;login:
WINTER 2018 VOL. 43, NO. 4

Meltdown Remedies and Performance
Daniel Gruss, Dave Hansen, and Brendan Gregg

The Deconstructed Database
Amandeep Khurana and Julien Le Dem

The Five Stages of SRE
Ben Purgason

Atlas Cluster Trace Repository
George Amvrosiadis, Michael Kuchnik, Jun Woo Park, Chuck Cranor, Gregory R. Ganger, Elisabeth Moore, and Nathan DeBardeleben

Columns

Changes on the Horizon
Peter Norton

Using S3 from Golang
Chris "Mac" McEniry

Flow, Part II
Dave Josephsen

Counting Hosts
Dan Geer and Paul Vixie

The Great Simulation
Robert G. Ferrell

UPCOMING EVENTS

Enigma 2019
January 28–30, 2019, Burlingame, CA, USA
www.usenix.org/enigma2019

FAST '19: 17th USENIX Conference on File and Storage Technologies
February 25–28, 2019, Boston, MA, USA
Sponsored by USENIX in cooperation with ACM SIGOPS
Co-located with NSDI '19
www.usenix.org/fast19

Vault '19: 2019 Linux Storage and Filesystems Conference
Feburary 25–26, 2019
Co-located with FAST '19
www.usenix.org/vault19

NSDI '19: 16th USENIX Symposium on Networked Systems Design and Implementation
February 26–28, 2019, Boston, MA, USA
Sponsored by USENIX in cooperation with ACM SIGCOMM and ACM SIGOPS
Co-located with FAST '19
www.usenix.org/nsdi19

SREcon19 Americas
March 25–27, 2019, Brooklyn, NY, USA
www.usenix.org/srecon19americas

OpML '19: 2019 USENIX Conference on Operational Machine Learning
May 20, 2019, Santa Clara, CA, USA
Submissions due January 11, 2019
www.usenix.org/opml19

SREcon19 Asia/Australia
June 12–14, 2019, Singapore

2019 USENIX Annual Technical Conference
July 10–12, 2019, Renton, WA, USA
Submissions due January 10, 2019
www.usenix.org/atc19

Co-located with USENIX ATC '19

HotStorage '19: 11th USENIX Workshop on Hot Topics in Storage and File Systems
July 8–9, 2019
Submissions due March 12, 2019
www.usenix.org/hotstorage19

HotCloud '19: 11th USENIX Workshop on Hot Topics in Cloud Computing
July 8, 2019

HotEdge '19: 2nd USENIX Workshop on Hot Topics in Edge Computing
July 9, 2019

SOUPS 2019: Fifteenth Symposium on Usable Privacy and Security
August 11–13, 2019, Santa Clara, CA, USA
Co-located with USENIX Security '19

28th USENIX Security Symposium
August 14–16, 2019, Santa Clara, CA, USA
Co-located with SOUPS 2019
Winter quarter submissions due February 15, 2019
www.usenix.org/sec19

SREcon19 Europe/Middle East/Africa
October 2–4, 2019, Dublin, Ireland

LISA19
October 28–30, 2019, Portland, OR, USA

USENIX Open Access Policy

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

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

www.usenix.org/membership

;login:

WINTER 2018 VOL. 43, NO. 4

EDITORIAL

2 **Musings** *Rik Farrow*

SRE AND SYSADMIN

5 **The Five Stages of SRE** *Ben Purgason*

SECURITY

10 **Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer** *Daniel Gruss, Dave Hansen, and Brendan Gregg*

15 **The Secure Socket API: TLS as an Operating System Service** *Mark O'Neill, Kent Seamons, and Daniel Zappala*

21 **Strings Considered Harmful** *Erik Poll*

26 **CSET '18: The 11th USENIX Workshop on Cyber Security Experimentation and Test** *Peter A. H. Peterson*

SYSTEMS

29 **The Atlas Cluster Trace Repository** *George Amvrosiadis, Michael Kuchnik, Jun Woo Park, Chuck Cranor, Gregory R. Ganger, Elisabeth Moore, and Nathan DeBardeleben*

36 **The Modern Data Architecture: The Deconstructed Database** *Amandeep Khurana and Julien Le Dem*

COLUMNS

41 **And Now for Something Completely Different** *Peter Norton*

44 **Custom Binaries to Ease Onboarding Using Go** *Chris "Mac" McEniry*

48 **iVoyeur: Flow, Part II** *Dave Josephsen*

53 **For Good Measure: Nameless Dread** *Dan Geer and Paul Vixie*

57 **/dev/random: Simulation Station** *Robert G. Ferrell*

BOOKS

59 **Book Reviews** *Mark Lamourine and Rik Farrow*

USENIX NOTES

62 **Meet the Board: Kurt Andersen** *Liz Markel*

63 **USENIX Association Financial Statements for 2017**

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.

©2018 USENIX Association
USENIX is a registered trademark of the USENIX Association. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. USENIX acknowledges all trademarks herein. Where those designations appear in this publication and USENIX is aware of a trademark claim, the designations have been printed in caps or initial caps.

Cover Image created by freevector.com and distributed under the Creative Commons Attribution-ShareAlike 4.0 license (https://creativecommons.org/licenses/by-sa/4.0/).

EDITORIAL

Musings

RIK FARROW

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

Change is hard. Once you have things working, however shakily, you realize that making changes could result in a slippery slide back to where you were. Yet change is critical if you are going to advance.

The SRE culture is built upon supporting rapid change and automating away every routine activity possible. But for people outside of that culture, change remains scary, as failure is always an option. And an unpleasant option at that if things were already basically working.

The first article in this issue, by Ben Purgason, got me thinking about the culture of change. There's also a lot about security in this issue, and the combination of the two reminded me of a security audit I helped with many years ago.

Changeless

I was sitting in a cubicle in a state-level IT department. The other part of the audit team had the easier job. They were running password sniffers, and on a late '90s mostly Windows network; they were laughing a lot as the sniffer scarfed up passwords. I, on the other hand, had to comprehend the Cisco router's access control lists, about 15 pages of them.

After a few hours, I could see I was wasting my time. The network guys weren't using most of the ACLs they had given me. They were a smoke screen. They were using exactly one ACL.

And that ACL referenced a system they hadn't told me about and were pretending wasn't there. I ran an early vulnerability scanner (SATAN) on the demilitarized zone (DMZ), looking both for systems and for vulnerabilities, as presented by network services, on those systems. And not only was the mystery system there, it was unpatched and vulnerable.

The IT group had a replacement system in place and ready to go. But they hadn't enabled it, postponing the change, one that would affect any user who logged in remotely in this large state. Not changing over appeared safer then leaving the old, dangerously vulnerable system in place. Fear of what might happen if they changed over was greater than the fear of penetration of their internal network. This was in the early days of intrusion detection, and ID was beyond the scope of the audit. I still find what was likely happening scary to think about.

I can look at myself, and see how often in my life I've been forced into making changes I should have made earlier. Systems that died long after I should have migrated them to newer hardware, not hardening security because of fears I would break things I had fixed long before, and changes forced upon me by advances in technology.

In many ways, the rapid pace of changes in the big Internet-facing companies of today is both daunting and frightening. They have learned how, not just to live with change, but to thrive. For the rest of us, these are important lessons that we need to learn as well.

The Lineup

Ben Purgason starts off this issue with the five stages of SRE, based on his experience working with the SRE teams at LinkedIn. Purgason begins this article, based on his keynote at SRECon18 Asia, by describing the three founding principles of SRE at LinkedIn: Site Up, Empower Developer Ownership, and Operations Is an Engineering Problem. Purgason then enumerates the five stages, as he experienced them, working in the trenches.

Gruss, Hansen, and Gregg examine the patches made to three major OSes in response to the Meltdown exploit found in recent Intel processors. The initial work was done using Linux on a single hardware platform more as an academic project, but the solutions tried there later became the basis for patches to Linux, macOS, and Windows. Gruss also discusses how, and why, this patch impacts performance. For the most common workloads, the impact on performance is surprising.

O'Neill, Seamons, and Zappala discuss their USENIX Security '18 paper [1], where they examine the failures in the usage of TLS in OpenSSL and how to avoid them. Instead of allowing programmers to set possibly insecure defaults, their solution moves TLS into the sockets interface of the Linux kernel, and uses a system-wide set of minimal defaults. The programming model is simple enough that adding support for TLS in a tool like `netcat` required only five lines of code.

Erik Poll explains the intersection of forwarding and parsing. Most service applications today consist of front and back ends, with the front end accepting requests, then distributing the work to be done to back-end services. The problem, as Poll points out, is where and how should forwarded input, or potential injection faults, be processed, as these may conceal forward attacks. Poll describes anti-patterns, techniques that have been shown to be flawed, and then suggests remedies that are safer and follow LangSec best practices, as found in his paper [2].

Peter Peterson wrote a summary of CSET '18, a one-day workshop about cybertesting, measurement, and experimental testbeds. The talks were wider-ranging than I would have thought, and Peterson provides succinct summaries of each talk and one panel.

George Amvrosiadis et al. have also taken a crack at experimental verification. In their USENIX ATC '18 paper [3], the group compares Google cluster-behavior traces to traces they have collected from two US defense/scientific and two financial analytics clusters. The Google cluster traces have been the standard for measuring the value of techniques in cluster research, but this group shows that Google traces might actually be outliers when compared to how many others use cluster computing.

Amandeep Khurana and Julien Le Dem demonstrate how database storage has changed over the decades. From IBM mainframes specifically designed for storing records, to SQL, and finally to the much more loosely structured data-lakes we see today, this team explains how and why data storage has evolved.

Peter Norton takes his column in an unusual direction. Peter focuses on the changes taking place in the Python community with the departure of its Benevolent Dictator for Life, Guido van Rossum. The community seems to be cautiously and carefully moving forward, perhaps to avoid disturbing the ecosystem that is Python.

David Blank-Edelman is taking this issue off.

Mac McEniry chose to write about how to access AWS S3 storage using Golang. His particular challenge was how to provide some configuration information to a widely distributed set of servers, and while his example does rely on S3, the ideas in his column can be used as the basis for other ways of sharing some small amount of data.

Dave Josephsen continues the discussion of flow that he started in his Fall '18 column. Dave explains the problems the team at Sparkpost ran into when they attempted to use `syslogd` to collect and distribute log messages from their front-end Nginx servers. Think "firehose" and you are getting the idea of the volume of log messages.

Dan Geer and Paul Vixie work at enumerating the number of systems attached to the Internet. Starting with named systems, they examine other means of at least estimating just how many devices are out there, a daunting task. The ballooning growth of IoT devices, along with generally poor security (by negligence) and inability ever to be updated bodes poorly for the future of Internet security.

Robert Ferrell considers the notion that we exist only as a great simulation. Other great minds, like Elon Musk, have proposed this recently, but, as always, Robert has his own take on our digital future.

Mark Lamourine has three book reviews, and I have written one.

Announcements

The year 2019 marks the 50th anniversary of the UNIX system (™), and Clem Cole, past USENIX Board President, has written an article that helps to explain just why UNIX has been so successful. I've read his article several times, and attempted to either create a "digest" version or split his article into smaller parts. In the end, I felt you are better off reading the original.

Clem makes many good points about UNIX, such as, unlike other operating systems of the time, UNIX was a system written by programmers for programmers. If you consider IBM's OS/360 using the perspective presented by Khurana and Le Dem in their article, you can see that Clem's point is valid: UNIX had a very different purpose right from the start.

Clem's article originally appeared in CNAM, a French publication that examines the history of technology sciences [4], and we present an English version of his article here [5]. If you plan on disrupting the usual course of events when it comes to new systems, I recommend that you take the time to read his entire article.

