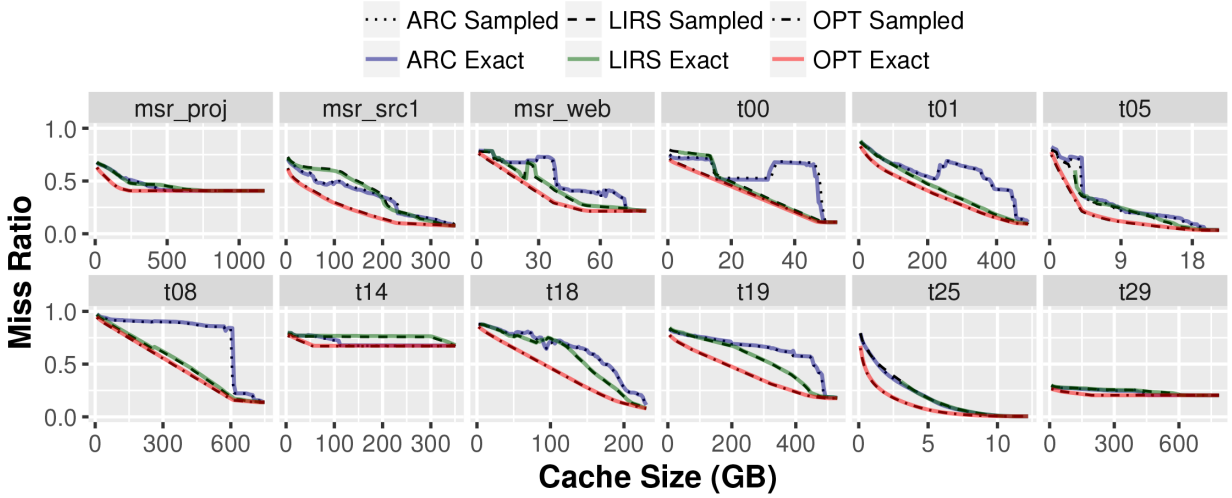


Figure 3: Example Mini-Sim MRCs. Exact and approximate MRCs for 12 representative traces. Approximate MRCs are constructed using scaled-down simulation with sampling rate R = 0.001. Each line type represents a different cache algorithm.

SHARDS evaluation [9]. These represent 120 week-long virtual disk traces from production VMware environments collected by CloudPhysics, 12 week-long enterprise server traces collected by Microsoft Research Cambridge, and five day-long server traces collected by FIU. For our experiments, we use a 16-KB cache block size, and misses are read from storage in aligned, fixed-size 16-KB units. Reads and writes are treated identically, effectively modeling a simple write-back caching policy.

Accuracy

For each trace, we compute MRCs at 100 discrete cache sizes, spaced uniformly between zero and a maximum cache size. To ensure these points are meaningful, the maximum cache size is calculated as the aggregate size of all unique blocks referenced by the trace.

Figure 3 contains 12 small plots that illustrate the accuracy of approximate MRCs with R = 0.001 on example traces with diverse MRC shapes and sizes. In most cases, the approximate and exact curves are nearly indistinguishable. In all cases, miniature simulations model cache behavior accurately, including complex non-monotonic behavior by ARC and LIRS. These compelling results with such diverse algorithms and workloads suggest that scaled-down simulation is capable of modeling nearly any caching algorithm.

To quantify accuracy, we compute the difference between the approximate and exact miss ratios at each discrete point on the MRC, and aggregate these into a mean absolute error (MAE) metric, as in related work [9, 3]. The box plots in Figure 4 show the MAE distributions for ARC, LIRS, and OPT with sampling rates R = 0.01 and R = 0.001. The average error is surprisingly small in all cases. For R = 0.001, the median MAE for each algorithm is below 0.005, with a maximum of 0.033. With R = 0.01, the median MAE for each algorithm is below 0.002, with a maximum of 0.012.

Performance

For our performance evaluation, we used a platform configured with a six-core 3.3 GHz Intel Core i7-5820K processor and 32 GB RAM, running Ubuntu 14.04. Experiments compare traditional exact simulation with our lightweight scaled-down approach. In all cases, simulations track only metadata, and do not store data blocks.

Resource consumption was measured using our five largest traces. We simulated three cache algorithms at five emulated sizes S_e (8 GB, 16 GB, 32 GB, 64 GB, and 128 GB), using multiple

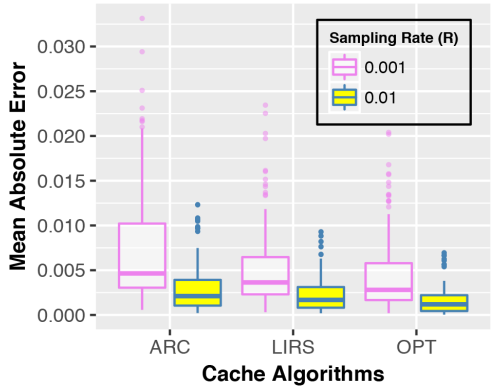


Figure 4: Error Analysis. Distribution of mean absolute error for all 137 traces with three algorithms (ARC, LIRS, OPT) at two different sampling rates (R = 0.01, R = 0.001).

	Linear Function		Example Trace (t22)	
Policy	**Fixed**	**Variable**	**R=.001**	**R=1**
ARC	1.37 MB	71 B	1.57 MB	284 MB
LIRS	1.59 MB	75 B	1.80 MB	301 MB
OPT	7.10 MB	37 B	19.55 MB	18,519 MB

Table 1: Memory Footprint. Memory usage for ARC and LIRS is linear in the cache size, $R \cdot S_e$, while for OPT, it is linear in the number of sampled references, $R \cdot N$. Measured values are shown for CloudPhysics trace t22 with S_e = 64 GB.

sampling rates R (1, 0.1, 0.01, and 0.001) for a total of 60 experiments per trace.

Unsurprisingly, the memory footprint for cache simulation is a simple linear function consisting of fixed overhead (for policy code, libraries, etc.) plus variable space. For ARC and LIRS, the variable component is proportional to the cache size, $R \cdot S_e$. For OPT, which must track all future references, it is proportional to the number of sampled references, $R \cdot N$. Table 1 reports the fixed and variable components of the memory overhead determined by linear regression ($r^2 > 0.99$). As expected, accurate results with $R = 0.001$ require 1000x less space than full simulation, excluding the fixed overhead.

We also measured the CPU usage consumed by our single-threaded cache implementations with both exact and scaled-down simulations for ARC, LIRS, and OPT. The runtime consists of two main components: cache simulation, which is roughly linear in R, and sampling overhead, which is roughly constant; each reference must be hashed to determine if it should be sampled. The scaled-down simulation with $R = 0.001$ requires about 10x less CPU time than full simulation, and achieves throughput exceeding 53 million references per second for ARC and LIRS, and 39 million references per second for OPT. Fortunately, for multi-model optimization, hash-based sampling costs are incurred only once, not for each mini-cache. In an actual production cache, the cost of data copying would dwarf the hashing overhead. Moreover, a separate hash for sampling isn't needed if one is already available; storage caches and key-value stores typically hash keys for performing lookups.

Cache Optimization

A single cache instance runs with a single policy and a single set of configuration parameters. Unfortunately, policy and parameter tweaking is typically performed only at design time, considering few benchmarks.

Low-cost online modeling allows efficient instantiation of multiple concurrent models with different cache configurations, offering a powerful framework for dynamic optimization. Quantifying the impact of hypothetical parameter changes allows the best settings to be applied to the actual cache. Such a multi-model approach can optimize cache block size, write policy, algorithm-specific tunables, or even replacement policy.

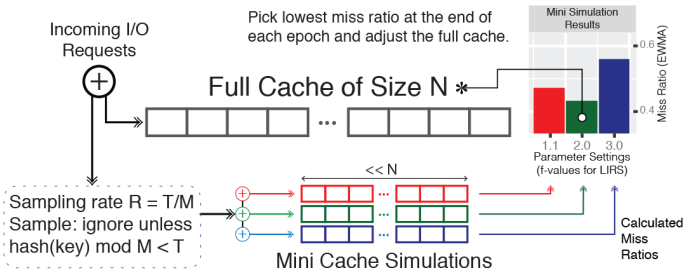


Figure 5: Online Optimization. Simultaneous miniature simulations enable automatic selection of the best parameter setting.

Lightweight MRCs can also guide efficient cache sizing, allocation, and partitioning for both individual workloads and complex multi-workload environments. For example, Talus [1], which requires an MRC as input, can remove performance cliffs within a single workload and improve cache partitioning across workloads.

Adapting Cache Parameters

As illustrated in Figure 5, our multi-model optimization framework leverages miniature simulations to evaluate the impact of different candidate parameter values. The best setting is applied to the actual cache periodically. We have implemented optimizations that adapt tunable parameters automatically for two well-known cache policies, LIRS [4] and 2Q [5], but we discuss only the LIRS results; the results for 2Q are similar [8].

While MRCs are typically stable over short time periods, they frequently vary over longer intervals. To adapt dynamically to changing workload behavior, we divide the input reference stream into a series of epochs. Our experiments use epochs consisting of one million references, although many alternative definitions based on wall-clock time, evictions, or other metrics are possible.

After each epoch, we calculate an exponentially weighted moving average (EWMA) of the miss ratio for each mini-cache to balance historical and current cache behavior. Our experiments use an EWMA weight of 0.2 for the current epoch. The parameter value associated with the mini-cache exhibiting the lowest smoothed miss ratio is applied to the actual cache for the next epoch.

LIRS Adaptation

We adapt the size of the LIRS S stack, which controls the number of metadata-only ghost entries that are tracked [4], by setting f, a parameter that specifies the size of S as a fraction of the over-

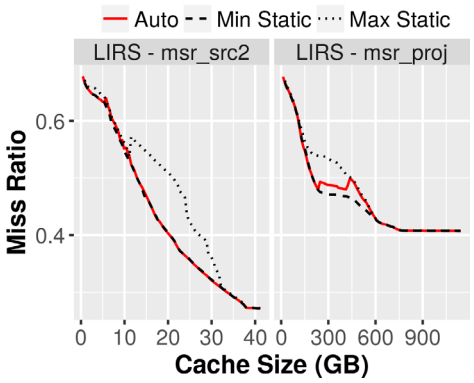


Figure 6: Adaptive Parameter Tuning. Dynamic optimization selects good values for the LIRS *f* parameter at most sizes with potential gains. The `msr_src2` and `msr_proj` workloads show the best- and worst-case results for the MSR traces.

all cache [8]. For each workload, five scaled-down simulations are performed with different values for *f*: 1.1, 1.5, 2.0, 2.5, and 3.0. Each simulation emulates the same cache size, equal to the size of the actual cache, with a fixed sampling rate $R = 0.005$. After each epoch consisting of one million references, the miss ratios for each mini-cache are examined, and the best *f* value is applied to the actual cache.

Figure 6 presents the best-case and worst-case results across the 12 MSR traces. The goal of automatic LIRS adaptation is to find the best value of *f* for each cache size. These ideal static settings form an MRC that traces the lower envelope of the curves for different static *f* values, plotted as the dashed curve. Similarly, the dotted curve shows the MRC with the pessimal static *f* setting at each cache size. The auto-adapted results for `msr_src2` hug the ideal lower envelope closely at nearly all cache sizes. In contrast, `msr_proj` deviates from the ideal for many cache sizes, but still does well for others. We are currently experimenting with techniques that automatically disable adaptation when it is ineffective.

SLIDE

SLIDE (*S*harded *L*ist with *I*nternal *D*ifferential *E*viction) is a completely different cache optimization technique that leverages scaled-down MRCs constructed by running miniature simulations. SLIDE was inspired by Talus [1], a powerful technique introduced in the computer architecture community for set-associative processor caches. Talus removes performance cliffs using interpolation to effectively operate at a point on the convex hull of an MRC—the shape formed by stretching a rubber band across the bottom of the curve. In the presence of cliffs, the large gap between an MRC and its convex hull represents a significant optimization opportunity.