There was another event that occurred recently, although I suspect that it passed well below most people's radar. Early issues of *;login:*, starting with volume 8 in February 1983, are now online. Thanks to the efforts of USENIX staff, particularly Olivia Vernetti and Arnold Gatilao, you can now find these issues of *;login:*, up to the year 2000 at the time of this writing, on archive.org [6]. When USENIX made the transition to a modern web server design, some older issues were lost, and these are now also being hosted online again.

I can sincerely say that great changes are afoot. Not because I have a crystal ball or can pull predictions out of my naval lint, but simply because change constantly occurs. There is no stopping change, not even with death.

References

[1] M. O'Neill, S. Heidbrink, J. Whitehead, T. Perdue, L. Dickinson, T. Collett, N. Bonner, K. Seamons, and D. Zappala, "The Secure Socket API: TLS as an Operating System Service," in *Proceedings of the 27th USENIX Security Symposium (Security '18), pp. 799–816: https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-o_neill.pdf.*

[2] E. Poll, "LangSec Revisited: Input Security Flaws of the Second Kind," in *Proceedings of the IEEE Symposium on Security and Privacy Workshops*, 2018, pp. 329–334: http://spw18.langsec.org/papers/Poll-Flaws-of-second-kind.pdf.

[3] G. Amvrosiadis, J. W. Park, G. R. Ganger, G. A. Gibson, E. Baseman, and N. DeBardeleben, "On the Diversity of Cluster Workloads and Its Impact on Research Results," in *Proceedings of the 2018 USENIX Annual Technical Conference (ATC '18)*, pp. 533–546: https://www.usenix.org/system/files/conference/atc18/atc18-amvrosiadis.pdf.

[4] Conservatoire national des arts et métiers (CNAM), Cahiers d'histoire du Cnam, vol. 7–8, La recherche sur les systèmes : des pivots dans l'histoire de l'informatique, vol. 7–8: http://technique-societe.cnam.fr/la-recherche-sur-les-systemes-des-pivots-dans-l-histoire-de-l-informatique-ii-ii-988170.kjsp?RH=cdhte.

[5] C. T. Cole, "Unix: A View from the Field as We Played the Game," Cahiers d'histoire du Cnam, vol. 7–8, La recherche sur les systèmes : des pivots dans l'histoire de l'informatique, January 2 to July 1, 2018, pp. 111–127: www.usenix.org/login_winter18_cole_unix_cnam.pdf. This article is licensed under CC BY 2.0: https://creativecommons.org/licenses/by/2.0/.

[6] USENIX Archives: *;login:* magazine: https://archive.org/details/usenix-login.

The Five Stages of SRE

BEN PURGASON

Ben Purgason is a Director of Site Reliability Engineering at LinkedIn, responsible for the operational integrity of LinkedIn's internal software development, trust, and security infrastructure. In his six years at LinkedIn, he's developed seven successful SRE teams that partner with more than 21 distinct teams to build reliability into their entire software life cycle.
bpurgason@linkedin.com

I am a rebel in a world that made development vs. operations the status quo. I am an SRE leader with LinkedIn, and I'd like to share with you my observations and experiences building and making Site Reliability Engineering (SRE) teams successful there. In particular, I want to share a few key observations on the evolutionary path of SRE, how that path is influenced by the relationship with the software engineer (SWE), and how the allocation of roles and responsibilities changes with time.

Founding Principles

Back in 2010, SRE was founded at LinkedIn as a scrappy band of firefighters. Their first job was to put out the blazing operational fires that threatened LinkedIn's stability on a daily basis. I joined two years later in 2012, and though progress had been made, the environment was still chaotic. Putting things in perspective, when I joined the company, all on-calls were expected to be on-site by 6 a.m. each day. At the time, most of LinkedIn's traffic came from the Americas, which resulted in a drastic traffic increase as the continents woke up, usually knocking over the site. In spite of the chaos, or perhaps because of it, three fundamental principles were created to guide the development of the SRE organization.

The first is **Site Up** [1]. This is our highest technical priority—always. Site Up is ensuring that the site, service, experience, app, etc. you offer to customers work correctly and quickly enough to be relevant. Without Site Up you cannot uphold the commitments you made to your customers. If you cannot uphold your commitments, your users will eventually lose their trust in your company and will take their business elsewhere. Let this go on long enough and eventually everyone at your company (including you) will be out on the street looking for a new job.

The second is **Empower Developer Ownership**. As Bruno Connelly, head of SRE at LinkedIn, likes to say: "It takes a village to run a website." In other words, it is crucial that developers own the Operations problem as much as everyone else. Operations is a hard problem, and we're going to do a better job solving it if we bring 100 percent of our engineering talent to bear rather than just the 10 percent that happens to have "site reliability" in their title.

The third is **Operations Is an Engineering Problem**. We think Operations should be solved like any other software problem: by engineering permanent solutions to problems or, better, preventing them from happening in the first place. We do not want to solve our problems through heroes and raw effort.

As previously mentioned, Site Up is our chief technical operating priority, and nothing will ever beat it. This introduces a challenge, for Site Up is a far-reaching concept that can lead us down a short-sighted and dangerous path if we aren't careful. The other two principles, "Empower Developer Ownership" and "Operations Is an Engineering Problem," are both constraints. They ensure that our efforts to uphold Site Up are sustainable today and become increasingly effective with time.

Generation 1: The Firefighter

Let's face it, few companies invest in creating an SRE organization until either they have a raging operational fire on their hands or the minimum change velocity—that's the minimum amount of change that must occur each day for the company to still remain competitive—exceeds what traditional operations processes can handle.

At this early stage, SRE is essentially fighting fires whenever the need arises while simultaneously trying to automate the process of fire suppression. With each piece of reliable automation written, the time saved on fire suppression is freed up for use on permanently fixing problems or for forward-looking priorities such as monitoring and alerting.

Though the focus is on Site Up (incident management), we can also see the earliest influences of Operations Is an Engineering Problem through the automation of manual operational work.

Tools SRE, the Firefighters

Tools SRE, the team I founded at LinkedIn, is responsible for all SRE-related tasks around our internal Tooling (development, build, and deploy pipelines), along with those around Trust and Security.

When Tools SRE was founded, the mean time to resolve an incident (MTTR) was over 1500 minutes (yes, that is more than a full 24-hour day). So frequent and long were the outages that if we lined them all up end-to-end we would have had some level of outage every second of every day for the entire calendar year. Worse, for just under half the year, two would have been active at the same time.

So what did we do? We got after it and began our pursuit of Site Up. After every outage we held a blameless postmortem with our partner SWEs. We learned from our collective mistakes. We did our best and we didn't give up.

We instrumented our products as well as we could, and when we couldn't, we developed external observers to generate the missing metrics. We wrote alerts, we got woken up in the middle of the night, and we kept solving problems.

Be Relentless and Measure Success

Winston Churchill has been quoted, or misquoted, over the years as saying: "If you're going through hell, keep going." In a nutshell, that is the theme behind moving past this dysfunctional stage. Digging out of this particular hole takes time, patience, and an unyielding determination to succeed, but it isn't rocket science.

First, understand that **every day is Monday in Operations** [2]. There will never be an end to the problems we need to solve. In spite of that, we must continuously identify the biggest, most impactful problems and solve them. Eventually, we end up with a collection of problems that are roughly the same in quantity but greatly reduced in level of impact. Gone will be the days when developers go home early after being unable to get a build for the better part of six hours. Now you'll have users complaining about a six-minute delay in getting their build. The problems don't go away, but they do shrink in size.

Second, **what gets measured gets fixed** [3]. "What gets measured gets fixed" is a famous adage taken from a company that knew a thing or two about measurements: Hewlett Packard. Long before printers, computers, and the Internet, HP built test and measurement devices that would be more at home in a scientific laboratory than in a tech company (think oscilloscopes). They knew what they were doing—if you can measure something, you can reason about it, understand it, discuss it, and act upon it with confidence. If you cannot measure something, not only are you unable to fix it, you're unable to really understand if it's even broken or not.

If you can do just these two things, you will eventually achieve improved site stability, have more time available to invest in the future, and have a better understanding of where the next set of problems are hiding.

Generation 2: The Gatekeeper

A quick disclaimer: the gatekeeper is an evolutionary dead end that can be (almost) entirely bypassed. I include it here not to show it as the next logical step in SRE's evolution but, instead, to help any SRE team that finds itself already in it to grow past it.

As we grow past the first generation of firefighters, we achieve a basic level of operational integrity. The active operational fires are put out, and we have an increasing amount of time available for forward-looking tasks. So where do we go from here?

The natural instinct is a protective one. We just spent years digging ourselves out of a major operational hole and the last thing we want to do is immediately fall back into it. Usually this translates into a set of behaviors that amount to building a wall around "our" production with a few locked doors and keeping the keys with SRE. That way, "those" pesky developers will have to go through "us" before "their" change can impact "our" prod.

The thought is straightforward, and not entirely without merit if I'm being honest. It's motivated by fear, the fear of being in pain again.

Avoid Creating an "Us" vs. "Them" Culture

The problem with this mindset is that it quickly cements itself as a culture of "us vs. them." This is incredibly dangerous to the development of any SRE team because it will essentially limit your ability to contribute to the company and that of your SWE partners as well.

During this stage it is common for SREs to leverage their role power to claim ownership of production deployments or more generally change control. In doing so we add a new job responsibility to our SWE counterparts: get past SRE gatekeeping in the most efficient way possible.

Tools SRE, Never the Gatekeepers

The Tools SRE team I lead managed to skip this stage almost entirely. It wasn't because we were more creative or had a unique vision—it's because we got started late. When Tools SRE was founded we were outnumbered 41:1 (SWE:SRE). As a result, we knew immediately we couldn't succeed alone. Worse, any attempts at human gatekeeping would likely be ineffective—we'd just get overrun by the SWEs even if we tried to use our role power.

However, we did get a few requests for features (we were Tools SRE, after all) that directly supported human gatekeeping from other teams. Most notably was one called "service guard," which let SREs create a whitelist of individuals who could run deployments for a particular product. Essentially, this was a feature designed to force untrusted individuals to route through trusted individuals in order to do their jobs. This feature accomplished its tactical purpose in stopping unapproved deployments but also greatly increased the friction between teams, reducing their ability to collaborate.

Tribalism Has No Place in an Effective SRE Team

A man far wiser than I, Fred Kofman, has said many times that "There is no such thing as 'the hole is in your side of the boat.'" The moment you accept that your success is unachievable without your SWE partners and vice versa, you have started to grow past this generation and have begun to realize your potential. To accomplish this, you only have to do two things.

First, you must **attack the problem, not the person** [4]. Remember, your problem is neither the developers nor their changes. Your problem is Site Up. Do not attack people, they're just trying to do their jobs as best they can. Help them do their jobs better *while* supporting Site Up.

Second, we have to remember that Operations Is an Engineering Problem. If we accept that as an axiom, then we must reject the notion of human gatekeepers as the norm. Remember, we don't want to solve the operational problem using manual effort. We want to solve it as any other software problem, that is, by improving the software.

This isn't to say we don't need gatekeeping, we just don't want human gatekeepers. The second principle introduced earlier in this paper, Empower Developer Ownership, now begins to markedly influence our work.

To get rid of human gatekeeping, we need to mutually agree on what the acceptable standards are for change. Once we have an agreement, we can build automation that enforces the standards we agreed to with our SWEs. This empowers SWEs to do their job without interference, assuming the standards are met.