Talus uses hash-based partitioning to divide the reference stream for a single workload into two shadow partitions, *alpha* and *beta*, which operate as separate sub-caches. Each partition is made to emulate the performance of a smaller or larger cache by controlling its size and its input load, represented by the fraction of the reference stream it receives. Talus requires the workload's MRC as an input and computes the partition sizes and their respective loads in a clever manner that ensures their combined aggregate miss ratio lies on the convex hull of the MRC. We view the hash-based partitioning employed by Talus for optimization and our hash-based monitoring for efficient modeling as two sides of the same coin. Both rely on the property that hash-based sampling produces a smaller reference stream that is statistically self-similar to the original stream.

One key challenge with applying Talus to non-stack algorithms is constructing MRCs efficiently at runtime. Fortunately, scaled-down models provide a convenient solution. As with parameter adaptation, we divide the input reference stream into a series of epochs. After each epoch, we construct a discretized MRC from multiple scaled-down simulations with different cache sizes, smoothing each miss ratio using an EWMA. We then identify the subset that forms the convex hull for the MRC, and compute the optimal partition sizes and loads using the same inexpensive method as Talus.

Non-LRU Shadow Partitioning Challenges

In theory, combining scaled-down MRCs with Talus shadow partitioning can improve the performance of *any* caching policy by interpolating efficient operating points on the convex hulls of workload MRCs. In practice, it was much more difficult than we expected to apply Talus to caching algorithms such as ARC and LIRS.

Talus requires distinct cache instances for its alpha and beta partitions, which have a fixed aggregate size. This hard division becomes problematic in systems where partition boundaries change dynamically as MRCs evolve over time. Similarly, when per-partition input loads change dynamically, some cache entries may reside in the "wrong" partition based on their hash values.

Eager strategies, such as removing cache entries when decreasing the size of a partition or migrating entries across partitions to ensure each resides in the correct partition, perform poorly since migration is expensive and data may be evicted from one partition before the other needs the space. Moreover, it's not clear how migrated state should be integrated into its new partition, since list positions are not ordered across partitions.

Lazy strategies for reallocation and migration fare better but complicate the core caching logic. More importantly, while migrating to the MRU position on a hit seems reasonable for an

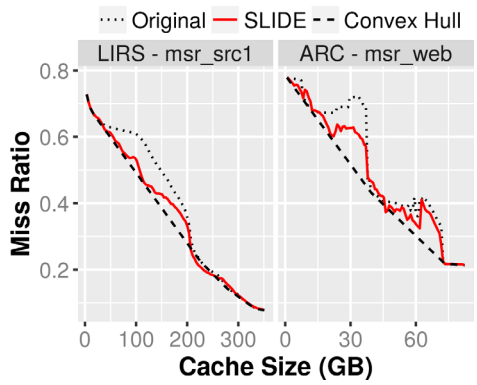


Figure 7: SLIDE Cliff Reduction. Scaled-down MRCs are constructed dynamically from seven mini-cache simulations. SLIDE improves miss ratios for LIRS and ARC at most sizes with potential gains but does exhibit some regressions.

LRU policy, it's not clear how to merge state appropriately for more general algorithms.

Transparent Shadow Partitioning

Faced with these challenges, we developed SLIDE. In contrast to Talus, SLIDE maintains a single unified cache and defers partitioning decisions until eviction time, conveniently avoiding the resizing, migration, and complexity issues discussed above.

A SLIDE list is a new abstraction that serves as a drop-in replacement for the standard LRU list used as a common building block by many sophisticated algorithms. Since SLIDE interposes on primitive LRU operations that add, reference, and evict entries, it is transparent to cache replacement decisions. An unmodified algorithm can support Talus-like partitioning by simply relinking to substitute SLIDE lists for existing ones. We have optimized ARC (*T1*, *T2*, *B1*, and *B2*), LIRS (*S* and *Q*), 2Q (*Am*, *A1in*, and *A1out*), and LRU in this manner.

SLIDE extends a conventional doubly linked LRU list, which remains totally ordered from MRU (head) to LRU (tail). Each entry is augmented with a compact hash of its key, which is compared to a current threshold that dynamically classifies it as belonging to either the alpha or beta "partition." Additional state supports efficient SLIDE versions of all list operations [8]. SLIDE preferentially evicts from the tail of the over-quota partition.

It is not obvious that substituting SLIDE lists for internal lists will approximate Talus partitions. The basic intuition is that configuring each internal list with identical SLIDE partition sizes and input loads effectively divides the occupancy of each individual list—and therefore the entire aggregate algorithm state—to achieve the desired split between alpha and beta. While SLIDE may differ from strict Talus partitioning, it empirically works well for ARC, LIRS, 2Q, and LRU.

Experiments

For each workload, a separate experiment is performed at 100 cache sizes. For each size, a discrete MRC is constructed via multiple scaled-down simulations with sampling rate $R = 0.005$. SLIDE is reconfigured after each one million-reference epoch, using an EWMA weight of 0.2.

Seven emulated cache sizes are positioned exponentially around the actual size, using relative scaling factors of 1/8, 1/4, 1/2, 1, 2, 4, and 8. For $R = 0.005$, the mini-cache metadata is approximately 8% of the actual metadata size (R times the sum of the scaling factors), representing less than 0.04% of total memory consumption for an actual cache. Alternative configurations provide different tradeoffs between time, space, and accuracy.

Figure 7 plots some example results of SLIDE performance cliff reduction for LIRS and ARC policies with workloads that exhibit cliffs. Ideally, SLIDE would trace the convex hull of the original MRC. In practice, this is not attainable, since the MRC evolves dynamically, and its few discrete points yield a crude convex hull. Nevertheless, for these examples, SLIDE captures a significant fraction of the potential gain, represented by the area between the MRC and its convex hull: 69% for LIRS and 38% for ARC. For workloads with MRCs that are already mostly convex, there is little opportunity for improvement, so SLIDE typically yields marginal benefits.

Conclusion

We have explored the use of miniature caches for modeling and optimizing cache performance. Compelling experimental results demonstrate that scaled-down simulation works extremely well for a diverse collection of complex caching algorithms—including ARC, LIRS, and OPT—across a wide range of real-world traces. This suggests our technique is a robust method capable of modeling nearly any cache policy accurately and efficiently.

Lightweight modeling has many applications, including online analysis and control. We presented a general method that runs scaled-down simulations to evaluate hypothetical configurations, and applied it to optimize tunable cache policy parameters automatically. We also introduced SLIDE, a new transparent technique that performs Talus-like performance cliff removal.

Miniature caches offer the tantalizing possibility of improving performance for most caching algorithms on most workloads automatically. We hope to make additional progress in this direction by exploring opportunities to refine and extend our optimization techniques.

References

[1] N. Beckmann and D. Sanchez, "Talus: A Simple Way to Remove Cliffs in Cache Performance," in *Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA-21)* (February 2015): https://people.csail.mit.edu/sanchez/papers/2015.talus.hpca.pdf.

[2] L. A. Belady, "A Study of Replacement Algorithms for Virtual Storage Computers," *IBM Systems Journal,* vol. 5, no. 2 (1966), pp. 78–101: http://users.informatik.uni-halle.de/~hinnebur/Lehre/Web_DBIIb/uebung3_belady_opt_buffer.pdf.

[3] X. Hu, X. Wang, L. Zhou, Y. Luo, C. Ding, and Z. Wang, "Kinetic Modeling of Data Eviction in Cache," in *Proceedings of the 2016 USENIX Annual Technical Conference (ATC '16)*, pp. 351–364: https://www.usenix.org/conference/atc16/technical-sessions/presentation/hu.

[4] S. Jiang and X. Zhang, "LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance," in *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '02)*, pp. 31–42: http://web.cse.ohio-state.edu/hpcs/WWW/HTML/publications/papers/TR-02-6.pdf.

[5] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," in *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*, pp. 439–450: http://www.vldb.org/conf/1994/P439.PDF.

[6] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal*, vol. 9, no. 2 (June 1970), pp. 78–117.

[7] N. Megiddo and D. S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, pp. 115–130: http://www2.cs.uh.edu/~paris/7360/PAPERS03/arcfast.pdf.

[8] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park, "Cache Modeling and Optimization Using Miniature Simulations," in *2017 USENIX Annual Technical Conference (ATC '17)*, pp. 487–498: https://www.usenix.org/system/files/conference/atc17/atc17-waldspurger.pdf.

[9] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad, "Efficient MRC Construction with SHARDS," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, pp. 95–110: https://www.usenix.org/system/files/conference/fast15/fast15-paper-waldspurger.pdf.

Using gRPC with Go

CHRIS "MAC" MCENIRY

Chris "Mac" McEniry is a practicing sysadmin responsible for running a large e-commerce and gaming service. He's been working and developing in an operational capacity for 15 years. In his free time, he builds tools and thinks about efficiency. cmceniry@mit.edu

In the past few articles, we've used Go's `net/rpc` library to build a simple file metadata server. In this article, we're going to look at using gRPC (https://grpc.io) to fulfill the same purpose.

gRPC has many advantages over the built-in RPC library, namely:

- Fast and efficient network communication
- Ability to stream inputs and outputs
- Automatic transport encryption
- Ability to interact with other languages
- Ready extensions to support authentication and connection handling.

gRPC is typically boiled down to the description "Protobuf messages over HTTP/2." This is true to a first pass, but it also encompasses the libraries, middleware extensions, and interactions with other languages.

For the sake of brevity, some sections of the code examples here are left out. The full code for this example can be found at https://github.com/cmceniry/login-grpcls.

Getting Started

Our service building story goes a little something like this:

- First, we have to get the `protobuf` tools and gRPC Golang libraries.
- Next, we'll create a language-independent `protobuf` definition for our service.
- With the `protobuf` definition in hand, we'll generate Golang hooks.
- After that, we can fill in our interactions with those hooks.
- And finally, we can compile and run.

Getting the Tools

Building gRPC applications requires a couple of tools. The first is `protoc` which is the language-independent protobuf compiler. The second is the `protoc-gen-go` plugin which is used to generate Golang code for the data and interface types. In addition, we're going to need the Golang `grpc` library.

The installation for `protoc` depends on your platform. It is packaged up for various platforms (rpm, brew, etc.), but when in doubt you can get it from the official release location: https://github.com/google/protobuf/releases.

For the Golang-specific items, you can grab these with the `go get` command.

```
go get -u google.golang.org/grpc
go get -u github.com/golang/protobuf/protoc-gen-go
```

The former is a library, and it is possible to alternatively vendor it and manage it with dep. The latter produces the protoc-gen-go executable, so it needs to be installed in such a way that that is available, typically with a go get into the home workspace. However you choose to install it, you will want to make sure that its location is in your PATH since protoc will look there for it. For the above, you can use

```
export PATH=${PATH}:~/go/bin
```

Now that we have a good environment, we can move on to working on the code. If you're following the go get commands from above, you can grab the example code and change into that directory now. This is dependent on the TLS credentials generated by the previous login-glss, so we'll need to get that and create the certificates first.

```
go get -u github.com/cmceniry/login-glss
go get -u github.com/cmceniry/login-grpcls
cd ~/go/src/github.com/cmceniry/login-glss
go run certs/generate_certs.go
cd ../login-grpcls
```

The Protobuf Definition File

Like any protobuf protocol, gRPC starts with a .proto file. On a first pass, this is a simplified description of the messages that will be transported over the connection. Specifically for gRPC (well, RPCs in general, but other implementations are rare), it is also a description of the services and their interfaces, which use these messages.

To determine what should be in our .proto file, we need to think about what we're passing back and forth between the client and the server. To mirror glss, we will want to pass from the client to the server the Path that we're going to be using. From the server to the client, the resulting File information blocks for the client-supplied path. In addition, we want an RPC to call LS. The RPC semantics in .proto also define a service, Lister, which encapsulates several RPCs and properties.

So we'll want to define the following four items in our .proto: Path, File, LS, and Lister, for which we'll need a bit of boilerplate. We are telling protobuf which version we'll be using, and we need to wrap our collection of services and messages into a package.

```
syntax = "proto3";
package directorycontents;
```

Next, we need to define what will be transporting over our connections: Path and File. We will structure these as message items that will be used in our remote calls. message is the generic type for data passing between the client and server, regardless of whether it is a parameter or return value. Each message is a combination of a message type name and specific typed fields that are of meaning in that message.

First, we'll tackle Path, which just has a single field in it, string name.

```
message Path {
  string name = 1;
}
```

The 1 associated with name is a field numeric ID to allow for compatibility between clients and servers of different versions. This allows for nonbreaking changes to the API without having to upgrade *every* client and server out there. You can enable new fields by appending to the end with a new number. You can change existing fields by adding a new field with the appropriate changes for the old field. You will end up populating both fields for a period of time, but it does allow for newer servers and clients to speak to both current and old versions of themselves.

Next, we'll build out our File response. It also has a string name, as well as size, mode, and modtime:

```
message File {
  string name = 1;
  int64 size = 2;
  string mode = 3;
  string modtime = 4;
}
```

Now that we have our two message definitions, we can move on to our service definition. Since we're providing a generalized directory lister service, we're going to call this Lister. As mentioned, this will contain our collection of RPC calls (in this case, it's just one). Each RPC has a list of inputs and outputs (in this case, it's just Path and File).

```
service Lister {
  rpc LS (Path) returns (stream File) {}
}
```

As mentioned in the introduction, gRPC allows us to stream inputs and outputs. In this case, we're going to call LS with a single input item Path, but we're going to get back a large list of File blocks. For efficiency and demonstrative purposes, we're going to stream the File responses—hence the stream modifier in the LS return values. We could have wrapped them all up in an array, but this way we don't need to maintain all of that in memory as we go through it. As we'll see in the application code, once a file is found, it can immediately be sent back to the client.

Generating gRPC Code

Now that we have the data types and function-call semantics, we want to put this language-independent form into something that

we can use in Golang. This is where `protoc` and `protoc-gen-go` come into play. From the root of our project:

```
protoc --go_out=plugins=grpc:. \
        directorycontents/directorycontents.proto
```

This invokes `protoc` and tells it to use the `proto-gen-go` with the gRPC plugin to process our `.proto` file. This will produce `directorycontents/directorycontents.pb.go`.

The full file is a bit much to go over in this article, but its key contributions are:

- It defines Go native structs for `Path` and `File`:

```
type Path struct {
        Name string protobuf:"bytes,1,opt,name=name"
json:"name,omitempty"
}
...
type File struct {
        Name    string protobuf:"bytes,1,opt,name=name"
json:"name,omitempty"
        Size    int64  protobuf:"varint,2,opt,name=size"
json:"size,omitempty"
        Mode    string protobuf:"bytes,3,opt,name=mode"
json:"mode,omitempty"
        Modtime string protobuf:"bytes,4,opt,name=
        modtime"
json:"modtime,omitempty"
}
```

- These each have some accessor functions to them, or you can manipulate the struct field directly.
- It defines a `ListerClient` interface (with accompanying constructor func). This interface is how we're going to call the gRPC functions from our code. Specifically, we're going to be calling the `LS` function on the return `ListerClient` interface.

```
type ListerClient interface {
        LS(ctx context.Context, in *Path, opts
...grpc.CallOption) (Lister_LSClient, error)
}
...
func NewListerClient(cc *grpc.ClientConn) ListerClient {
```

- It defines the `ListerServer` interface for the server side. We're going to build a struct that implements this interface as our way of responding to gRPC calls. Related to this, it defines the `Lister_LSServer` interface that is used specifically for our outlet to send responses to the `LS` calls.

```
type ListerServer interface {
        LS(*Path, Lister_LSServer) error
}
...
type Lister_LSServer interface {
        Send(*File) error
...
```

The Server Implementation

Now it's time to focus on our application code, starting with the server side. gRPC has presented us with an interface that we need to implement. As mentioned in the last section, we're going to implement the `ListerServer`.

Since this is Golang code, let's take care of the imports. Common practice is to import the generated `.proto` definitions with the alias name of `pb`. In addition, we're going to import the `grpc` library itself, and the `grpc/reflection` library to support API information sharing.

```
import (
    pb "github.com/cmceniry/login-grpcls/directorycontents"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials"
    "google.golang.org/grpc/reflection"
```

Next, we'll divide the server into two parts. The first is the actual remote procedure to be called `LS`. The second is wiring up everything.

Since `ListerServer` is an interface, we need to set up two parts to it: a struct and the supporting member methods.

```
type server struct{}

func (s *server) LS(p *pb.Path, fileInfoStream
pb.Lister_LSServer) error {
```

You can wrap this up with much more, but for this example, our struct is as simple as can be. The `func` signature for `LS` must match the one from `directorycontents.pb.go`. Note that the gRPC response values are not a part of the return values from the function. Since we're going to stream the results back, we will be working with the `fileInfoStream` value as our conduit to send the data back while inside of our function.

The remainder of the func uses `filepath.Walk` just as the original gls and glss servers did. It only has modifications to handle sending the data directly back on the `fileInfoStream`.

```
        err := filepath.Walk(p.Name, func(path string, info
os.FileInfo, err error) error {
```

```
f := &pb.File{
    Name:    info.Name(),
    Size:    info.Size(),
    Mode:    info.Mode().String(),
    Modtime: info.ModTime().Format("Jan _2 15:04"),
}
err = fileInfoStream.Send(f)
```

In this, we're converting every `os.FileInfo` we see as we receive it. For it to go over the gRPC connection, it has to be in the form of the messages from the `.proto`. Here, we convert it to `pb.File`, which allows us to send it back using `fileInfoStream.Send`. To reiterate, this is happening as we see every file, so we don't have to construct any intermediate arrays before sending them all back.

Finally, on the server, we need to wire the network level up to our `ListerServer`. We'll start with a standard TCP network listener.

```
func main() {
    l, err := net.Listen("tcp", ":4270")
```

Since we want to enable TLS authentication, we need to prepare that. The first part is to load in the certificates and keys. This is identical to the way that we loaded them in `login-glss`.

```
certificate, err := tls.LoadX509KeyPair(
    "../login-glss/certs/server.crt",
    "../login-glss/certs/server.key",
)
if err != nil {
    log.Fatalf("could not load client key pair: %s", err)
}
caCert, err := ioutil.ReadFile("../login-glss/certs/CA.crt")
if err != nil {
    log.Fatal(err)
}
caCertPool := x509.NewCertPool()
caCertPool.AppendCertsFromPEM(caCert)
```

With the credentials loaded, we need to format these for gRPC to use. This involves wrapping a typical `tls.Config` struct with a `grpc.credentials` struct. It's the latter that is used by the gRPC services. Much like the glss server, we need to provide our certificate and configure the pool and flag for client auth. In addition, we need to ensure that our expected TLS ServerName is supplied so that the client can validate against that.

```
creds := credentials.NewTLS(&tls.Config{
    ServerName:   "localhost",
    Certificates: []tls.Certificate{certificate},
    ClientCAs:    caCertPool,
    ClientAuth:   tls.RequireAndVerifyClientCert,
})
```

Next, we create a `grpc.Server` struct, then register a ListerServer with it. When creating the `grpc.Server`, we indicate that we're using the TLS configuration that we just set up.

```
s := grpc.NewServer(grpc.Creds(creds))
pb.RegisterListerServer(s, &server{})
```

Next, we enable information on the service via the `grpc/reflection` library. This is an optional step that allows generic gRPC clients to interact with our `Lister` service. You can inspect the information exposed using the gRPC command line tool found at https://github.com/grpc/grpc/blob/master/doc/command_line_tool.md.

```
reflection.Register(s)
```

Finally, we can start the `grpc.Server` by telling it to act on our `tcp.Listener`.

```
err = s.Serve(l)
```

If all has gone well, we've successfully wired our server together. Now, on to the client side. Other than the import, it consists strictly of a `main` func. Like glss, it takes a single command line argument, which is the directory to get the listing.

The Client Implementation

The new imports for the client all involve the gRPC libraries. The client refers to the same `.proto`-generated definitions as the server, so it will need to import them as well. And, obviously, it needs to import the `grpc` library.

```
import (
    pb "github.com/cmceniry/login-grpcls/directorycontents"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials"
```

As the first part of our main function, we need to load our TLS values.

Again, this is identical to how it is configured in glss.

```
certificate, err := tls.LoadX509KeyPair(
    "../login-glss/certs/client.crt",
    "../login-glss/certs/client.key",
)
if err != nil {
    log.Fatalf("could not load client key pair: %s", err)
}
caCert, err := ioutil.ReadFile("../login-glss/certs/CA.crt")
if err != nil {
```

```
        log.Fatal(err)
    }
    caCertPool := x509.NewCertPool()
    caCertPool.AppendCertsFromPEM(caCert)
```

As with the server side, we need to wrap these as `grpc.credentials`. Note that our TLS config only requires that we supply our certificate and provide a RootCA pool against which to validate the server.

```
    creds := credentials.NewTLS(&tls.Config{
        Certificates: []tls.Certificate{certificate},
        RootCAs:      caCertPool,
    })
```

Next, we form our network connection. Unlike on the server side, the `grpc` library has its own Dialer for the client side. We need to supply this with the endpoint to connect to and our general gRPC configuration—in this case, our credentials configuration.

```
    conn, err := grpc.Dial("localhost:4270",
grpc.WithTransportCredentials(creds))
```

At this point, we've only established a general gRPC connection. There is nothing specific about our particular API, so we need to remedy that. We accomplish this by wrapping the general gRPC connection with a client that is specific to our Lister service.

```
    c := pb.NewListerClient(conn)
```

Now we can make our RPC call. This uses the context library for handling timeouts and cancellations. In this example, we're just going to use the `context.Background()`, so we're skipping over additional `context` library handling of timeouts and cancellations. Our actual argument to `LS` is the `pb.Path` wrapped value from the command line.

```
    files, err := c.LS(context.Background(), &pb.Path{Name:
os.Args[1]})
```

Since the return value from `LS` is a `protobuf` stream, we need to read each value from it. We do this by looping around `files.Recv()`. If the stream is complete, `LS` returns the `io.EOF` sentry error and allows us to break out of the loop. Otherwise, unless there's an error, we print out of the file information.

```
    for {
        f, err := files.Recv()
        if err == io.EOF {
            break
        }
        if err != nil {
            log.Fatalf("LS file failure: %s", err)
        }
            fmt.Printf("%s %10d %s %s\n", f.Mode, f.Size,
f.Modtime, f.Name)
    }
```

Running It All

Now that we have the client and server, we can run it all together.

```
login-grpcls$ go run server/main.go &
[1] 11488
login-grpcls$ 2017/12/16 21:44:17 Starting server
login-grpcls$ go run client/main.go .
drwxr-xr-x       204 Dec  6 21:27 .
drwxr-xr-x       102 Dec 16 11:43 client
drwxr-xr-x       136 Dec 13 19:16 directorycontents
-rw-r--r--       267 Dec  6 21:27 links
drwxr-xr-x       102 Dec 16 11:32 server
```

Conclusion

This article has provided a brief introduction to using gRPC with Golang. In addition, this series of articles has given us two implementations for our `LS` service—one using the `net/rpc` from Golang, and one using gRPC. I hope that you now feel comfortable enough to consider using gRPC in your work and, more importantly, to be able to weigh the pros and cons of when to use it or `net/rpc` as appropriate for your situation.

A few specific similarities and differences to remember between the two:

- Both setups involve configuring a generalized RPC server and then registering calls to it.
- Outside of some wrapping, both interact with TLS in the same way. The underlying implementation at the TLS layer is the same. Given the end-to-end principle, it should not be surprising to see the same behavior from a wrapping layer.
- With `net/rpc`, we're handling Golang data structures. With `grpc`, we're handling more generic data structures (which can be referenced by multiple languages). The `net/rpc` way is easier to handle in Golang but does limit the interaction to Golang. Which one you should use depends on the users of your API and the contract you need or want to maintain.
- While we did not demonstrate it in this example, gRPC has several middleware wrappers. These provide higher-level API enrichments to help enable resiliency and visibility. Since there are interface patterns, there is the possibility that the same exists for `net/rpc`, but its goal has been to be a solid simple standard library. It's unlikely that these will exist for `net/rpc`.