Let's look at how this might be applied to deployments. If an engineer is told they can't deploy their build, they're going to be unhappy. If a human tells them their build is too slow, they resent the (human) SRE, saying "They won't let me deploy." If, instead, the deployment system tells an engineer they can't deploy because their build violates the agreed upon standards, the engineer will say, "I need to make my build faster so that I can deploy." You've effectively turned a very human problem, "They won't let me deploy," into a simple engineering conversation: "I need to improve the performance of this build so that I can deploy." Well done.

As you near the exit from gatekeeper, all three core principles are now clearly on display: Site Up, Empower Developer Ownership, and Operations Is an Engineering Problem.

Generation 3: The Advocate

The advocate SREs are the ones who lay the foundational relationships required to collaborate well with our partner teams at scale. Their biggest value add is the repair and rebuilding of trusted relationships damaged during the firefighter and gatekeeper generations. How do they accomplish such a feat? They uphold the original three founding principles through their engineering solutions and their interactions with others.

Finally, we can see the list of roles and responsibilities beginning to converge with "monitoring and alerting" now appearing on both lists. By using mutually agreed upon data as a gatekeeper, both SWE and SRE end up losing something significant if the signal quality provided by the gatekeeping data degraded.

Tools SRE, the Advocates

Tools SRE was founded years after our SWE counterparts. That meant we had years of code previously written in order to keep up with the needs of a company that was rapidly growing its business and its engineering body. Reliability was difficult to achieve. Remember, we were also outnumbered 41:1. Even if we tried to use human gatekeeping, it would have failed—what good is building a gate when you don't have the resources to build a fence?

Instead, we tried a different approach. We explained that we didn't want to hold anyone back. To the contrary, we wanted to empower every engineer to do more and spend less of their time fighting fires. Over time, we refined the pitch until it came down to just a few sentences: "Look, we need your help to ensure the products built are reliable and scalable. You can either spend your time helping us fix the outages as they happen or you can create new features. Which do you want it to be?"

Trust Is Everything

As Jeff Weiner, CEO of LinkedIn, has said, "Consistency over time equals trust." This generation of SRE is all about rebuilding trusted relationships by consistently propagating Site Up culture and through building trusted relationships.

First: **be an advocate, make an advocate**. In every conversation or interaction, make sure everyone understands why Site Up matters to them. Be relentless in making this point. At a minimum, they'll eventually agree to help you because of the benefits you've attributed to Site Up and, possibly, even because they believe in the concept. Either way, once the benefits begin to appear, they'll become advocates themselves. Your job will get a bit easier as you end up with an increasingly large number of advocates.

Second: **do not insulate, share pain**. Both the firefighter and the gatekeeper tend to insulate their partner teams from pain indirectly. When the firefighters enter the scene, they help shoulder the burden of incident management. When gatekeepers build their wall, the only pain felt is that of the SREs' change-management process. If both groups are in pain, you can expect easy commitments to end the suffering of both. If only one group is in severe pain and the other feels none, you can expect only disagreements.

Generation 4: The Partner

From this point forward, SRE and SWE need to increasingly function as a single logical unit to do the most possible good. The first step in this alignment is to ensure both SWE and SRE have an equal level of dissatisfaction with the current state of reliability. Another good starting point is to begin joint SRE-SWE planning, if you haven't already. This provides a chance for mutual understanding and will serve to prevent the bulk of mid-quarter priority misalignments.

At first, your team won't be involved in every project; not every SWE team will want to play ball, and that is entirely OK. As the planning cycles go by, it'll become obvious that projects that had both SWE and SRE funding were more reliable, easier to maintain, consumed less time due to scaling problems, and were generally more successful. No one likes missing out on a competitive advantage, and any holdouts will be banging down your door demanding SRE engagement. Once this happens, your planning process gets much easier. You don't need to spend as much time trying to get involved with projects, you just have to agree to work on projects that are going to matter most to your company. Even better, when there are too many "important projects that need SRE partnership," you can go together with your SWE team to justify increased head count, since the value add is apparent to everyone in the conversation.

The roles and responsibilities of SWE-SRE are now beginning to converge rapidly, with many responsibilities being the same. One notable call out: SWE is no longer "an escalation point for SRE." Instead, both groups command a strong understanding of the code base, enough so that you may have a single hybrid on-call rotation comprising both SWE and SRE. Whether a SWE or SRE picks up the phone would simply be a matter of which week you happened to call. Alternatively, if you continued with a traditional tiered escalation format, then an "escalation" to SWE isn't so much a call for a subject matter expert but, instead, for an additional collaborator to help track things down in parallel. Most commonly, we see this as a prelude to a war-room.

A second big departure from prior allocations of roles and responsibilities are the type and scope of our contributions to Site Up. At this point we should be directly improving the products we own or partner on through meaningful engineering contributions.

From this point forward it's all about building reliability and scalability into every product we create or partner with SWE on.

Tools SRE, the Partners

A quick disclaimer: not all of my teams have made it to generation four. For those that have, part of the reason they reached this level was because they had freed up a tremendous amount of their time to focus on the future. We looked for opportunities to allow others to leverage our skills without needing to necessarily talk to us. We created or overhauled services as well as core libraries so that others could be more reliable simply by leveraging our code.

As a team we began to embody "Operations is an engineering problem" by providing leverage to the rest of the company. More importantly, we continued to prioritize the work that would make the most impact for the company, and it naturally led us to more rewarding engineering work.

One Team, One Plan, One Set of Priorities

To advance past Partner SREs, you will need to make two foundational improvements to your team. First, you must participate in **unified SRE-SWE planning**. While overall alignment of teams is mentioned in the "Partner" paragraph, a shared pain needs to be felt by both teams. This helps to drive the unified planning phase. It's important to actually produce a single plan that allocates SRE-SWE resources for the projects where they can add the most value.

Second, you must **leverage the plan to create a unified set of priorities**. This should be a single, stack-ranked set of business priorities that both leadership teams have publicly committed to. These should include priorities such as Talent, Site Up, and Site Secure. By creating the plan and priorities, any engineer in any organization will be able to understand not just what they're doing but why and how they fit into it.

Generation 5: The Engineer

This generation functions as a true north for what SRE should be: fully capable engineers that just so happen to prioritize reliability, scalability, and operability. By this point there should be no further references, save organizational structure itself, to "us vs. them" in policy, day-to-day interaction, or planning. This brings us to the defining characteristic of a generation 5 SRE team: every engineer, regardless of title, organizational affiliation, or the specific job functions of their day-to-day role should be able to answer my favorite question with absolute confidence.

"What is your job?"

And the answer? "My job is to help our company win."

References

[1] Site Up: https://www.linkedin.com/pulse/site-up-benjamin-purgason/.

[2] Every day is Monday in Operations: https://www.linkedin.com/pulse/every-day-monday-operations-benjamin-purgason/.

[3] What gets measured gets fixed: https://www.linkedin.com/pulse/what-gets-measured-fixed-benjamin-purgason/.

[4] Attack the problem, not the person: https://www.linkedin.com/pulse/attack-problem-person-benjamin-purgason/.

Kernel Isolation

From an Academic Idea to an Efficient Patch for Every Computer

DANIEL GRUSS, DAVE HANSEN, AND BRENDAN GREGG

Daniel Gruss (@lavados) is a postdoc fellow at Graz University of Technology. He has been involved with teaching at the university since 2010. In 2015, he demonstrated Rowhammer.js, the first remote fault attack running in a website. He was part of the research team that found the Meltdown and Spectre bugs published in early 2018. daniel.gruss@iaik.tugraz.at

Dave Hansen works in Intel's Open Source Technology Center in Hillsboro, Oregon. He has been involved in Linux for over 15 years and has worked on side-channel hardening, scalability, NUMA, memory management, and many other areas. dave.hansen@intel.com

Brendan Gregg is an industry expert in computing performance and cloud computing. He is a Senior Performance Architect at Netflix, where he does performance design, evaluation, analysis, and tuning. He is the author of Systems Performance published by Prentice Hall, and he received the USENIX LISA Award for Outstanding Achievement in System Administration. Brendan has created performance analysis tools included in multiple operating systems, and visualizations and methodologies for performance analysis, including flame graphs. bgregg@netflix.com

The disclosure of the Meltdown vulnerability [9] in early 2018 was an earthquake for the security community. Meltdown allows temporarily bypassing the most fundamental access permissions before a deferred permission check is finished: that is, the userspace-accessible bit is not reliable, allowing unrestricted access to kernel pages. More specifically, during out-of-order execution, the processor fetches or stores memory locations that are protected via access permissions and continues the out-of-order execution of subsequent instructions with the retrieved or modified data, *even if the access permission check failed*. Most Intel, IBM, and Apple processors from recent years are affected as are several other processors. While AMD also defers the permission check, it does not continue the out-of-order execution of subsequent instructions with data that is supposed to be inaccessible.

KAISER [4, 5] was designed as a software-workaround to the userspace-accessible bit. Hence, KAISER eliminates any side-channel timing differences for inaccessible pages, making the hardware bit mostly superfluous. In this article, we discuss the basic design and the different patches for Linux, Windows, and XNU (the kernel in modern Apple operating systems).

Basic Design

Historically, the kernel was mapped into the address space of every user program, but kernel addresses were not accessible in userspace because of the userspace-accessible bit. Conceptually, this is a very compact way to define two address spaces, one for user mode and one for kernel mode. The basic design of the KAISER mechanism and its derivates is based on the idea that the userspace-accessible bit is not reliable during transient out-of-order execution. Consequently, it becomes necessary to work around this permission bit and not rely on it.

As shown in Figure 1, we try to emulate what the userspace-accessible bit was supposed to provide, namely two address spaces for the user program: a kernel address space with all addresses mapped, protected with proper use of SMAP, SMEP, and NX; and a user address space that only includes a very small fraction of the kernel. This small fraction is required due to the way context switches are defined on the x86 architecture. However, immediately after switching into kernel mode, we switch from the user address space to the kernel address space. Thus, we only have to make sure that read-only access to the small fraction of the kernel does not pose a security problem.

As we discuss in more detail in the performance section, emulating the userspace-accessible bit through this hard split of the address spaces comes with a performance cost.

The global bit. As page table lookups can take much time, a multi-level cache hierarchy (the translation lookaside buffer, TLB) is used to improve the performance. When switching between processes, the TLB has to be cleared at least partially. Most operating systems optimize the performance of context switches by using the global bit for TLB entries that are also valid in the next address space. Consequently, we have to use it with care when implementing

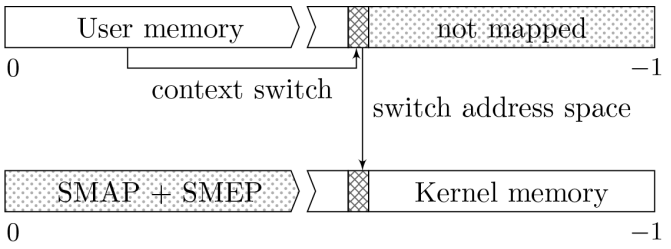


Figure 1: The basic KAISER mechanism

the design outlined above. In particular, marking kernel pages as global (as operating systems previously did) completely undermines the security provided by the KAISER mechanism. Setting the bit to 0 eliminates this problem but leads to another performance reduction.

Naming the Patches. The name KAISER is supposed to be an acronym for *Kernel Address Isolation to have Side channels Efficiently Removed*. It is also a reference to the emperor penguin (German: "Kaiserpinguin"), the largest penguin on earth, with the penguin being the Linux mascot and KAISER being a patch to make Linux stronger. Still under the name KAISER, a significant amount of work was put into the patches that we outline later in this article. Both the authors of the KAISER patch and the Linux kernel maintainers also discussed other names that were deemed less appropriate. Shortly before merging KAISER into the mainline kernel, it was renamed to KPTI, which fits in the typical Linux naming scheme.