Happy Going!

Practical Perl Tools
Top of the Charts

DAVID N. BLANK-EDELMAN

David has over 30 years of experience in the systems administration/DevOps/SRE field in large multiplatform environments and is the author of the O'Reilly Otter book (new book on SRE forthcoming!). He is one of the co-founders of the now global set of SREcon conferences. David is honored to serve on the USENIX Board of Directors where he helps to organize and engineer conferences like LISA and SREcon. dnb@usenix.org

I sometimes wonder if the people who make statements about Perl's health in the world (some nostalgic, some a little more mean-spirited) have a sense of just how vibrant the Perl world is. I wonder whether seeing some of the interesting things being developed even as we speak or the range of projects available would change their thinking.

This leads to a good question: how do you find out about the interesting things happening in Perl on a week-to-week basis? In this column I'd like to focus on one of the answers to that question: the weekly reports that are published about modules.

We'll look at three of them and for fun pick and consider interesting modules from each. All three of these listings are published each week on a blog created by Spanish Perl hacker Miguel Prz ("NICEPERL"), which can be found at http://niceperl.blogspot.com. Before we dive in, I should mention that these listings came to my attention thanks to the lovely newsletter started by Gabor Szabo called *Perl Weekly*. You can sign up and find past issues at http://perlweekly.com.

CPAN Great Modules Released Last Week

The first list we are going to look at is indeed titled "CPAN great modules released last week." This is an ordered list of modules newly published to the Comprehensive Perl Archive Network which has garnered 12 or more "favorite" ratings—that is, 12 or more people "++"d these modules on the MetaCPAN (metacpan.org) listing site.

On the off chance you are brand new to the Perl world, let me quickly mention that CPAN is an archive where people in the Perl community share their modules and other Perl work for everyone to use. It is one of the best things Perl has going for it. And to be totally candid, it is not always the best; some of the code there isn't the greatest. To give you a sense of its scale, here are the stats as of today from the cpan.org home page:

```
The Comprehensive Perl Archive Network (CPAN) currently has 194,457 Perl modules
in 35,953 distributions, written by 13,329 authors, mirrored on 256 servers.
The archive has been online since October 1995 and is constantly growing.
```

So what's in the list of great modules for the week of December 17, 2017, the one in which I am writing?

This week we find a few old friends of the column like Mojolicious and perlbrew. Instead of retreading, let's instead look at DBIx::Simple. In the past, we've talked a bit about the framework that was a true innovator in the space at the time it was introduced: DBI. The idea of having a single portable abstraction layer for code that communicated with databases independent of the database backend being used was a great step forward at the time. This idea was subsequently expanded in many different directions (and the basic concept was repurposed for other non-database contexts as well). My magic 8-ball predicts an entire column on DBI-related expansions coming in our near future... But in the meantime, let's look at DBIx::Simple.

#	CPAN module	Version	Votes	Abstract
1	App::perlbrew	0.81	149	App::perlbrew - Manage Perl installations in your $HOME
2	Catalyst::Action::REST	1.21	16	Automated REST method dispatching
3	Data::Alias	1.21	12	Comprehensive set of aliasing operations
4	DBIx::Simple	1.37	27	Very complete easy-to-use OO interface to DBI
5	Digest::SHA	6.00	19	Perl extension for SHA-1/224/256/384/512
6	experimental	0.019	29	Experimental features made easy
7	libwww::perl	6.30	135	The World Wide Web library for Perl
8	Math::Prime::Util	0.70	12	Utilities related to prime numbers, including fast sieves and factoring
9	Mojolicious	7.58	352	Real-time web framework
10	SQL::Translator	0.11023	32	SQL DDL transformations and more
11	Test::Class::Moose	0.91	14	Serious testing for serious Perl
12	Text::Xslate	v3.5.3	58	Scalable template engine for Perl5

Table 1: This is what the table at https://niceperl.blogspot.com/2017/12/clxii-cpan-great-modules-released-last.html looked like on December 17, 2017.

Standard DBI has you writing code that looks like this (examples excerpted from the DBI doc with my comments inserted):

```
use DBI;

# connect to a database
$dbh = DBI->connect($data_source, $username, $auth, \%attr);

# execute a random SQL statement
$rv  = $dbh->do($statement);

# various ways of retrieving the results
$ary_ref  = $dbh->selectall_arrayref($statement);
$ary_ref  = $dbh->selectcol_arrayref($statement);
@row_ary  = $dbh->selectrow_array($statement);
$ary_ref  = $dbh->selectrow_arrayref($statement);
$hash_ref = $dbh->selectrow_hashref($statement);

# more efficient ways of running/rerunning queries
$sth = $dbh->prepare($statement);
$rv  = $sth->execute;

# other ways of retrieving results
@row_ary  = $sth->fetchrow_array;
$ary_ref  = $sth->fetchrow_arrayref;
$hash_ref = $sth->fetchrow_hashref;

# close connection to the database
$rc  = $dbh->disconnect;
```

These are some of the more commonly used statements when working with DBI, certainly when first getting started. It's worth reading the entire doc (several times) to get a good handle on the proper idioms and performant ways to work with DBI. And, hoo boy, is there plenty of doc to read—124 pages if you were to print it all out as of the time of this writing.

DBIx::Simple aims to, well, you probably guessed it, make some of the coding with DBI more simple. With DBIx::Simple, you write code that looks almost identical to plain DBI:

```
use DBIx::Simple;

my $db = DBIx::Simple->connect(
   DBI:mysql:database=test,    # DBI source specification
   test, test,              # Username and password
   { RaiseError => 1 }         # Additional options
);
```

but then you can write code of this form (as stated in the doc):

```
$db->query($query, @variables)->what_you_want;
```

Some examples of this from the doc would be:

```
my ($name, $email) = $db->query(
   SELECT name, email FROM people WHERE email = ? LIMIT 1,
   $mail
)->list;
```

Here we're querying the people table for a list of two fields—name and email—given the email address (`$mail`). We ask for the information back as a list.

If we didn't want to chain methods like that, we could write:

```
$result = $db->query(...)
```

and then work from `$result` object returned using methods like:

```
@columns = $result->columns
```

or

```
@row  = $result->list     # return as a list
@rows = $result->flat     # return as a flattened list
$row  = $result->array    # return as an array ref
@rows = $result->arrays   # return as an array of arrays
$row  = $result->hash     # return as a hash
@rows = $result->hashes   # return as an array of hashes
```

Here's an example of a query that returns a set of rows which we then iterate over to print separate fields:

```
for my $row (
  $db->query(SELECT name, email FROM people)->hashes) {
    print "Name: $row->{name}, Email: $row->{email}\n";
}
```

I like how legible something like `$row->{fieldname}` is in the code.

DBIx::Simple also hooks into a few other modules (some of which we'll likely talk about soon). For example, if you have Text::Table installed, then code like:

```
print $result->text("box")
```

makes pretty output like:

```
+----+--------+----------+
| ID | Animal | Type     |
+----+--------+----------+
|  1 | camel  | mammal   |
|  2 | llama  | mammal   |
|  3 | owl    | bird     |
+----+--------+----------+
```

MetaCPAN Weekly Report

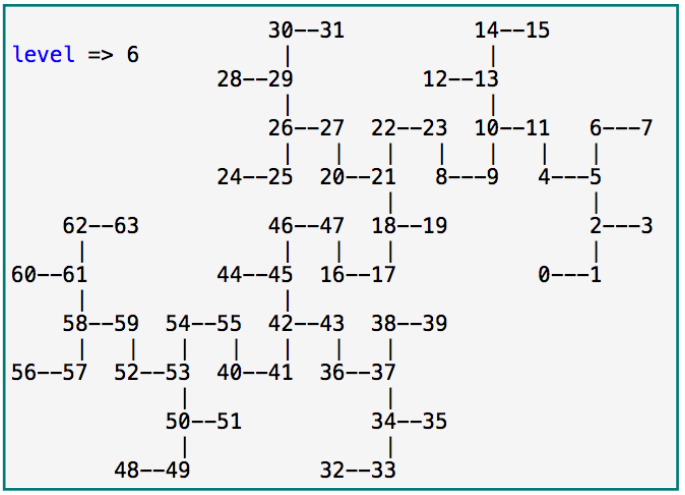
The second report to mention is also from data pulled from the MetaCPAN site. This report lists both the newly favorited modules ("Clicked for the first time") and those whose popularity is on the rise ("Increasing its reputation"). If enough people vote for a particular module, this report will call out that module as "special," but that did not happen this week.

You can find an example of the table at https://niceperl.blogspot.com/2017/12/ccxciv-metacpan-weekly-report.html.

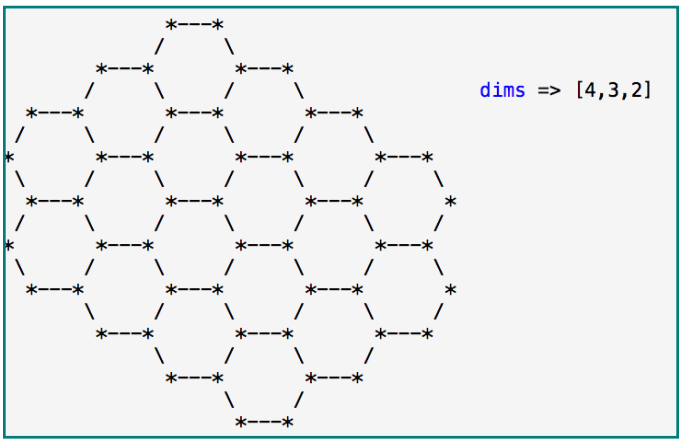
In the "first click" section, I found a couple of different modules to be interesting, not because they helped me discover a module I might use, but because they offered solutions for problems in spaces that I knew very little about. It can be fun to have something like this shoot you off on tangents (not to mention build your procrastination muscles). For example, I had never heard of a Confusion Matrix until I met AI::ConfusionMatrix. In case you are curious, Wikipedia defines them as follows:

> In the field of machine learning and specifically the problem of statistical classification, a confusion matrix, also known as an error matrix, is a specific table layout that allows visualization of the performance of an algorithm, typically a supervised learning one (in unsupervised learning it is usually called a matching matrix). Each row of the matrix represents the instances in a predicted class while each column represents the instances in an actual class (or vice versa). The name stems from the fact that it makes it easy to see if the system is confusing two classes (i.e. commonly mislabeling one as another).

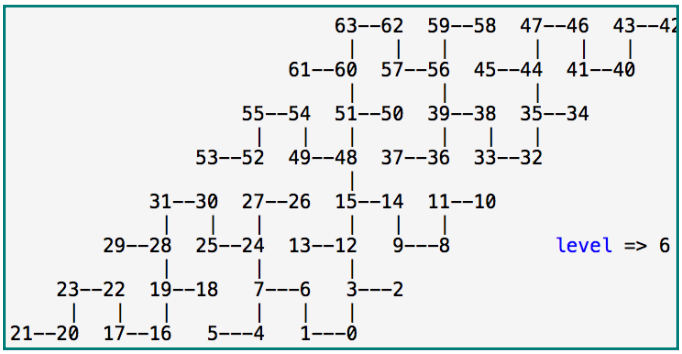
Similarly, I realized how woefully inadequate my understanding of graph theory was when I encountered Graph::Maker::Other. This appears to be a collection of modules for making graphs like Beineke, bi-star, quartet tree, twindragon area tree, and others I hadn't heard of. Some are quite pretty—for example, that last one:



or hexgrid:



or twin alternate area tree:



The documentation in this module distribution also references a website called House of Graphs (https://hog.grinvin.org), which is a "Database of interesting graphs." And there goes that afternoon...