Naturally, Microsoft and Apple could not just copy either of the names of the Linux patch. Consequently, they came up with their own names (i.e., KVA Shadow and Double Map) for their own variants of the same idea.

Actual Implementations

The KAISER implementation, developed mainly on virtual machines and a specific off-the-shelf Skylake system, focused on proving that the basic approach was sound. Consequently, reliability and stability that would allow deployment in a real-world environment were out of scope for KAISER. Bringing KAISER up to industry and community standards required ensuring support for all existing hardware and software features and improving its performance and security properties. Furthermore, for Windows and XNU, the patches had to be redeveloped from scratch since their design and implementation is substantially different from Linux.

While the focus on specific machine environments limited the scope of the effort and enabled the implementation of a rapid proof of concept, the environment did not have to cope with certain hardware features like non-maskable interrupts (NMIs), or corner cases when entering or exiting the kernel. These corner cases are rarely encountered in the real world but must still be handled because they might be exploited to cause crashes or escalate privileges (e.g., CVE-2014-4699). NMIs are a particular challenge because they can occur in almost any context, including while the kernel is attempting to transition to or from userspace. For example, before the kernel attempts to return from an interrupt to userspace, it first switches to the user address space. At least one instruction later, it actually transitions to userspace. This means there is always a window where the kernel appears to be running with the "wrong" address space. This can confuse the address-space-switching code, which must use a different method to determine which address space to restore when returning from the NMI.

Linux's KPTI

Much of the process of building on the KAISER proof of concept (PoC) was iterative: find a test that fails or crashes the kernel, debug, fix, check for regressions, then move to the next test. Fortunately, the "x86 selftests" test many infrequently used features, such as the `modify ldt` system call, which is rarely used outside of DOS emulators. Virtually all of these tests existed before KAISER. The key part of the development was finding the tests that exercised the KAISER-impacted code paths and ensuring the tests got executed in a wide variety of environments.

KAISER focused on identifying all of the memory areas that needed to be shared by the kernel and user address spaces and mapping those areas into both. Once it neared being feature-complete and fully functional, the focus shifted to code simplification and improving security.

The shared memory areas were scattered in the kernel portion of the address space. This led to a complicated kernel memory map that made it challenging to determine whether a given mapping was correct, or might have exposed valuable secrets to an application. The solution to this complexity is a data structure called `cpu_entry_area`. This structure maps all of the data and code needed for a given CPU to enter or exit the kernel. It is located at a consistent virtual address, making it simple to use in the restricted environment near kernel entry and exit points. The `cpu_entry_area` is strictly an alias for memory mapped elsewhere by the kernel. This allows it to have hardened permissions for structures such as the "task state segment," mapping them read-only into the `cpu_entry_area` while still permitting the other alias to be used for modifications.

While the kernel does have special "interrupt stacks," interrupts and **system call** instructions still use a process's kernel stack for a short time after entering the kernel. For this reason, KAISER mapped all process kernel stacks into the user address space. This potentially exposes the stack contents to Meltdown,

and it also creates performance overhead in the `fork()` and `exit()` paths. To mitigate both the performance and attack exposure, KPTI added special "entry stacks" to the `cpu_entry_area`. These stacks are only used for a short time during kernel entry/exit and contain much more limited data than the full process stack, limiting the likelihood that they might contain secrets.

Historically, any write to the `CR3` register invalidates the contents of the TLB, which has hundreds of entries on modern processors. It takes a significant amount of processor resources to replace these contents when frequent kernel entry/exits necessitate frequent `CR3` writes. However, a feature on some x86 processors, Process Context Identifiers (PCIDs), provides a mechanism to allow TLB entries to persist over `CR3` updates. This allows TLB contents to be preserved over system calls and interrupts, greatly reducing the TLB impact from `CR3` updates [6]. However, allowing multiple address spaces to live within the TLB simultaneously requires additional work to track and invalidate these entries. But the advantages of PCIDs outweigh the disadvantages, and it continues to be used in Linux both to accelerate KPTI and to preserve TLB contents across normal process context-switching.

Microsoft Windows' KVA Shadow

Windows introduced Kernel Virtual Address (KVA) Shadow mapping [7], which follows the same basic idea as KAISER, with necessary adaptations to the Windows operating system. However, KVA Shadow does not have the goal of ensuring the robustness of KASLR in general, but only seeks to mitigate Meltdown-style attacks. This is a deliberate design choice made to avoid unnecessary design complexity of KVA Shadow.

Similar to Linux, KVA Shadow tries to minimize the number of kernel pages that remain mapped in the user address space. This includes hardware-required per-processor data and special per-processor transition stacks. To not leak any kernel information through these transition stacks, the context switching code keeps interrupts disabled and makes sure not to trigger any kernel traps.

The significant deviations from the basic KAISER approach are in the performance optimizations implemented to make KVA Shadow practical for the huge Windows user base. Similar to Linux, this included the use of PCIDs to minimize the number of implicit TLB flushes. Another interesting optimization is "user/global acceleration" [7]. As stated in the Basic Design section, above, the global bit tells the hardware whether or not to keep TLB entries across the next context switch. While the global bit can no longer be used for kernel pages, Windows now uses it for user pages. Consequently, switching from user to kernel mode does not flush the user TLB entries, although the `CR3` register is switched. This yields a measurable performance advantage. The user pages are not marked global in the kernel address space, and, hence, the corresponding TLB entries are correctly invalidated during the context switch to the next process.

Windows further optimizes the execution of highly privileged tasks by letting them run with a conventional shared address space, which is identical to what the "kernel" address space is now.

With a large number of third-party drivers and software deeply rooted in the system (e.g., anti-viruses), it is not unexpected that some contained code assumes a shared address space. While this first caused compatibility problems, subsequent updates resolved these issues.

Apple XNU's Double Map

Apple introduced the Double Map feature in macOS 10.13.2 (i.e., XNU kernel 4570.31.3, Darwin 17.3.0). Apple used PCIDs on x86 already in earlier macOS versions. However, because mobile Apple devices are also affected by Meltdown, mitigations in the ARMv8-64 XNU kernel were required. Here Apple introduced an interesting technique to leverage the two Translation Table Base Registers (TTBRs) present on ARMv8-64 cores and the Translation Control Register (TCR), which controls how the TTBRs are used in the address translation.

The virtual memory is split into two halves, a userspace half mapped via TTBR0 and a kernel space half mapped via TTBR1. The TCR allows splitting the address space and assigning different TTBRs to disjoint address space ranges. Apple's XNU kernel uses the TCR to unmap the protected part of the kernel in user mode. That is, the kernel space generally remains mapped in every user process, but it's unmapped via the TCRs when leaving the kernel. Kernel parts required for the context switch, interrupt entry code, and data structures are below a certain virtual address and remain mapped. When entering the kernel again, the kernel reconfigures the address space range of TTBR1 via the TCR and, by that, remaps the protected part of the kernel.

The most important advantage of this approach is that the translation tables are not duplicated or modified while running in user mode. Hence, any integrity mechanisms checking the translation tables continue to work.

Performance

When publishing the first unstable PoC of KAISER, the question of performance impact was raised. While the performance impact was initially estimated to be below 5% [5], KAISER showed once more how difficult it is to measure performance in a way that allows comparison of performance numbers. With PCIDs or ASIDs, as now used by all major operating systems, the performance overheads of the different real-world KAISER implementations were reduced, but there are still overheads that may be significant, depending on the workload and the specific hardware. Still, the performance loss for different use cases,

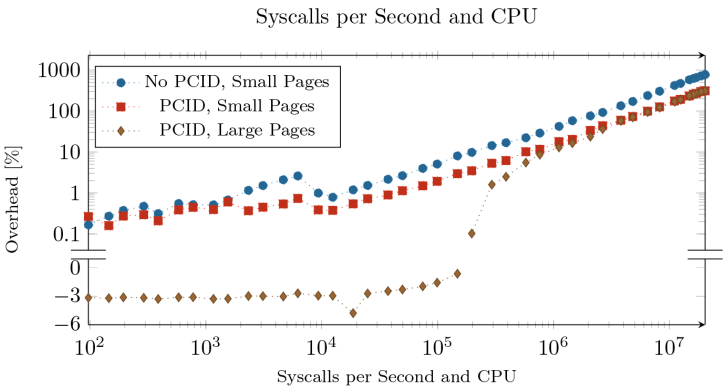


Figure 2: The runtime overhead for different workloads with different KPTI configurations [2]. The overhead increases with the system call rate due to the additional TLB flushes and `CR3` manipulations during context switches.

macrobenchmarks, and microbenchmarks varies between −5% and 800%. One reason is the increase in TLB flushes, especially on systems without PCID support, as well as extra cycles for `CR3` manipulation. More indirect is the increase in TLB pressure, caused by the additional TLB entries due to the large number of duplicated page table entries. CPU- or GPU-intense workloads that trigger a negligible number of context switches, and thus a negligible number of TLB flushes and `CR3` manipulations, are mostly unaffected.

The different implementations of KAISER have different optimizations. In this performance analysis, we focus on Linux (i.e., KPTI). However, the reported numbers are well aligned with reports of performance overheads on other operating systems [1, 7].

We explore the overheads for different system call rates [2] by timing a simultaneous working-set walk, as shown in Figure 2.

Without PCID, at low system call rates, the overheads were negligible, as expected: near 0%. At the other end of the spectrum, at over 10 million system calls per second per CPU, the overhead was extreme: the benchmark ran over 800% slower. While it is unlikely that a real-world application will come anywhere close tonthis, it still points out a relevant bottleneck that has not existed without the KAISER patches. For perspective, the system call rates for different cloud services at Netflix were studied, and it was found that database services were the highest, with around 50,000 system calls per second per CPU. The overhead at this rate was about 2.6% slower.

While PCID support greatly reduced the overhead, from 2.6% to 1.1%, there is another technique to reduce TLB pressure: large pages. Using large pages reduces the overhead for our specific benchmark so much that for any real-world system call rate there is a performance gain.

Another interesting observation while running the microbenchmarks was an abrupt drop in performance overhead, depending on the hardware and benchmark, at a syscall rate of 5000. While this was correlated with the last-level cache hit ratio, it is unclear what the exact reason is. One suspected cause is a sweet spot in either the amount of memory touched or the access pattern between two system calls, where, for example, the processor switches the cache eviction policy [3].

With PCID support and using large pages when possible, one can conclude that the overheads of Linux's KPTI and other KAISER implementations are acceptable. Furthermore, rudimentary performance tuning (i.e., analyzing and reducing system call and context switch rates) may yield additional performance gains.

Outlook and Conclusion

With KAISER and related real-world patches, we accepted a performance overhead to cope with the insufficient hardware-based isolation. While more strict isolation can be a more resilient design in general, it currently functions as a workaround for a specific hardware bug. However, there are more Meltdown-type hardware bugs [8, 10], causing unreliable permission checks during transient out-of-order execution, for other page table bits. Mitigating them requires additional countermeasures beyond KAISER. For now, KAISER will still be necessary for commodity processors.

Acknowledgments

We would like to thank Matt Miller, Jon Masters, and Jacques Fortier for helpful comments on early drafts of this article.

References

[1] fG!, "Measuring OS X Meltdown Patches Performance," January 2018: https://reverse.put.as/2018/01/07/measuring-osx-meltdown-patches-performance/.

[2] B. Gregg, "KPTI/KAISER Meltdown Initial Performance Regressions," 2018: http://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html.

[3] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '16)*, pp. 300–321: https://gruss.cc/files/rowhammerjs.pdf.