To pull things back to the world of Perl, I'd like to highlight the useful module File::HomeDir::Tiny, which describes itself as follows:

> This module is useful for the 90% of the time that you only need 10% of File::HomeDir's functionality. It depends on no other modules and consists of just fourteen lines of code.

so:

```
use File::HomeDir::Tiny;
$home = home;
```

Nothing complex, but highly useful. And if you want to have a quick party trick up your sleeve (presuming you go to the right parties) for when someone asks you about how nuts the Perl punctuation can be, check out the part of the docs that begins with:

> If your code is only going to run on Unix, you do not need to bother with any module. Just use the alien spaceship operator:

```
($home) = <~> ;
```

From the "Increasing its reputation" section of this report, there's a whole bunch of interesting modules to look at. There's Damien Conway's PPR (Pattern-based Perl Recognizer), which scares the pants off of me. It is basically a distribution of Perl regular expressions designed to match certain Perl constructs in a Perl program. Unlike PPI (which we looked at in a recent column), which actually parses Perl, this just allows you to easily say, "Is there a comment in this Perl code?" or give me back the code without the comment, the same way you might strip anything else out of text using a regular expression. I was too scared to look at the actual source code for this module.

Other interesting stuff:

- App::tt, a time tracking module/app. This is the sort of thing that people who have to keep track of their time spent on individual activities or projects would love. From the command line, you can say "started" and then later "finished." It can also do spiffy things like look at a directory with a Git repo in it and use the `reflog` there to automatically determine working times.
- App::RoboBot, an "event-driven, multi-protocol, multi-network, user-programmable, plugin-based, S-Expression chatbot. Any text-based chat service could be supported, with plugins currently for IRC, Slack, and Mattermost included." More info can be found at: https://robobot.automatomatromaton.com. Yup, there goes another afternoon.
- Promises, an implementation of Promises in Perl. If you ever get the chance to venture out into the wider programming world, especially in the world of JavaScript and the jQuery library, you may encounter a programming construct called "Promises." Promises are an attempt to deal with the complexity of writing reasonably sized asynchronous programs based on callbacks. At a certain point, those programs devolve into what people call "callback hell" because they invariably start to have callbacks calling other callbacks. If the forest of callbacks gets too thick or self-referential, it becomes very hard to debug or even to understand how the program will behave. As the doc says, "Promises give us back a top-to-bottom coding style, making async code easier to manage and understand. It looks like synchronous code, but execution is asynchronous." If you find yourself writing even medium-sized programs that are event/callback based, it would probably behoove you to check out the world of Promises to see how others are coping with the complexity you are sure to encounter. The external references in the docs are also well worth taking some time to go read.

StackOverflow Perl Report

Okay, last report. This one is less useful for finding cool modules or strange afternoon-wasting tangents and more helpful for keeping your head in the Perl game and refreshing your Perl knowledge. This section lists the top 10 rated Perl questions on StackOverflow. It also lists the number of answers provided for each. I find it useful to just scan the list each week to see if there are any questions that pique my curiosity (or that make me feel like "hmm, I know that" or "hmm, I really should know that"). Table 2 shows the current list of questions for 12/9/17.

1. Perl DBI (MySQL) puts single quote instead of actual parameter in prepared statement - [7/1]
2. How to search and replace multiple lines with multiple lines - [5/5]
3. In perl, when assigning a subroutines return value to a variable, is the data duplicated in memory? - [5/3]

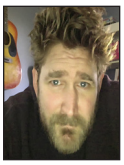
4. Is there a way to configure the default mirror for App::cpanminus (cpanm)? - [5/1]
5. NBSP malformed while using Mojo::DOM - [5/0]
6. How can I assign a weight for frequency score to a graph&'s edge using Graph::Easy - [4/2]
7. How to run multiple perl Dancer2 apps in same nginx server - [3/2]
8. How to accept self-signed certificates with LWP::UserAgent - [3/2]
9. Why are the referenced arrays values not changed? - [3/2]
10. Time::Piece (localtime/gmtime) calculation vs bash date - [3/1]

Table 2: The list at https://niceperl.blogspot.com/2017/12/cccxviii-stackoverflow-perl-report.html

And with that, we come to the end of these reports that are great for finding interesting things in the Perl world. Take care, and I'll see you next time.

iVoyeur
OpenTracing

DAVE JOSEPHSEN



Dave Josephsen is a book author, code developer, and monitoring expert who works for Sparkpost. His continuing mission: to help engineers worldwide close the feedback loop.
dave-usenix@skeptech.org

As I write this, I am just back from KubeCon and CloudNativeCon [1], where process isolation is a business plan and all your friends work for Microsoft. I freely admit: it was a confusing conference for me in many ways; in fact, trying to get it all down on paper now, I even find the ways in which it was confusing, confusing. Rarely do I find myself so confused that I must engage in the process of attempting to categorize my own confusion, but this is definitely one of those times, so let's see what we can do here.

I suppose the best place to begin is with the ecosystem, which is currently undergoing so rapid an explosion of growth that the Cloud Native Computing Foundation (CNCF) organizers literally had posters on the wall to remind everyone just what the CNCF actually consisted of. Each CNCF project was also given its own space in the keynotes wherein it introduced itself to the user-base, which gave one the sense that a great deal of the current organizational structure had only recently been ironed out. There were also 20 (!) keynotes, covering project updates on 14 CNCF core projects (many of which I was hearing about for the first time). That's ignoring, of course, the parallel explosion of Kubernetes-related start-ups outside the CNCF, all fighting for mind-share, whose founders seem invariably to happen to be former Google employees.

To be clear, I'm making that observation without my tinfoil hat on. To be sure, one might be tempted to infer from the founders homogeneity of pedigree, some greater and possibly diabolical plan, but if such a plan existed I feel like there would be a lot less redundancy among them. Currently there are, for example, 10 competing container-runtimes (at least): Docker, rkt, containerd, CRI-O, LXC, OpenVZ, systemd-nspawn, machinectl, lkvm, and Kata containers. (That's not counting the proprietary container runtimes used by the platform behemoths like AWS, Google Compute Engine, and Azure.)

Speaking of Azure, remember Microsoft? The company that stole all their core products and then spun off BSA [2] to sue everyone for copyright infringement? Remember? They were the ones who anti-competitively buried everyone they couldn't buy, and then sent SCO to assassinate Linux with a copyright lawsuit?

All totally so five minutes ago. At KubeCon, Microsoft is that low-key, tastefully appointed booth toward the back, where a well-spoken, highly tattooed twenty-something is speaking to passersby earnestly and excitedly about the future of open source while handing out rad Golang stickers. As for the other vendors on the floor, I only recognized half a dozen or so. It was like walking the vendor expo in a parallel dimension where Disney is an evil media syndicate hell-bent on owning everything, and Microsoft a benevolent open-source cheerleader and funder of hacky experimental Google code.

And speaking of anti-competitiveness, despite the myriad overlap in functionality between so many of the tools, I never came away with the sense that any of them were serious competitors. I mean, it's pretty obvious you're in competition if you are currently one of 10

possible mutually exclusive container-runtimes for Kubernetes, but having been in the room with them at the runtime salon, I can tell you, the lack of competitive tension between them was, well, confusing.

One thing there could be no confusion about was the CNCF's choice of monitoring tool, emphatically Prometheus [3]. And while, yes, we should talk about that eventually, right now my heart pulls me in another direction: namely, the OpenTracing API [4] project.

Have you read the Dapper [5] paper? Published in 2010, it describes Google's production distributed systems tracing infrastructure. I bring it up because OpenTracing owes its lineage directly to Dapper, so if you really want to sink your teeth in, that paper is probably the best place to start. It's also just a really good read.

I can hear you asking "Really, Dave?! Distributed tracing?!" I know, I know; talk about confusing. First of all, it isn't even monitoring, it's application performance debugging or something like that. And second, it's basically made of magic and impossible for laymen to comprehend, and anyway all the tracing stuff out there is proprietary and expensive. Also, I heard sampling is involved, and anyone who has read anything about monitoring in the last 10 years *obviously* knows that nothing but raw, unsampled, nanosecond-resolution metrics can solve production issues in the real world. MONITOR EVERYTHING HOOYA!

Hear me out for a second, though; I've been doing this for a while, and one thing I've seen quite a bit of is abstraction layers that make monitoring irrelevant. VPNs and tunneling protocols breaking SNMP traps, JVMs breaking systems memory monitoring, VMs breaking process monitoring, containers breaking VM monitoring, and on and on. If the Fundamental Theorem of Software Engineering [6] states all problems can be solved with one additional layer of abstraction, I propose this corollary: *every monitoring system can be made irrelevant with one additional layer of abstraction.*

Here's a heads-up from yours truly, even if Kubernetes isn't the inevitable future of computing everyone says it is; we're in for a *drastic* increase in the use of abstraction layers in the next 10 years. This is an important reason I'm such a big fan of StatsD and the process emitter/reporter pattern [7], wherein we move our monitoring up the stack into the process itself and let the processes we care about emit metrics directly to a monitoring system rather than trying to infer "badness" from system-level metrics. It's difficult for anything to break your monitoring when the programs you care about carry their monitoring around inside them, but even the process-emitter pattern has some abstraction to worry about—namely, microservices infrastructure.

The services design pattern reduces the amount of work that a given process performs. A service is the smallest useful piece of software that can be defined simply and deployed independently (a program that does one thing and does it well), and therefore the metrics it emits are smaller in scope. Instead of, say, one process emitting 10 metrics that expose the entire inner workings of a given job, we now have one metric each from 10 different processes.

Maybe that's fine. If we have a problem that's endemic to one service, it should be easy to pinpoint, but if our problem is the result of a particular call-path or the accumulated latency of many calls to multiple problematic services, we have a correlation problem on our hands. To solve problems with requests moving between multiple processes, we need to know which metric measurements relate to the same individual request.

In many ways, I think distributed tracing acts like a multi-process-aware metrics emitter. Tracing is monitoring; it's just scoped a little differently. Instead of monitoring a host, or a daemon, or an application, we are monitoring requests.

But how do we monitor a request, Dave? Requests are ephemeral. We can't put our hands on them.

Hogwash. Ops has been doing it for decades. Think of the `Received:` header in an SMTP request. Each mail server that has a hand in message delivery knows to unpack and add its own `Received:` line to the email headers. Using those lines, we can dissect the path an email took from sender to recipient, as well as using the date/time stamps to derive latency metrics between hops. Distributed tracing does the same thing to propagate ad hoc metrics between hops by way of the HTTP headers, or whatever other transport is being used.

All we need is a standard that describes the structure of that header, and a collection of language APIs that implement the standard so services can search for, unpack, modify, and repackage the header regardless of the language they were written in or the architecture they run on. SMTP's `Received:` header, along with the rest of the email headers, doesn't work by magic; they've been implemented and reimplemented in every language on every architecture that has ever needed to send an email.

Another interesting aspect of SMTP's `Received:` header is that anything can consume it at any time. The implementation is indifferent with respect to its consumers; rather than being purposefully designed for this or that sort of introspection system, it can use anything that knows how to unpack and parse it.

Like SMTP's `Received:` header, the OpenTracing project provides a consumer-agnostic means of tracing individual requests through large, high-volume distributed systems. It's implemented as a header that piggy-backs along as a request makes

its way through a distributed application. OpenTracing provides APIs in nine languages, which makes it trivial for you to unpack, modify, and repackage the header without having to worry about the wire-protocol details.

Unlike SMTP, distributed application requests aren't linear by nature. Your request to `/foo/events` might spawn subsequent requests to `/foo/user-events` and `/foo/app-events`, for example, along with one or more database requests to look up user-IDs or authorize your request. When one request begets another that it depends on, OpenTracing describes that relationship in terms of parents and children. When a request begets another that it doesn't depend on (say a non-blocking callout to a logging service or a cache-write), OpenTracing describes the relationship as a "FollowsFrom" relationship. Each individual request (or operation) is described by a `span` struct, while the relationships between individual spans are maintained by a `SpanContext`.

Your job as a user of the API is to instrument your code to create a span roughly at each process boundary (wherever a request is sent or received and at exit). Within each span, you can create tags to track metrics like wait times or log process details.

My mention of database calls in the paragraphs above was intentional. How can we hope to meaningfully trace requests that cross process boundaries into binary monoliths like MySQL or Cassandra? To be sure, we can time our DB interactions from the client-side, but everything happening inside the DB is a black box to us.

The good news is, given that OpenTracing is an API, support for it is slowly being proliferated into web-frameworks like Flask, RPC-layers like gRPC, databases like Cassandra, and even web-servers like Nginx. These tools all fully support existing OpenTracing SpanContexts today, automatically unpacking them and adding new spans to provide a uniform source of insight into critical processes that have historically required vastly different monitoring strategies.

Confusingly (but on-message with respect to the greater Kubernetes community), there are nine (!) different tracer implementations that can be used to inspect OpenTracing data: Zipkin, Jaeger, Appdash, Lightstep, Hawkular, Instana, sky-walking, inspectIT, and stagemonitor. Some of these are language specific and others proprietary. Jaeger, Zipkin, and Lightstep are all good places to start for generalists.

I'm kind of in love with the OpenTracing project's goal and implementation, and I hope I've done both of those justice in this introduction. Tracing is monitoring, and it's not made of magic, though it is somewhat magical. I'm really looking forward to API support in tools like Ruby-Rails and Node, and if I can get things arranged to be able to afford the time, they're on the top of my list for OSS contributions in the new year.

Take it easy!

References

[1] KubeCon + CloudNativeCon: https://events.linuxfoundation.org/events/kubecon-cloudnativecon-north-america-2018/.

[2] The Business Software Alliance: http://www.bsa.org/.

[3] Prometheus monitoring software: https://prometheus.io/.

[4] OpenTracing API: http://opentracing.io.

[5] Dapper paper: https://research.google.com/pubs/pub36356.html.

[6] FTSE: https://en.wikipedia.org/wiki/Fundamental_theorem_of_software_engineering.

[7] Process emitter/reporter pattern: http://blog.librato.com/posts/collector-patterns.

For Good Measure
Between a Block and a Hard Place

DAN GEER AND DAN CONWAY

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

Daniel G. Conway serves on the faculty at the University of South Florida, where he teaches blockchain technology, analytics, and data science. He has previously also served at Notre Dame, Indiana, Iowa, and Northwestern. He can be reached at dconway@usf.edu.

That Rumbling Sound

If you have been living under a rock the past five years, you will still have detected increasing vibrations in your rock, perhaps even out and out warming, depending on the damping effect of your rock's material. If your rock is made of ideal material, then the repeated message in those vibrations has been clear. Otherwise, we will disintermediate the Fourier transform for you:

Bitcoin. Ethereum. Blockchain.

In this column, we examine the risk footprint of Bitcoin and other cryptocurrencies—not general risk, but risk in the computer security sense.

Debuting shortly after the fall of the iconic Lehman Brothers, and driven by global anti-establishment sentiment, the top 100 virtual currencies have now amassed a market value exceeding $500B, 35% greater than that of JP Morgan Chase (~$366B). Bitcoin itself is up over 100,000% in those five years past. All the numbers in this column were true as of the hour we turned the column in; you will, yourself, want to update them because all are volatile, to say the least. Go to http://geer.tinho.net/fgm/fgm.geer.1803.references.html for the full set.

While some of this movement might be reasonably classified as hysteria (the CryptoKitties game clogging up the Ethereum network, with the highest priced "cat" selling for $117,712; Bitcoin mining in the trunk of a Tesla; and a FuckCoin raising $30K in 30 minutes), dismissing the entire movement seems too easy—but how much is dismissing it like whistling past the graveyard?

Until this year, China had been the most active trader in Bitcoin, but then the government made it all but illegal. Japan is currently the most active trader, accounting for 43% of transactions, followed by the US at 29%, European countries at 8%, South Korea at 4%, and China at 1.6%. Both this year and last, and interestingly parallel, according to the World Intellectual Property Organization, technology patents are running at three million per year, and the four countries that account for 78% of BTC trading account for 78% of those three million patents: China 36%, US 18%, Japan 16%, and South Korea 8%.

But to the point, on Sunday, December 10, the Chicago Board of Exchange launched Bitcoin futures. The day CBoE "GXBT" was announced, 100,000 new accounts were created on Coinbase, which now claims over 10,000,000 users. India has approximately 30,000 users active at any moment.

What Do the Numbers Say?

Estimates of the number of detectible Bitcoin miners range from 5,000 to 100,000. The wide uncertainty range can be interpreted as the existence of discomfort regarding their discovery. The other 1355 cryptocurrencies, of course, have their own mining interests, and these are just the public blockchains. It is reasonable to believe that private, permissioned blockchains, such as those built on Multichain, will dwarf public blockchains in scale and variety going forward.

In round numbers, the current hash rate is 12M hashes per second to power 450,000 transactions per day, with transactions totaling $2B/day, an average transaction of $4,444. This compares to 704,000,000 credit card transactions per day totaling $54.8B, an average transaction of $77. The Federal Reserve ACH (Automated Clearing House) reports $86.7B/day in clearance, an average transaction of $1,680. To steal a line from the musical *A Chorus Line*, Bitcoin is "Too young to take over, too old to ignore; I'm almost ready, but...what...for?"

The growth rate for Bitcoin has been exponential, not linear, yet, according to Cisco estimates, overall peer-to-peer network traffic is not expected to grow at all through 2021, the smallest change of all network sub-segments, versus the leading sub-segment, gaming, at 62% growth.

The annual electricity consumption of Bitcoin's proof-of-work has been variously estimated, but we'll go with 8.27 terawatt-hours per year. That is "less than an eighth of what U.S. data centers use, and only about 0.21 percent of total U.S. consumption... Global production of cash and coins consumes an estimated 11 terawatt-hours per year. Gold mining burns the equivalent of 132 terawatt-hours. And that doesn't include armored trucks, bank vaults, security systems, and such." So, one can plausibly argue that Bitcoin is a "green" technology [1].

Attack Surface

All new technology introduces new failure modes, that much is certain, but what is the proportionality constant here? Is it nodes, hashes, wallets, latency, or what? Bitcoin uses a peer-to-peer distributed ledger technology; integrating that ledger with the incentive structure to participate in the network, the attack surface then consists of several facets and edges, everything from the characteristics of peer-to-peer networking to that of wallet (private key) management.

Remember, peer-to-peer networks do not have a single point of failure based on IP addresses, but also understand that the mining operation is not uniformly distributed among miners. For example, 56% of Bitcoin mining is done using technology from AntPool in Beijing. As a different measure of concentration, if one were to disrupt the top-five mining pools, then one might be able to remove 70.4% of the competition. As yet another, "The top 100 Bitcoin addresses control 17.3 percent of all the issued currency. With Ethereum, a rival to Bitcoin, the top 100 addresses control 40 percent of the supply, and with coins such as Gnosis, Qtum, and Storj, top holders control more than 90 percent. Many large owners are part of the teams running these projects" [2].

In the meantime, the bounty for breaking into and modifying the "immutable" record of Bitcoin is the market cap itself: $500+B. As you probably know, that immutable record is based on elliptical curve encryption, i.e., discrete logarithm problems widely considered to be Computationally Hard and hash functions. So far, only 109-bit curves are known to have been broken, though there is some interest in understanding the random numbers used to pick initial private keys.

Bitcoin's incentive nature differs from that of BitTorrent—BitTorrent has a throttle system to restrict bandwidth to those freeloading. Bitcoin has no such self-repair facility. In fact, it is believed that 3.79 million bitcoins have been permanently lost (out of the 21 million that are the maximum number of bitcoins there will ever be), meaning the corresponding private keys required to access the bitcoins on the ledger have been lost, thus leaving those 3.79 million ledger entries orphaned. That makes the ultimate bitcoin pool size 18% smaller, and the single bitcoin 18% more valuable just for that reason alone. With a fixed upper bound on the number of bitcoins, you profit from causing other people to lose their private keys, and all without having to receive stolen property.

It's not as if those who exploit security flaws are too busy elsewhere to have noticed all this; Poloniex, a large marketplace, is warning customers not to use the app available through Google's store as they haven't created an app—it's malware.

Is Immutability a Good Thing?

EOS is a blockchain operating system that will be released in July 2018. Its ICO currently has a market cap of $4.6B. This puts it above Wendy's, Cracker Barrel, and MorningStar. That will add another layer to our security concerns.

In so many words, purported money is not the only thing that a blockchain can make immutable; by putting smart contract programs (code) on a blockchain, the code becomes immutable, and, with Ethereum and its Turing-complete language Solidity, we can trick the blockchain into executing updates by carefully using the equivalent of pointers. (Paging the Language Theoretic Security Group...)

Immutability, like anything else, is not without tradeoffs. As a case in point, Bitcoin is transparent as far as a history of what wallets have what amount of currency. Mapping those wallet addresses to IP addresses or user identities is likely not a great challenge today. In other words, blockchain immutability carries the same freight as biometric identifiers—there's no invalidating the information once revealed. (Monero is more challenging and that is so on purpose, but remember the first rule of any serious investigation: "Follow the money." What if you can't?)

All of this is perhaps too speculative, too dynamic, too ephemeral for a column on "security metrics," but our central point is that the faster the value-at-risk rises, the more certain it becomes that the structural advantage that offense has over defense will out. How these numbers change in the next year should offer some insight as to where perceived value is being pursued.

In any case, should you decide that being under your rock is not such a bad place after all, you still may want to consider investing in FortitudeCoin, which will give you priority to the survivalist community Fortitude Ranch, and thus purchase priority accommodations should your rock prove inadequate. We recommend you first make a sizable investment in our ICO "DanCoin," a fork off of FreeLunchCoin, before pursuing this path.

References

[1] Elaine O-u, "No, Bitcoin Won't Boil the Oceans," *Bloomberg View*, December 7, 2017: https://www.bloomberg.com/view/articles/2017-12-07/bitcoin-is-greener-than-its-critics-think.

[2] Olga Kharif, "The Bitcoin Whales: 1,000 People Who Own 40 Percent of the Market," *Bloomberg Businessweek*, December 8, 2017: https://www.bloomberg.com/news/articles/2017-12-08/the-bitcoin-whales-1-000-people-who-own-40-percent-of-the-market.

/dev/random
Web of Darkness

ROBERT G. FERRELL

Robert G. Ferrell is a fourth-generation Texan, literary techno-geek, and finalist for the 2011 Robert Benchley Society Humor Writing Award. rgferrell@gmail.com

Behavior outside the acceptable social norm is part and parcel of human nature. We live, work, play, and interact on a bell curve. We also compute there, as has become increasingly evident in recent years.

I won't claim to have been present at the birth of the Internet—I was only 12 years old in 1969 when the ball really got rolling—but I spent a fair amount of time in the early to mid-1980s playing with the first incarnations of TCP/IP and hanging around on USENET. As with most of my colleagues at the time, I could see the potential for NSFNET to connect, eventually, all the world's academic institutions and libraries and thus be a really useful thing to have around. I don't remember predicting the monster it would actually become, but I did once tell a skeptical boss in the early '90s that having a web presence and email would be essential to doing business within a few years.

By the time Mosaic was released, enabling pictures as well as text to dangle in the sticky participles of the World Wide Web, the Internet had been public for a couple of years, and the highways, byways, and alleys were beginning to take shape. Some of those Information Alleyways were better lit than others. It didn't take long for commerce to insinuate itself into the barrage of packets getting flung to and fro across this nascent behemoth. The object of commerce is profit, and profit can further be divided into legitimate and illicit—although that demarcation line can get a little smeary on occasion.

In the age of our innocence, Americans as a nation tended to believe that the default setting for the human conscience was "beneficent." That might even have been true at some point. I can assure you, that is no longer the case. Our government and the Internet are prime examples. I leave the rationale for my first case in point to your own research, but for the Internet exemplar, allow me to break it down for you.

The Internet was built, largely, on a platform of hacking. The original architects were working without blueprints or manuals, innovating as they went and solving technical challenges by the seat of their pants: that, dear friends, is the purest incarnation of the hacker's art. As I have insisted on several occasions in this column, hacking has no innate connection whatever with criminality. That does not mean, however, that some of the less savory individuals who have taken up the hacker's mantle haven't applied those skills toward less than fully transparent pursuits.