[4] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR," in 23rd ACM Conference on Computer and Communications Security (CCS, 2016): https://gruss.cc/files/prefetch.pdf.

[5] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR Is Dead: Long Live KASLR," in *Proceedings of the 9th International Symposium on Engineering Secure Software and Systems (ESSoS '17)*, pp.161–176: https://gruss.cc/files/kaiser.pdf.

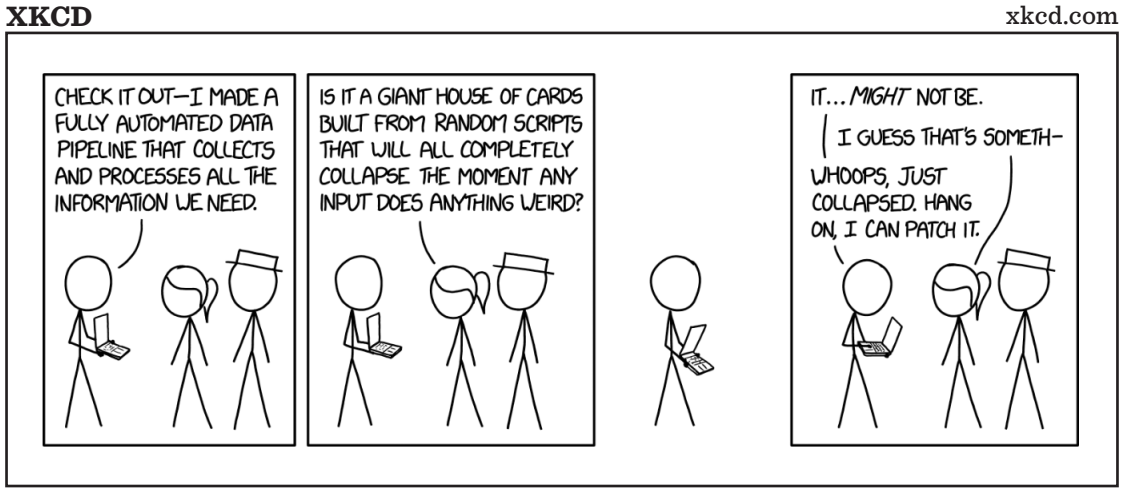
[6] D. Hansen, "KAISER: Unmap Most of the Kernel from User-space Page Table," Linux Kernel Mailing List, October 2017: https://lkml.org/lkml/2017/10/31/884.

[7] K. Johnson, "KVA Shadow: Mitigating Meltdown on Windows," March 2018: https://blogs.technet.microsoft.com/srd/2018/03/23/kva-shadow-mitigating-meltdown-on-windows/.

[8] V. Kiriansky and C. Waldspurger, "Speculative Buffer Overflows: Attacks and Defenses," *arXiv:1807.03757*, 2018.

[9] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security '18)*, pp. 973–990: https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-lipp.pdf.

[10] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security '18)*, pp. 991–1008: https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-van_bulck.pdf.



The Secure Socket API

TLS as an Operating System Service

MARK O'NEILL, KENT SEAMONS, AND DANIEL ZAPPALA

Mark O'Neill is a PhD candidate in computer science at Brigham Young University and currently working at ManTech International. His research interests include security, networking, and artificial intelligence. His dissertation is an effort to solve modern problems in TLS by leveraging operating system and administrator control. When he's not working on research, you can find him tinkering with robots and playing StarCraft 2. Mark@markoneill.name

Kent Seamons is a Professor of Computer Science at Brigham Young University and Director of the Internet Security Research Lab. His research interests are in usable security, privacy, authentication, and trust management. His research has been funded by NSF, DHS, DARPA, and industry. He is also a co-inventor on four patents in the areas of automated trust negotiation, single sign-on, and security overlays. seamons@cs.byu.edu

Daniel Zappala is an Associate Professor of Computer Science at Brigham Young University and the Director of the Internet Research Lab. His research interests include network security and usable security. His research has been funded regularly by NSF and DHS. Daniel is an open source enthusiast and has used Linux and Emacs for about 30 years. zappala@cs.byu.edu

TLS APIs are often complex, leading to developer mistakes. In addition, even with well-written applications, security administrators lack control over how TLS is used on their machines and don't have the ability to ensure applications follow best practices. Our solution is to provide a Secure Socket API that is integrated into the well-known POSIX sockets API. This is both simple for developers to use and allows system administrators to set device policy for TLS. In this article, we both explain and demonstrate how the Secure Socket API works.

Transport Layer Security (TLS) is the most popular security protocol used on the Internet. Proper use of TLS allows two network applications to establish a secure communication channel between them. However, improper use can result in vulnerabilities to various attacks. Unfortunately, popular security libraries, such as OpenSSL and GnuTLS, while feature-rich and widely used, have long been plagued by programmer misuse. The complexity and design of these libraries can make them hard to use correctly for application developers and even security experts. For example, Georgiev et al. find that the "terrible design of [security library] APIs" is the root cause of authentication vulnerabilities [1]. Significant efforts to catalog developer mistakes and the complexities of modern security APIs have been published in recent years. As a result, projects have emerged that reduce the size of security APIs (e.g., *libtls* in LibreSSL), enhance library security [2], and perform certificate validation checks on behalf of vulnerable applications [3, 4]. A common conclusion of these works is that TLS libraries need to be redesigned to be simpler for developers to use securely.

A related problem is that the reliance on application developers to implement security inhibits the control administrators have over their own machines. For example, administrators cannot currently dictate what version of TLS, which ciphersuites, key sizes, etc. are used by applications they install. This coupling of application functionality with security policy can make otherwise desirable applications unadoptable by administrators with incompatible security requirements. This problem is exacerbated when security flaws are discovered in applications and administrators must wait for security patches from developers, which may not ever be provided.

The synthesis of these two problems is that developers lack a common, usable security API, and administrators lack control over secure connections. To address these issues, we present the Secure Socket API (SSA), a TLS API that leverages the existing standard POSIX socket API. This reduces the TLS API to a handful of functions that are already offered to and used by network programmers, effectively making the TLS API itself nearly transparent. This drastically reduces the code required to use TLS, as developers merely select TLS as if it were a built-in protocol, such as TCP or UDP. Moreover, our implementation of this API enables administrators to configure TLS policies system-wide and to centrally update all applications using the API.

Secure Socket API Design

Under the POSIX socket API, developers specify their desired protocol using the last two parameters of the `socket` function, which specify the type of protocol (e.g., `SOCK_DGRAM`, `SOCK_STREAM`) and optionally the protocol itself (e.g., `IPPROTO_TCP`). Corresponding network operations such as `connect`, `send`, and `recv` then use the selected protocol in a manner transparent to the developer. In designing the SSA, we sought to cleanly integrate TLS into this API. Our design goals are as follows:

1. Enable developers to use TLS through the existing set of functions provided by the POSIX socket API without adding any new functions or changing function signatures. Modifications to the API are acceptable only in the form of new *values* for existing parameters.
2. Support direct administrator control over the parameters and settings for TLS connections made by the SSA. Applications should be able to increase, but not decrease, the security preferred by the administrator.
3. Export a minimal set of TLS options to applications that allow general TLS use and drastically reduce the amount of functions in contemporary TLS APIs.
4. Facilitate the adoption of the SSA by other programming languages, easing the security burden on language implementations and providing broader security control to administrators.

To inform the design of the SSA, we first analyzed the OpenSSL API and its use by popular software packages. This included automated and manual assessment of 410 Ubuntu packages using TLS in client and server capacities, and assessment of the OpenSSL API itself. More details regarding our methods and results for this analysis are available at https://owntrust.org.

The API

Under the Secure Socket API, all TLS functionality is built directly into the POSIX socket API. The POSIX socket API was derived from Berkeley sockets and is meant to be portable and extensible, supporting a variety of network communication protocols. Under our SSA extension, developers select TLS by specifying `IPPROTO_TLS` as the protocol in `socket`. Applications send and receive data using standard functions such as `send` and `recv`, which will be encrypted and decrypted using TLS, just as network programmers expect their data to be placed inside and removed from TCP segments under `IPPROTO_TCP`. To transparently employ TLS in this fashion, other functions of the POSIX socket API have specialized TLS behaviors under `IPPROTO_TLS` as well. In particular, `getsockopt` and `setsockopt` are used for developer configuration. A complete listing of the behaviors of the POSIX socket functions and the TLS socket options are provided in our recent paper [5].

To avoid developer misuse of TLS, the SSA is responsible for automatic management of various TLS parameters and settings, including selection of TLS versions, ciphersuites and extensions, and validation of certificates. All of these are subject to a system configuration policy with secure defaults, and customization options are exported to system administrators and developers.

To offer concrete examples of SSA use, we show code for a simple client and server below. Both the client and the server create a socket with the `IPPROTO_TLS` protocol. The client uses the standard `connect` function to connect to the remote host, also employing a new `AF_HOSTNAME` address family to indicate which hostname it wishes to connect to. In this case, the `connect` function performs the necessary host lookup and performs a TLS handshake with the resulting address. Alternatively, the client could have specified the hostname via a new socket option and called `connect` using traditional `INET` address families. The former method obviates the need for developers to explicitly call `gethostbyname` or `getaddrinfo`, which further simplifies their code. Either way, the SSA uses the provided hostname for certificate validation and the Server Name Indication extension to TLS. Later, the client uses `send` to transmit a plaintext HTTP request to the server, which is encrypted by the SSA before transmission. The response received is also decrypted by the SSA before placing it into the buffer provided by `recv`.

In the server case, the application binds and listens on port 443. Before it calls `listen`, it uses two calls to `setsockopt` to provide the location of its private key and certificate chain file to be used for authenticating itself to clients during the TLS handshake. Afterward, the server iteratively handles requests from incoming clients, and the SSA performs a TLS handshake with clients transparently. As with the client case, calls to `send` and `recv` have their data encrypted and decrypted in accordance with the TLS session, before they are delivered to their destinations.

```
/* Use hostname address family */
struct sockaddr_host addr;
addr.sin_family = AF_HOSTNAME;
strcpy(addr.sin_addr.name, "www.example.com");
addr.sin_port = htons(443);

/* Request a TLS socket (instead of TCP) */
fd = socket(PF_INET, SOCK_STREAM, IPPROTO_TLS);
/* TLS Handshake (verification done for us) */
connect(fd, &addr, sizeof(addr));

/* Hardcoded HTTP request */
char http_request[] = "GET / HTTP/1.1\r\n..."
char http_response[2048];
memset(http_response, 0, 2048);
/* Send HTTP request encrypted with TLS */
```

```
send(fd,http_request,sizeof(http_request)-1,0);
/* Receive decrypted response */
recv(fd, http_response, 2047, 0);
/* Shutdown TLS connection and socket */
close(fd);
return 0;
```

Listing 1: A simple HTTPS client example under the SSA. Error checks and some trivial code are removed for brevity.

```
/* Use standard IPv4 address type */
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY;
addr.sin_port = htons(443);

/* Request a TLS socket */
fd = socket(PF_INET, SOCK_STREAM, IPPROTO_TLS);
bind(fd, &addr, sizeof(addr));
/* Assign certificate chain */
setsockopt(fd, IPPROTO_TLS,
               TLS_CERTIFICATE_CHAIN,
               CERT_FILE, sizeof(CERT_FILE));
/* Assign private key */
setsockopt(fd, IPPROTO_TLS,
               TLS_PRIVATE_KEY,
               KEY_FILE, sizeof(KEY_FILE));
listen(fd, SOMAXCONN);

while (1) {
  struct sockaddr_storage addr;
  socklen_t addr_len = sizeof(addr);
  /* Accept new client and do TLS handshake
  using cert and keys provided */
  int c_fd = accept(fd, &addr, &addr_len);
  /* Receive decrypted request */
  recv(c_fd, request, BUFFER_SIZE, 0);
  handle_req(request, response);
  /* Send encrypted response */
  send(c_fd, response, BUFFER_SIZE, 0);
  close(c_fd);
}
```

Listing 2: A simple server example under the SSA. Error checks and some trivial code are removed for brevity.

Administrator Options

Reflecting our second goal, administrator control over TLS parameters, the SSA gives administrators a protected configuration file that allows administrators to indicate their preferences for TLS versions, ciphersuites, certificate validation methodologies, extensions, and other TLS settings. These settings are applied to all TLS connections made with the SSA on the machine. However, additional configuration profiles can be created or installed by the administrator for specific applications that override global settings.

Our definition of administrators includes both power users as well as operating system vendors, who may wish to provide strong default policies for their users.

Developer Options

The `setsockopt` and `getsockopt` POSIX functions provide a means to support additional settings in cases where a protocol offers more functionality than can be expressed by the limited set of principal functions. Linux, for example, supports 34 TCP-specific socket options to customize protocol behavior. Arbitrary data can be transferred to and from the API implementation using `setsockopt` and `getsockopt`, because they take a generic pointer and a data length (in bytes) as parameters, along with an `optname` constant identifier. Adding a new option can be done by merely defining a new `optname` constant to represent it and adding appropriate code to the implementation of `setsockopt` and `getsockopt`.

In accordance with this standard, the SSA adds a few options for `IPPROTO_TLS`. These options include setting the remote hostname, specifying a certificate chain or private key, setting a session TTL, disabling a cipher, requesting client authentication, and others. A full list is given in our recent paper [5]. Our specification of TLS options reflects a minimal set of recommendations gathered from our analysis of existing TLS use by applications, in keeping with our third design goal.

Porting Applications to the SSA

We modified the source code of four network programs to use the SSA for their TLS functionality. Two of these already used OpenSSL for their TLS functionality, and two were not built to use TLS at all. Table 1 summarizes the results of these efforts.

Both the command-line `wget` web client and the `lighttpd` web server required fewer than 20 lines of source code (Table 1), and each application was modified by a developer who had no prior experience with the code of these tools, the SSA, or OpenSSL. In addition, the modifications made it possible to remove thousands of lines of existing code. In porting these applications, most of the time spent was used to become familiar with the source code and remove OpenSSL calls.

We also modified an in-house web server and the `netcat` utility, neither of which previously supported TLS. The web server required modifying only one line of code—the call to `socket` to use `IPPROTO_TLS` on its listening socket. Under these circumstances, the certificate and private key used are from the SSA configuration. However, these can be specified by the application with another four lines of code to set the private

Program	LOC Modified	LOC Removed	Time Taken
wget	15	1,020	5 hrs.
lighttpd	8	2,063	5 hrs.
ws-event	5	0	5 min.
netcat	5	0	10 min.

Table 1: Summary of code changes required to port a sample of applications to use the SSA. `wget` and `lighttpd` used existing TLS libraries, `ws-event` and `netcat` were not originally TLS-enabled.

key and certificate chain and check for corresponding errors. The TLS upgrade for `netcat` for both server and client connections required modifying five lines of code. In both cases, TLS upgrades required less than 10 minutes.

Language Support

One of the benefits of using the POSIX socket API as the basis for the SSA is that it is easy to provide SSA support to a variety of languages, which is in line with our fourth design goal. This benefit accrues if an implementation of the SSA instruments the POSIX socket functionality in the kernel through the system-call interface. Any language that uses the network must interface with network system calls, either directly or indirectly. Therefore, given an implementation in the kernel, it is trivial to add SSA support to other languages.

To illustrate this benefit, we have added SSA support to three additional languages beyond C/C++: Python, PHP, and Go. Supporting these first two languages merely required making their corresponding interpreters aware of the additional constant values used in the SSA, such as `IPPROTO_TLS`. Since Go uses system calls directly and exports its own wrapper for these, we followed the same pattern by creating new wrappers for SSA functionality, which required fewer than 50 lines of code.

Implementation

We have developed a loadable Linux kernel module that implements the Secure Socket API. Source code is available at https://owntrust.org. A high-level view of a typical network application using a security library for TLS is shown in Figure 1. The application links to the security library, such as OpenSSL or GnuTLS, and then uses the POSIX socket API to communicate with the network subsystem in the kernel, typically using a TCP socket.

A corresponding diagram, Figure 2, illustrates how our implementation of the SSA compares to this normal usage. We split our SSA implementation into two parts: a kernel component and a userspace encryption daemon. At a high-level, the kernel component is responsible for registering all `IPPROTO_TLS` functionality with the kernel and maintaining state for each TLS

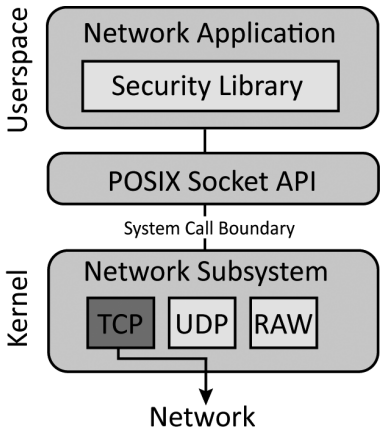


Figure 1: Data flow for traditional TLS library by network applications. The application shown is using TCP.

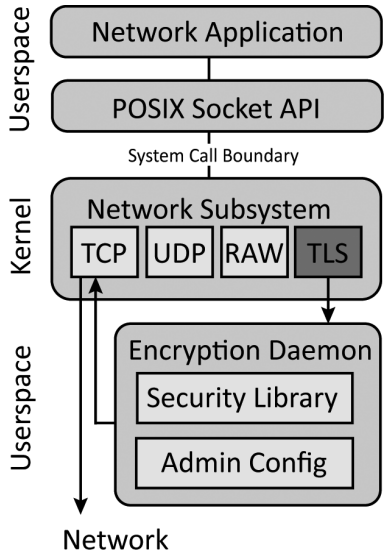


Figure 2: Data flow for SSA usage by network applications. The application shown is using the TLS, which uses TCP internally for connection-based `SOCK_STREAM` sockets.

socket. The kernel component offloads the tasks of encryption and decryption to the encryption daemon, which uses OpenSSL and obeys administrator preferences.

Note that our prototype implementation moves the use of a security library to the encryption daemon. The application interacts only with the POSIX socket API, and the encryption daemon establishes TLS connections, encrypts and decrypts data, implements TLS extensions, and so forth. The daemon uses administrator configuration to choose which TLS versions, ciphersuites, and extensions to support.

Alternative Implementations

POSIX is a set of standards that defines an OS API—the implementation details are left to system designers. Accordingly, our presentation of the SSA with its extensions to the existing POSIX socket standard and related options is separate from the presented implementation. While our implementation leveraged a userspace encryption daemon, other architectures are possible. We outline two of these:

- Userspace only: The SSA could be implemented as a userspace library that is either statically or dynamically linked with an application, wrapping the native socket API. Under this model the library could request administrator configuration from default system locations to retain administrator control of TLS parameters. While such a system sacrifices the inherent privilege separation of the system-call boundary and language portability, it would not require that the OS kernel explicitly support the API.
- Kernel only: Alternatively, an implementation could build all TLS functionality directly into the kernel, resulting in a pure kernel solution. This idea has been proposed within the Linux community [6] and gained some traction in the form of patches that implement individual cryptographic components. Some performance gains in TLS are also possible in this space. Such an implementation would provide a back end for SSA functionality that required no userspace encryption daemon.

Discussion

Our work explores a TLS API conforming to the POSIX socket API. We reflect now on the general benefits of this approach and the specific benefits of our implementation.

By conforming to the POSIX API, using TLS becomes a matter of simply specifying TLS rather than TCP during socket creation and setting a small number of options. All other socket calls remain the same, allowing developers to work with a familiar API. Porting insecure applications to use the SSA takes minutes, and refactoring secure applications to use the SSA instead of OpenSSL takes a few hours and removes thousands of lines of code. This simplified TLS interface allows developers to focus on the application logic that makes their work unique rather than spending time implementing standard network security.

Because our SSA design moves TLS functionality to a centralized service, administrators gain the ability to configure TLS behavior on a system-wide level, and tailor settings of individual applications to their specific needs. Default configurations can be maintained and updated by OS vendors, similar to Fedora's CryptoPolicy [7]. For example, administrators can set preferences for TLS versions, ciphersuites, and extensions, or automatically upgrade applications to TLS 1.3 without developer patches.

By implementing the SSA with a kernel module, developers who wish to use it do not have to link with any additional userspace libraries. With small additions to libc headers, C/C++ applications can use `IPPROTO_TLS`. Other languages can be easily modified to use the SSA, as demonstrated with our efforts to add support to Go, Python, and PHP.

Adding TLS to the Linux kernel as an Internet protocol allows the SSA to leverage the existing separation of the system call boundary. Due to this, privilege separation in TLS usage can be naturally achieved. For example, administrators can store private keys in a secure location inaccessible to applications. When applications provide paths to these keys using `setsockopt` (or use them from the SSA configuration), the SSA can read these keys with its elevated privilege. If the application becomes compromised, the key data (and master secret) remain outside the address space of the application.

Conclusion

We feel that the POSIX socket API is a natural fit for a TLS API and hope to see it advanced through its use, new implementations, and standardization. We hope to encourage community involvement to further refine our implementation and help develop support in additional operating systems. For source code and documentation, please visit https://owntrust.org. For a more in-depth look at the SSA, see our paper presented at USENIX Security 2018 [5].

Acknowledgments

This work was supported in part by the National Science Foundation under Grant No. CNS-1528022 and the Department of Homeland Security under contract number HHSP233201600046C.

References

[1] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS '12)*, pp. 38–49: http://www.cs.utexas.edu/~shmat/shmat_ccs12.pdf.

[2] L. S. Amour and W. M. Petullo, "Improving Application Security through TLS-Library Redesign," in *Security, Privacy, and Applied Cryptography Engineering (SPACE)*, Springer, 2015, pp. 75–94.

[3] A. Bates, J. Pletcher, T. Nichols, B. Hollembaek, D. Tian, K. R. Butler, and A. Alkhelaifi, "Securing SSL Certificate Verification through Dynamic Linking," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS '14)*, pp. 394–405.

[4] M. O'Neill, S. Heidbrink, S. Ruoti, J. Whitehead, D. Bunker, L. Dickinson, T. Hendershot, J. Reynolds, K. Seamons, and D. Zappala, "TrustBase: An Architecture to Repair and Strengthen Certificate-Based Authentication," in *Proceedings of the 26th USENIX Security Symposium (USENIX Security '17)*: https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-oneill.pdf.

[5] M. O'Neill, S. Heidbrink, J. Whitehead, T. Perdue, L. Dickinson, T. Collett, N. Bonner, K. Seamons, and D. Zappala, "The Secure Socket API: TLS as an Operating System Service," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security '18)*, pp. 799–816: https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-o_neill.pdf.

[6] J. Edge, "TLS in the Kernel," LWN.net: https://lwn.net/Articles/666509/; accessed: December 15, 2017.

[7] N. Mavrogiannopoulos, M. Trmac, and B. Preneel, "A Linux Kernel Cryptographic Framework: Decoupling Cryptographic Keys from Applications," in *Proceedings of the 27th ACM Symposium on Applied Computing (SAC '12)*, pp. 1435–1442: https://core.ac.uk/download/pdf/34512267.pdf.