While we've subsequently built a huge framework of what is somewhat ironically referred to as "legitimate" commerce on top of this hacker-originated underpinning, it should come as no surprise to anyone sentient that amoral entrepreneurship still thrives down in the Internet's moldy sub-basement. What I'm not sure most people realize is that without those nefarious roots permeating its foundation, the commercial aspects of the Internet would probably collapse.

Let's do a little deconstruction. What drives most technological advances? Is it pure research? The creative spirit? Impressing that girl in AP calculus? Or is it the military-industrial juggernaut that sucks up most of our tax dollars as it rolls past and spits them

out again as contractor funding? If you answered (D), you're making forward progress toward the distant, craggy shoreline of enlightenment.

Now, where does all that money go, really? Into "research and development," as firms with their fingers shoved far into the national budgetary pie will invariably claim? No. It goes to the Internet. It goes to two-day shipping, streaming media services, online gaming, day trading, and cryptocurrency transactions. It also goes to porn and drugs, both licit and illicit—commodities in which the "Dark Web" specializes. Without these last two line items the infrastructure of the Internet, and therefore of our national prosperity, would fold in upon itself like a house of cards caught in a sudden gale.

If that sounds needlessly cynical, consider this: there are more programmers in the world than jobs to support them, due at least in part to the uneven clumping of money available to pay said coders. If you want to make a decent living as a programmer in the conventional arena you need to live and work in a place where salaries are commensurate with the value you can add to your employer's products. The virtual nature of modern labor pools has ameliorated this market imbalance to a certain extent, but some considerable inequity remains.

The Dark Web makes no such distinctions. The amount of money one can earn there is not at all related to where one calls home physically. In fact, the less likely a nation is to grease the wheels of traditional commerce, the easier it is to set those of the Dark Web spinning. Those people who make money on the Dark Web return at least some of it to the conventional commercial sectors of their nations of residence. They also go on in many cases to apply the talents they've honed in the underworld to more conventional projects—to which they would not have been able to contribute without the succor of their ill-gotten gains.

There is a growing industry related to combatting the efforts of those who populate the Dark Web. The cat-and-mouse games played with pirates, for example, have led to many advances in peer-to-peer networking, cryptography, protections against DDOS and other large-scale attacks, and even file integrity algorithms. Without the economic pressures presented by actual and potential losses to criminal activity, these developments would have been far slower in coming.

The "dox the government" movement, which has revealed for the first time the depths to which intelligence agencies have penetrated the lives of ordinary citizens, makes considerable use of the Dark Web as well. One might argue that these releases have done more damage than good, but no matter your stance on the issue, you can't realistically question their impact. The spiders spinning the Dark Web shine their light in unexpected places.

Welcome, valued customer. Your call is very important to us. Please have your stolen ID and someone else's credit card number handy. Remember to stay on the line until you have verified the color, size, model number, and/or dosage of your selected items, and thank you for choosing the Dark Web. We know you don't really have a choice for most of this stuff.

Book Reviews

MARK LAMOURINE

Fluent Python

Luciano Ramalho
O'Reilly Media, 2015, 474 pages
ISBN 978-1-491-9-46008

For a long time, I've told people I only need one book on Python: David Beazley's *Python Essential Reference*. I don't need tutorials any more—none of the references I've read does a better job than Beazley of balancing completeness and compactness. In *Fluent Python*, Ramalho has given me both a second book to keep close and an archetype for a type of book that I would like to see more of.

Ramalho sets out to teach not just what Python can do but how it works. Python has a lot of history, and the feature set is a mixed bag. It inherits from a lot of sources, and Ramalho is familiar with them and talks about them where it adds to the reader's understanding.

The thing I really like about *Fluent Python* is that Ramalho talks about the characteristics that make Python special, different, and, especially, expressive and readable. He devotes a lot of space to practical aspects and to idiom.

This isn't a book for beginning programmers or even developers just starting to learn Python. While there are echoes of the topics you'd expect to see in a programming language book, they are treated from the standpoint of internals and language-design choices. In many cases, Ramalho addresses would-be language purists on their own terms explaining why the language developers made their choices and how those affect Python operation and performance. One standout is how Python has adopted functional programming concepts, and how the traditional constructs (filter, map, reduce), while they exist, are largely better written using list comprehensions in Python. Ramalho explains how comprehensions in Python are both more clearly expressive and more efficiently implemented than a classic MapReduce construct.

The author doesn't shy away from what even he considers to be warts on the language. Python's syntax restricts the lambda construct in ways that make anonymous functions nearly useless. He shows instead how to use regular functions, which, while more verbose, are often clearer for the reader and developer trying to debug a set of deeply nested anonymous functions. Developers coming from other scripting languages also face difficulties that arise when trying to extend built-in types. In this, I agree with him that the seeming problem is, in the grand scheme, a good thing, discouraging people from trying to do things which lead to obscure or too-clever code.

I especially liked the sections on decorators and his coverage of iterators and generators. I've often seen tutorials on the syntax for creating and using both of these constructs, but Ramalho discusses both the theory behind them (decorators are closures? Oh!) and how they behave in operation. I find the under-the-hood aspects to be useful when I'm deciding when to use constructs like these.

At the end of each chapter, the author includes an extensive references section and, my favorite, a "Soap-Box" section where he talks about his preferences, biases, and impressions. These give the reader both a sense of his background and some input on topics that can be interpreted as opinion (or religion).

These days I mostly skim books and then set them aside. *Fluent Python* is one I mean to revisit. It's too meaty to completely digest in one pass. Now I know where to look when I want to learn more about Python's more interesting possibilities.

Once Upon an Algorithm

Martin Erwig
Massachusetts Institute of Technology, 2017, 317 pages
ISBN 978-0-262-0-36603

I was both intrigued and dubious when I first picked up Erwig's book. I like the idea of using metaphor and, especially, storytelling to make technical subjects accessible. I like to use them even when talking to my peers since a good metaphor can often be a shortcut to understanding. Presenting technical topics this way, however, risks oversimplifying. Such a presentation can either give a clear but incomplete treatment or bend metaphors so badly in an attempt to be make them rigorous that they lose their relevance. You can only push stories so far when applying them to complex topics.

Erwig starts off simply enough: Hansel and Gretel mark their path and then find their way home. They do it by following a series of repeated steps, first marking their path with stones and later bread and then following the markers back. This is the kind of thing I'd expect in a popular treatment of algorithms. The vocabulary and writing style is at odds with the simple story. Erwig is using children's stories, but he's not telling one.

It turns out that *Once Upon an Algorithm* is aimed at neither the purely technical nor the broad popular audience. Instead the intended audience, one that I am part of, is outside the field of computer science: the curious, dedicated lay reader.

The author organizes the book into two sections: "Algorithms" and "Languages." This works because, contrary to my expec-

tations, he's not trying to explain algorithms by coding them. He actually treats the Theory of Computation right from the beginning, using the stories and the algorithms they illustrate to introduce the deepest concepts of computation: abstraction, representation, complexity, and computability.

As noted, in the first chapter, Erwig uses Hansel and Gretel to conceptualize an algorithm as a set of steps that can be followed to achieve some goal. In the second chapter, Sherlock Holmes is used to illuminate modeling, data representation, and abstraction. In the third, Indiana Jones is the focus used to discuss searching and sorting. In all three chapters, Erwig ends by talking about the deep questions that arise when trying to represent the real world with mathematics, logic, and machines. The third chapter closes with the best non-mathematical description I've read about the meaning of *P* (the set of problems computable in polynomial time), *NP* (problems where a given solution can be tested in polynomial time), and why the idea that *P* = *NP* (or not) is important for computation.

As if that's not enough, the second half of the book covers the theory of formal languages. Erwig uses the song "Somewhere Over the Rainbow" and the example of musical notation to show how ubiquitous "computation" is. In a very real sense, a musical score is a "program" that can be converted into a result (a performance) by a computer (the conductor and musicians). The movie *Groundhog Day* serves to show how iteration and looping work (and why terminating conditions are so important). *Back to the Future* is the inspiration for a discussion of recursion, and *Harry Potter* serves as the backdrop for the final chapters on the theory of abstraction and types.

In the end, I found the discussion to be on-point and clear. Erwig doesn't condescend to the reader despite how easy that would be given the thesis that common stories can illustrate the theory of computation. He shows in this way how computation isn't really some strange esoteric field but is grounded in everyday ideas and activities that anyone can relate to. The title of the book might lead someone to expect a watered down popular "dummies" treatment, but that would be a mistake. Erwig does indeed know his audience and writes to them. That audience will be well served by *Once Upon an Algorithm.*

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.

USENIX Board of Directors

Communicate directly with the USENIX Board of Directors by writing to board@usenix.org.

PRESIDENT
Carolyn Rowland, *National Institute of Standards and Technology*
carolyn@usenix.org

VICE PRESIDENT
Hakim Weatherspoon, *Cornell University*
hakim@usenix.org

SECRETARY
Michael Bailey, *University of Illinois at Urbana-Champaign*
bailey@usenix.org

TREASURER
Kurt Opsahl, *Electronic Frontier Foundation*
kurt@usenix.org

DIRECTORS
Cat Allman, *Google*
cat@usenix.org

David N. Blank-Edelman, *Apcera*
dnb@usenix.org

Angela Demke Brown, *University of Toronto*
demke@usenix.org

Daniel V. Klein, *Google*
dan.klein@usenix.org

EXECUTIVE DIRECTOR
Casey Henderson
casey@usenix.org

Notice of Annual Meeting

The USENIX Association's Annual Meeting with the membership and the Board of Directors will be held at 6:00 pm on Tuesday, July 10, in Boston, MA, during the 2018 USENIX Annual Technical Conference.

Announcement of Changes to Future LISA Conferences

A Statement by the LISA Steering Committee on behalf of the USENIX Association

Starting in 2018, LISA will run for three days, rather than six.

We heard you! You couldn't get your boss to agree to let you go for six days, or stepping away from other responsibilities for that long was too hard. It was also a firehose of content, and trying to keep it all in your brain upon returning to work was a real challenge. Could we take LISA and make it into a reasonably sized conference that still had amazing talks and training? Yes, we could, and we have.

LISA is one of the longest running tech conferences, originating in 1987, and we are proud that it continues to be among the most popular events in an increasingly crowded field. Over the past 30 years, LISA has grown from a short workshop to a multi-track conference, evolving in format to suit the needs of a changing community. Most recently, we've added training sessions within the main conference program to allow instructors to rapidly expose attendees to new topics; created LISA Lab and LISA Build to increase hands-on experience; and expanded invited talks to allow more industry experts to share their knowledge.

The new format will better allow attendees to fit LISA into their busy travel, work, and personal lives. We'll continue to highlight the strengths of LISA, with three days of talks and training, LISA Build and LISA Lab, the expo, and the great conversations that happen in the hallway track and Birds-of-a-Feather (BoF) sessions. The more focused scope also gives us the opportunity to push the proposal deadline closer to each

conference to allow for emerging content and the latest updates, while also giving USENIX more flexibility around location and timing. This change also aligns the format with the rest of the family of systems engineering conferences, spread throughout the year in multiple geographic locations: LISA, SREcon Americas, SREcon Europe/Middle East/Africa, and SREcon Asia/Australia.

LISA is an amazing conference and one of the highlights of our year. The LISA18 Call for Participation will be released this quarter, and we encourage you to submit a proposal. The deadline for submissions will be May 24, 2018. Interested in helping organize the conference? Send email to lisa@usenix.org. We look forward to seeing all of you in Nashville this coming October!

Members: Cast Your Vote!

Ballots due March 30, 2018

Ballots for the 2018 USENIX Board of Directors election have been mailed to all USENIX members. Please mark your ballot and return it in the mail to arrive no later than March 30, 2018.