Strings Considered Harmful

ERIK POLL

Erik Poll is Associate Professor at Radboud University Nijmegen. His research focuses on the use of formal methods to analyze the security of systems, especially of the software involved. Application areas that provided case studies for his research include smart cards, security protocols, payment systems, and smart grids.
erikpoll@cs.ru.nl

Buggy parsers are an important source of security vulnerabilities in software: many attacks use malicious inputs designed to exploit parser bugs. Some security flaws in input handling do not exploit parser bugs, but exploit correct—albeit unexpected—parsing of inputs caused by the forwarding of inputs between systems or components. This article, based on an earlier workshop paper [11], discusses anti-patterns and remedies for this type of flaw, including the anti-pattern mentioned in the title.

LangSec and Parsing Flaws

The LangSec paradigm [3, 8] gives good insights into the root causes behind the majority of security problems in software, which are problems in handling inputs. It recognizes that the input languages used play a central role. More particularly, it identifies the following root causes for security problems: the sheer number of input languages that a typical application handles; their complexity; their expressivity; the lack of clear, unambiguous specifications of these languages; and the handwritten parser code (which often mixes the parsing and subsequent processing, in so-called shotgun parsers, where input is parsed piecemeal and in various stages scattered throughout the code). All this leads to parser bugs, with buffer overflows in processing file formats such as Flash or network packets for protocols such as TLS as classic examples. It can also lead to differences between parsers that can be exploited, with, for example, variations in interpreting X509 certificates [6] as a result. In all cases, these bugs provide weird behavior—a so-called weird machine, in LangSec terminology—that attackers can try to abuse.

Much of the LangSec research therefore concentrates on preventing parsing flaws: by having simpler input languages; by having clearer, formal specs for them; and by generating parser code to replace handwritten parsers, using tools such as Hammer (https://github.com/UpstandingHackers/hammer), Nail [2], or protocol buffers (https://developers.google.com/protocol-buffers). For a more thorough discussion of LangSec anti-patterns and remedies, see [8].

Forwarding Flaws

However, not all input-related security flaws are due to buggy parsing. A large class of flaws involves the careless *forwarding* of malicious input by some front-end application to some back-end service or component where the input is correctly—but unexpectedly and unintentionally—parsed and processed (Figure 1). Classic examples are format string attacks, SQL injection, command injection, path traversal, and XSS.

In the case of a SQL injection attack, the web server is the front end and SQL database is the back end. In the case of a format string attack, the back end is not a separate system like a database but consists of the C system libraries. In an XSS attack, the web browser is the back end and the web server the front end; this can get more complex, e.g., in reflected XSS attacks, where malicious input is forwarded back and forth between browser and server before finally doing damage in the browser.

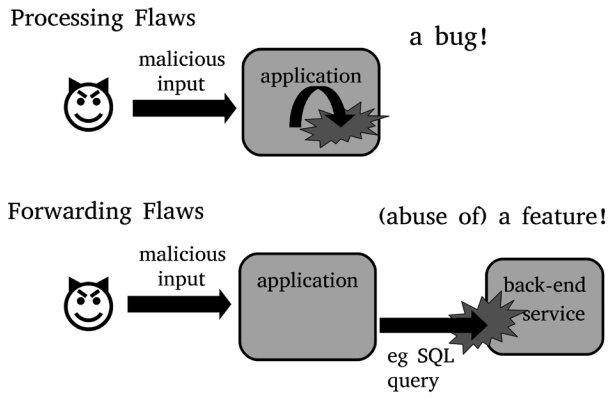


Figure 1: Processing vs. forwarding flaws

Forwarding attacks do not (necessarily) exploit parser bugs: the back-end service, say the SQL database, may well parse and process its inputs correctly. The problem is not that this SQL functionality is buggy but rather that it can be triggered by attackers that feed malicious input to the front end. Unlike attacks that exploit parsing bugs, where attackers abuse weird behavior introduced accidentally, attackers here abuse functionality that has been introduced deliberately but which is exposed accidentally.

Forwarding flaws are also called *injection flaws*, e.g., in the OWASP Top Ten, where they occupy the first spot. We prefer the term "forwarding flaws" because in some sense all input attacks are injection attacks; the forwarding aspect is what sets these input attacks apart from the others.

Input or Output Problem?

Forwarding flaws involve two systems—a front-end application and a back-end service—and both input and output, since the malicious input to the front end ultimately ends up as output from the front end to the back end. This not only introduces the question of *how* to tackle this problem but also the question of *where* to tackle it. Should the front end prevent untrusted input from ending up in the back end, and if so, should it sanitize data at the program point where the data is output to the back end, or should it do that earlier, at the program point where it received the original malicious input? Or should the back end simply not provide such a dangerously powerful interface to the front end? We can recognize anti-patterns that can lead to forwarding flaws, or to bad solutions in tackling them, as well as some remedies to address them in a structural way.

Anti-Pattern: Input Sanitization

There are very different ways to treat invalid or dangerous input. It can be completely *rejected* or it can be *sanitized*. Sanitization can be done by escaping or encoding dangerous characters to make them harmless, typically by adding backslashes or quotes, or by stripping dangerous characters and keywords. A complication here is that ideally one would like to validate input at the point where the input enters an application, because at that program point it is clear whether such input is untrusted or not. However, at that point we may not yet know in which context the input will be used, and different contexts may require different forms of escaping. For example, the same input string could be used in a path name, a URL, an SQL query, and in HTML text, and these contexts may require different forms of escaping.

Because escaping is context-sensitive in this way, it is well known that using one generic operation to sanitize all input is highly suspect, as one generic operation is never going to provide the right escaping for a variety of back-end systems. This also means that *input* sanitization, i.e., sanitization at the point of input rather than at the point of output, is suspect since the context typically is not known there.

The classic example here is the infamous PHP magic quotes setting, which caused all incoming data to be automatically escaped. It took a while to reach consensus that this was a bad idea: magic quotes were deprecated in PHP 5.3.0 and finally removed in PHP 5.4.0 in 2012.

Anti-Pattern: String Concatenation

A well-known anti-pattern in forwarding attacks is the use of string concatenation. Concatenating several pieces of data, some of which are user input, and feeding the result to an API call, as is done in dynamic SQL queries, is the classic recipe for disaster.

Given that the LangSec approach highlights the importance of parsing, it is interesting to note that string concatenation is a form of *unparsing*. Indeed, the whole problem in forwarding attacks is that the back-end service parses strings in a different way than the front end intended.

Anti-Pattern: Strings

We would argue that a more general anti-pattern than the use of string concatenation for dynamic queries is the use of strings at all. There are several reasons why heavy use of strings can spell trouble:

- *Strings can be used for all sorts of data*: usernames, email addresses, file names, URLs, fragments of HTML, pieces of JavaScript, etc. This makes it a very useful and ubiquitous data type, but it also causes confusion: from a generic string type, we cannot tell what the intended use of the data is or, for instance, whether it has been escaped or validated.
- *Strings are by definition unparsed data*. So if a program uses strings, it typically has to do parsing at runtime. Much of this parsing could be avoided if more structured forms of data were used instead. The extra parsing creates a lot of room for trouble, especially in combination with the point above, which tells us that the same string might end up in different parsers.

The shotgun parsing that the LangSec literature warns against, where partial and piecemeal parsing is spread throughout an application, also inevitably involves the use of strings, namely for passing around unparsed fragments of input.

- *String parameters often bring unwanted expressivity*. Interfaces that take strings as a parameter often introduce a whole new language (e.g., HTML, SQL, the language of pathnames, OS shell commands, or format strings), with all sorts of expressive power that may not be necessary and which only provides a security risk.

In summary, the problem with strings is that it is one generic data type, for completely unstructured data, and for many kinds of data, obscuring the fact that there are many different languages involved, possibly very expressive ones, each with its own interpretation. Of course, others have warned about the use of strings before, e.g., [1].

The disadvantages above apply equally to *char* pointers in C, *string* objects in C++, or *String* objects in Java. Of course, for security it is better to use memory-safe, type-safe, or immutable and hence thread-safe data types rather than more error-prone versions.

Remedy: Reducing Expressive Power

An obvious way to prevent forwarding flaws, or at least mitigate the potential impact, is to reduce the expressive power exposed by the interface between the front end and the back end.

For SQL injections this can be done with parameterized queries (or with stored procedures, provided that these are safe). The use of parameterized queries reduces the expressive power of the interface to the back-end database, and it reduces the amount of runtime parsing. So clearly this mechanism involves key aspects highlighted in the LangSec approach, namely expressivity and parsing.

Remedy: Types to Distinguish Languages and Formats

Different types in the programming language can be used to distinguish the different languages or data formats that an application handles. These types reduce ambiguity: ambiguity about the intended use of data and ambiguity about whether or not it has been parsed and validated. This then also reduces the scope for unintended interactions.

Note that standard security flaws such as double decoding bugs or problems with null terminator characters in strings also indicate confusion about data representations that use of a type system could—and should—prevent.

For example, an application could use different types for URLs, usernames, email addresses, file names, and fragments of HTML. The type checker can then complain when a username is included inside HTML and force the programmer to add an escaping function to turn a username into something that is safe to render as HTML.

For data that is really just a string, like a username, one might use a struct or object with a single string field. (Type annotations, as exist in Java for example, could also be used to distinguish different kinds of strings [10].) However, for structured data, say a URL, the type would ideally not just be a wrapper for the unparsed string but, instead, an object or struct with fields and/or methods for the different components, such as the protocol, domain, path, etc., to reduce the amount of code that handles data in unparsed form.

When data is forwarded between components inside an application or between applications written in the same programming language, data can be forwarded "as is," with all type information preserved and without the need for any (un)parsing. However, when data is exchanged with external systems, it may have to be serialized and deserialized. Here the risk of parsing bugs re-emerges, and the classic LangSec strategies to avoid these should be followed by, ideally, generating the code for (de) serialization from a formal spec.

Remedy: Types to Distinguish Trust Levels

Types can also be used for different *trust levels*. This then allows information flows from untrusted sources in the code to be traced and restricted. An example would be to use different types for trusted string constants hard coded in the application and for untrusted (aka tainted) strings that stem from user input to then only allow the former to be used as parameters to certain security-sensitive operations.

Efforts at Google to prevent XSS in web applications [7] use types in this way (https://github.com/google/safe-html-types/blob/master/doc/index.md). For instance, it uses different types to distinguish

- URLs that can be used in HTML documents or as arguments to DOM APIs, but not in contexts where this would lead to the referred resource being executed as code, and
- more trusted URLs that can also be used to fetch JavaScript code (e.g., by using them as `scr` of a `script` element).

A more recent proposal to combat XSS, called Trusted Types (https://github.com/WICG/trusted-types), extends Google's approach to fighting XSS using types by replacing all string-based APIs of the DOM with typed APIs. This approach tackles the root cause that makes it so hard to deal with the more complicated forms of (DOM-based) XSS: the ubiquitous use of string parameters in the DOM APIs.

The two ways to use types—to distinguish different kinds of data or different trust levels—are of course orthogonal and can be combined. Using trust levels for security goes back to work on information flow in the 1970s [4]. It has been used in many static and dynamic analyses over the years, including many security type systems and source code analyzers, and has given rise to a whole research field of language-based information-flow security [12].

Clearly, the notion of information flow goes to the heart of what forwarding flaws are about. A type system for information flow is precisely what can solve the fundamental problem of keeping track of whether data has been or should be validated or sanitized. Instead of just tracking untrusted data to prevent malicious input from being forwarded to places where it can do damage, type systems for information flow can also be used to track confidential information to prevent information leaks (see, e.g., [5]).