The election results will be announced in April.

www.usenix.org/board/elections18

FAST '19: 17th USENIX Conference on File and Storage Technologies

February 25–28, 2019, Boston, MA, USA

Sponsored by USENIX, the Advanced Computing Systems Association

Important Dates

- Submissions due: **Wednesday, September 26, 2018, 8:59 pm PDT**
- Notification to authors: **Tuesday, December 11, 2018**
- Final papers due: **Thursday, January 24, 2019**

Conference Organizers

Program Co-Chairs
Arif Merchant, *Google*
Hakim Weatherspoon, *Cornell University*

Program Committee
TBA

Steering Committee
Remzi Arpaci-Dusseau, *University of Wisconsin—Madison*
Angela Demke Brown, *University of Toronto*
Greg Ganger, *Carnegie Mellon University*
Casey Henderson, *USENIX Association*
Kimberly Keeton, *HP Labs*
Geoff Kuenning, *Harvey Mudd College*
Florentina Popovici, *Google*
Erik Riedel, *Dell Technologies*
Jiri Schindler, *SimpliVity*
Bianca Schroeder, *University of Toronto*
Margo Seltzer, *Harvard University and Oracle*
Keith A. Smith, *NetApp*
Eno Thereska, *Amazon*
Carl Waldspurger, *Carl Waldspurger Consulting*
Ric Wheeler, *Red Hat*
Erez Zadok, *Stony Brook University*

Overview

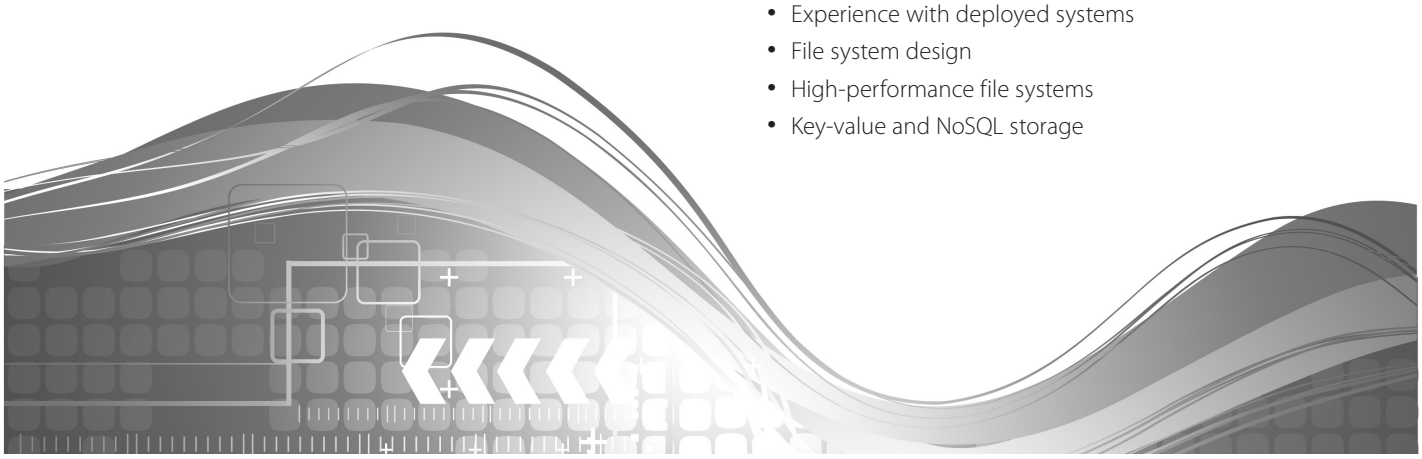
The 17th USENIX Conference on File and Storage Technologies (FAST '19) brings together storage-system researchers and practitioners to explore new directions in the design, implementation, evaluation, and deployment of storage systems. The program committee will interpret "storage systems" broadly; papers on low-level storage devices, distributed storage systems, and information management are all of interest. The conference will consist of technical presentations including refereed papers, Work-in-Progress (WiP) reports, poster sessions, and tutorials.

FAST accepts both full-length and short papers. Both types of submissions are reviewed to the same standards and differ primarily in the scope of the ideas expressed. Short papers are limited to half the space of full-length papers. The program committee will not accept a full paper on the condition that it is cut down to fit in the short paper page limit, nor will it invite short papers to be extended to full length. Submissions will be considered only in the category in which they are submitted.

Topics

Topics of interest include but are not limited to:

- Archival storage systems
- Auditing and provenance
- Big data, analytics, and data sciences
- Caching, replication, and consistency
- Cloud storage
- Data deduplication
- Database storage
- Distributed and networked storage (wide-area, grid, peer-to-peer)
- Empirical evaluation of storage systems
- Experience with deployed systems
- File system design
- High-performance file systems
- Key-value and NoSQL storage



- Memory-only storage systems
- Mobile, personal, embedded, and home storage
- Parallel I/O and storage systems
- Power-aware storage architectures
- RAID and erasure coding
- Reliability, availability, and disaster tolerance
- Search and data retrieval
- Solid state storage technologies and uses (e.g., flash, byte-addressable NVM)
- Storage management
- Storage networking
- Storage performance and QoS
- Storage security

Deployed Systems

In addition to papers that describe original research, FAST '19 also solicits papers that describe large-scale, operational systems. Such papers should address experience with the practical design, implementation, analysis, or deployment of such systems. We encourage submission of papers that disprove or strengthen existing assumptions, deepen the understanding of existing problems, and validate known techniques at scales or in environments in which they were never before used or tested. Deployed-system papers need not necessarily present new ideas or results to be accepted, although that is certainly welcome, but should offer useful guidance to practitioners.

Authors should indicate on the title page of the paper and in the submission form that they are submitting a deployed-system paper.

Submission Instructions

Please submit full and short paper submissions (no extended abstracts) by 8:59 pm PDT on September 26, 2018, in PDF format via the submission form, which will be available here soon. Do not email submissions. There is no separate deadline for abstract submission.

- The complete submission must be no longer than 11 pages for full papers and 5 pages for short papers, excluding references. The program committee will value conciseness, so if an idea can be expressed in fewer pages than the limit, please do so. Supplemental material may be added as a single-but-separate file without page limit; however the reviewers are not required to read such material or consider it in making their decision. Any material that should be considered to properly judge the paper for acceptance or rejection is not supplemental and will apply to the page limit. Papers should be typeset on U.S. letter-sized pages in two-column format in 10-point Times Roman type on 12-point leading (single-spaced), in a text block being no more than 7" wide by 9" deep. Labels, captions, and other text in figures, graphs, and tables must use reasonable font sizes that, as printed, do not require extra magnification to be legible. Because references do not count against the page limit, they should not be set in a smaller font. Submissions that violate any of these restrictions will not be reviewed. The limits will be interpreted strictly. No extensions will be given for reformatting.
- A LaTeX template and style file are available on the USENIX templates page (www.usenix.org/conferences/author-resources/paper-templates).
- Authors must not be identified in the submissions, either explicitly or by implication. When it is necessary to cite your own work, cite it as if it were written by a third party. Do not say "reference removed for blind review." Any supplemental material must also be anonymized.
- Simultaneous submission of the same work to multiple venues, submission of previously published work, or plagiarism constitutes dishonesty or fraud. USENIX, like other scientific and technical conferences and journals, prohibits these practices and may take action against authors who have committed them. See the USENIX Conference Submissions Policy (www.usenix.org/conferences/author-resources/submissions-policy) for details.
- If you are uncertain whether your submission meets USENIX's guidelines, please contact the program co-chairs, fast19chairs@usenix.org, or the USENIX office, submissionspolicy@usenix.org.
- Papers accompanied by nondisclosure agreement forms will not be considered.

Short papers present a complete and evaluated idea that does not need 11 pages to be appreciated. Short papers are not workshop papers or work-in-progress papers. The idea in a short paper needs to be formulated concisely and evaluated, and conclusions need to be drawn from it, just like in a full-length paper.

The program committee and external reviewers will judge papers on technical merit, significance, relevance, and presentation. A good research paper will demonstrate that the authors:

- are attacking a significant problem,
- have devised an interesting, compelling solution,
- have demonstrated the practicality and benefits of the solution,
- have drawn appropriate conclusions using sound experimental methods,
- have clearly described what they have done, and
- have clearly articulated the advances beyond previous work.

A good deployed-system paper will demonstrate that the authors:

- are describing an operational system that is of wide interest,
- have addressed the practicality of the system in more than one real-world environment, especially at large scale,
- have clearly explained the implementation of the system,
- have discussed practical problems encountered in production, and
- have carefully evaluated the system with good statistical techniques.

Moreover, program committee members, USENIX, and the reading community generally value a paper more highly if it clearly defines and is accompanied by assets not previously available. These assets may include traces, original data, source code, or tools developed as part of the submitted work.

Blind reviewing of all papers will be done by the program committee, assisted by outside referees when necessary. Each accepted paper will be shepherded through an editorial review process by a member of the program committee.

Authors will be notified of paper acceptance or rejection no later than Tuesday, December 11, 2018. If your paper is accepted and you need an invitation letter to apply for a visa to attend the conference, please contact conference@usenix.org as soon as possible. (Visa applications can take at least 30 working days to process.) Please identify yourself as a presenter and include your mailing address in your email.

All papers will be available online to registered attendees no earlier than Thursday, January 24, 2019. If your accepted paper should not be published prior to the event, please notify production@usenix.org. The papers will be available online to everyone beginning on the first day of the main conference, February 26, 2019. Accepted submissions will be treated as confidential prior to publication on the USENIX FAST '19 website; rejected submissions will be permanently treated as confidential.

By submitting a paper, you agree that at least one of the authors will attend the conference to present it. If the conference registration fee will pose a hardship for the presenter of the accepted paper, please contact conference@usenix.org.

If you need a bigger testbed for the work that you will submit to FAST '19, see PRObE at www.nmc-probe.org.

Best Paper Awards

Awards will be given for the best paper(s) at the conference. A small, selected set of papers will be forwarded for publication in *ACM Transactions on Storage (TOS)* via a fast-path editorial process. Both full and short papers will be considered.

Test of Time Award

We will award a FAST paper from a conference at least 10 years earlier with the "Test of Time" award in recognition of its lasting impact on the field.

Work-in-Progress Reports and Poster Sessions

The FAST technical sessions will include a slot for short Work-in-Progress (WiP) reports presenting preliminary results and opinion statements. We are particularly interested in presentations of student work and topics that will provoke informative debate. While WiP proposals will be evaluated for appropriateness, they are not peer reviewed in the same sense that papers are.

We will also hold poster sessions each evening. WiP submissions will automatically be considered for a poster slot, and authors of all accepted full papers will be asked to present a poster on their paper. Other poster submissions are very welcome. Please see the Call for Posters and WiPs, which will be available soon, for submission information.

Birds-of-a-Feather Sessions

Birds-of-a-Feather sessions (BoFs) are informal gatherings held in the evenings and organized by attendees interested in a particular topic. BoFs may be scheduled in advance by emailing the Conference Department at bofs@usenix.org. BoFs may also be scheduled at the conference.

Tutorial Sessions

Tutorial sessions will be held on February 25, 2019. Please submit tutorial proposals to fasttutorials@usenix.org .

Registration Materials

Complete program and registration information will be available in December 2018 on the conference website.

Rev. 02/09/2018

SAVE THE DATES!

SRECON® AMERICAS

MARCH 27–29, 2018 • SANTA CLARA, CA, USA
www.usenix.org/srecon18americas

SRECON® ASIA AUSTRALIA

JUNE 6–8, 2018 • SINGAPORE
www.usenix.org/srecon18asia

SRECON® EUROPE MIDDLE EAST AFRICA

AUGUST 29–31, 2018 • DUSSELDORF, GERMANY
The Call for Participation is now available. Submissions are due April 3, 2018.
www.usenix.org/srecon18europe

SREcon is a gathering of engineers who care deeply about site reliability, systems engineering, and working with complex distributed systems at scale. It strives to challenge both those new to the profession as well as those who have been involved in it for decades. The conference has a culture of critical thought, deep technical insights, continuous improvement, and innovation.

Follow us at @SREcon

usenix®
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

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