Beyond Types: Programming Language Support

Instead of using the type system of a programming language to distinguish the different languages and data formats that an application has to handle, one can go one step further and provide native support for them in the programming language. This approach is taken in Wyvern [9], called a type-specific programming language by the designers.

An added advantage is that the programming language can provide more convenient syntax to tempt programmers away from convenient but insecure coding styles. For example, it can provide syntax for safe parameterized SQL queries that is just as convenient as the unsafe dynamic SQL queries, with the nice infix notation for string concatenation that programmers like. The idea is that a type-specific programming language allows any number of input and output languages to be embedded. In the original use case of web programming, the embedded languages would include SQL and HTML. These languages then show up as different types in the programming languages, with all the convenient syntax support.

Conclusion

Many of the remedies suggested by the LangSec paradigm focus on eradicating parser bugs: e.g., insisting on clear specifications of input languages, keeping these languages simple, generating parsers from formal specs instead of handrolling written parser code, and separating parsing and subsequent processing in an attempt to avoid shotgun parsers.

However, these remedies are not sufficient to root out forwarding flaws, which can exist even if our code does not contain any parser bugs. Fortunately, there *are* remedies to tackle forwarding flaws, as discussed above, which already appear in the literature and in practice:

- Using more structured forms of data than strings
- Using types, not only to distinguish different languages and formats that are manipulated (e.g., distinguishing HTML from SQL), but also to distinguish different trust assumptions about the data (e.g., distinguishing untrusted user input from sanitized values or constants)

The (anti-)patterns we discussed all center around the familiar LangSec themes of parsing and the expressive power of input languages; the remedies try to reduce expressive power, reduce the potential for confusion and mistakes in (un)parsing, or avoid (un)parsing altogether.

References

[1] I. Arce, K. Clark-Fisher, N. Daswani, J. DelGrosso, D. Dhillon, C. Kern, T. Kohno, C. Landwehr, G. McGraw, B. Schoenfield, M. Seltzer, D. Spinellis, I. Tarandach, and J. West, "Avoiding the Top 10 Software Security Design Flaws," Technical Report, IEEE Computer Society Center for Secure Design (CSD), 2014.

[2] J. Bangert and N. Zeldovich, "Nail: A Practical Tool for Parsing and Generating Data Formats," *;login:*, vol. 40, no. 1 (USENIX, 2015), pp. 24–30: https://www.usenix.org/system/files/login/articles/login_feb15_06_bangert.pdf.

[3] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, "Exploit Programming: From Buffer Overflows to Weird Machines and Theory of Computation," *;login:*, vol. 36, no. 6 (USENIX, 2011), pp. 13–21: https://www.usenix.org/system/files/login/articles/105516-Bratus.pdf.

[4] D. E. Denning and P. J. Denning, "Certification of Programs for Secure Information Flow," *Communications of the ACM*, vol. 20, no. 7, 1977, pp. 504–513: https://www.cs.utexas.edu/~shmat/courses/cs380s/denning.pdf.

[5] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. B. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu, "Collaborative Verification of Information Flow for a High-Assurance App Store," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS '14)*, pp. 1092–1104: https://homes.cs.washington.edu/~mernst/pubs/infoflow-ccs2014.pdf.

[6] D. Kaminsky, M .L. Patterson, and L. Sassaman, "PKI Layer Cake: New Collision Attacks against the Global X.509 Infrastructure," in *Financial Cryptography and Data Security*, vol. 6054 of LNCS (Springer, 2010), pp. 289–303: https://www.esat.kuleuven.be/cosic/publications/article-1432.pdf.

[7] C. Kern, "Securing the Tangled Web," *Communications of the ACM*, vol. 57, no. 9, 2014, pp. 38–47.

[8] F. Momot, S. Bratus, S. M. Hallberg, and M. L. Patterson, "The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them," in *Proceedings of the IEEE Conference on Cybersecurity Development (SecDev '16)*, pp. 45–52: http://langsec.org/papers/langsec-cwes-secdev2016.pdf.

[9] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich, "Safely Composable Type-Specific Languages," in *ECOOP 2014—Object-Oriented Programming*, vol. 8586 of LNCS (Springer, 2014), pp. 105–130: http://www.cs.cmu.edu/~aldrich/papers/ecoop14-tsls.pdf.

[10] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst, "Practical Pluggable Types for Java," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*, pp. 201–212: https://homes.cs.washington.edu/~mernst/pubs/pluggable-checkers-issta2008.pdf.

[11] E. Poll, "LangSec Revisited: Input Security Flaws of the Second Kind," in *Proceedings of the IEEE Symposium on Security and Privacy Workshops*, 2018, pp. 329–334: http://spw18.langsec.org/papers/Poll-Flaws-of-second-kind.pdf.

[12] A. Sabelfeld and A. C. Myers, "Language-Based Information-Flow Security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, 2003, pp. 5–19: https://www.cs.cornell.edu/andru/papers/jsac/sm-jsac03.pdf.

CSET '18

The 11th USENIX Workshop on Cyber Security Experimentation and Test

PETER A. H. PETERSON

Peter A. H. Peterson is an Assistant Professor of Computer Science at the University of Minnesota, Duluth, where he teaches and researches operating systems and security, with a focus on R&D to make security education more effective and accessible. He received his PhD from the University of California, Los Angeles, for work on "Adaptive Compression"—systems that make compression decisions dynamically to improve efficiency. He served as the Co-Chair of CSET '18. pahp@d.umn.edu

The 11th USENIX Workshop on Cyber Security Experimentation and Test (CSET '18) was held in Baltimore, Maryland, on Monday, August 13th, 2018—one of the co-located workshops preceding the 27th USENIX Security Symposium. The CSET Call for Papers "invites submissions on cyber security evaluation, experimentation, measurement, metrics, data, simulations, and testbeds." In other words, CSET emphasizes tools and methods in the service of computer security testing and research, making it somewhat unique. This year, our program consisted of 10 papers in four themed sessions and a panel. As usual, discussion was friendly and lively.

Data and Guidance was our first session, chaired by Christian Collberg. Josiah Dykstra presented the methodology and design behind the Cyber Operations Stress Survey (COSS), an instrument for measuring the effects of tactical cyber operations on the stress of operators. Stress, in the context of the COSS, is represented in terms of an operator's fatigue (measured using the Samn-Perelli Fatigue Scale, or SPFS), frustration, and cognitive workload (using the NASA Task Load Index, or TLX). Other contextual factors about an operation, such as team synergy and duration of the operation, are also captured. The COSS was used in four studies of tactical cyber operations with consistent results, demonstrating that the method was internally and externally reliable. The results of the project were presented in terms of lessons learned through the development and use of the COSS tool, including how low initial enthusiasm was mitigated by presenting initial results to operators to show their value, and how results led to policy changes. The paper also described how some of the contextual factors were chosen through pilot testing; one surprising result was that caffeine had no relationship to stress!

Next, Tyler Moore presented "Cybersecurity Research Datasets: Taxonomy and Empirical Analysis," a work that analyzed 965 recent papers for information about how the authors used, created, or shared data sets, and how those choices correlated to other factors, such as whether new data sets were shared and the citation count of the papers. They also created a taxonomy based on their observations. While the community values sharing data sets as a service, researchers may be reluctant to do so for privacy, competitive advantage, or other reasons. The authors' results suggest that there is a self-interested incentive to publish data sets—citation counts. The papers in their pool that created and released data sets received 42% more citations than papers that did not use data sets or used only public data, and 53% more citations per year than papers that created data sets but did not release them. Thus, in addition to altruism, authors may want to consider whether they would benefit more from citations than they might suffer due to competition.

After this, Samuel Marchal from Aalto University in Finland discussed "On Designing and Evaluating Phishing Webpage Detection Techniques for the Real World," which explores the state of phishing detection research, raising concerns about the effectiveness of detection strategies in terms of detection performance, temporal resilience, deployability, and usability. Too much focus, they said, is placed on numerical accuracy in an artificial environment,

without enough consideration of realism, practicality, change in accuracy over time, or comparability between experiments. For non-phish ground-truth examples, they suggest incorporating low- and high-popularity sites and using the full URLs that a user would see. For phish ground-truth examples, they recommend manual inspection to ensure that the data consists only of unique, legitimate phishes. For both, they recommend using a diverse set of current sites (not old data), languages, and types of sites. Older data should be used for training, while newer should be used for testing, and test input should have a realistic ratio of phishing to non-phishing samples. They also recommend longitudinal studies with updated data to evaluate accuracy over time, and that systems should be tested against adversarial machine-learning techniques. Their hope is that, by applying these guidelines to the evaluation of future phishing detection techniques, researchers might obtain meaningful and comparable performance results as well as fostering technology transfer.

The New Approaches session, chaired by Sven Dietrich, focused on novel approaches to experimentation. "DEW: Distributed Experiment Workflows" was presented by Genevieve Bartlett. DEW is a high-level approach to experiment representation that separates the infrastructure of a testbed experiment—the testbed hardware, software, and experimental network—from the observable behavior of the experiment in operation. The goal of DEW is to allow researchers to describe what an experiment should do, allowing DEW and the underlying testbed infrastructure to instantiate that behavior in the manner appropriate for that testbed. Meant to be human-readable, a DEW "program" consists of a scenario describing the general behavior of the experiment in terms of its actions, bindings implementing those actions (e.g., scripts), and constraints defining the required properties of the experiment's components (enforced by DEW), such as network links and computational nodes. The authors intend to produce translators that generate DEW based on an experiment setup, and generators that create an experiment setup from a DEW definition. The DEW paper includes a number of potential benefits, code samples, and a discussion of their NLP-enhanced GUI. The authors welcome communication and feedback from interested parties.

Xiyue Deng presented "Malware Analysis through High-Level Behavior," a paper that describes the Fantasm framework and some results from its use. Fantasm identifies malware type by recognizing patterns in a given malware's network behavior. After all, malware that is supposed to perform scans, exfiltrate keystrokes, or send spam needs to perform those activities, even if the binary is obfuscated. While malware can often detect and thwart monitoring performed through debuggers or virtualization, Fantasm avoids detection by running the malware on real bare-metal systems on DeterLab. Fantasm controls the malware execution on one host and monitors the network using a separate remote host, labeling malware samples based on their observed behavior. As Fantasm monitors network behavior, it can automatically decide whether to impersonate the receiver, forward the message to the original endpoint, or drop the traffic. Their paper details a number of challenges in addition to future work.

Next, Pravein Govindan Kannan told us about "BNV: Enabling Scalable Network Experimentation through Bare-metal Network Virtualization." BNV is a network hypervisor (based on OpenVirtex) that works in conjunction with specially configured switches to allow high-fidelity testing of a variety of large-scale network topologies while using a fraction of the hardware. BNV is a response to the desire to evaluate complex datacenter topologies with high accuracy, without having to physically construct those topologies. Existing approaches for topology testing are often either virtualized (e.g., Mininet or NS) or not flexible enough for their purposes (e.g., CloudLab, DeterLab); for example, BNV can change topologies within a few seconds. To provide for a large number of high-fidelity virtual switches, BNV "slices" physical switches into virtual switches and provides connectivity between these switches using physical loopback links. In this way, BNV can emulate 130 switches and 300 links with just five switches. BNV was evaluated by testing various topologies in both virtual and physical forms under a workload and comparing their performance, with good results. Currently, BNV is in use at the National Cybersecurity Lab (NCL) in Singapore.

After lunch, we held a round-table discussion on "Opportunities and Challenges in Experimentation and Test." In a sense, CSET is a workshop for research that produces tools, data, or guidance that enhances security research. While important, research of this type sometimes doesn't fit neatly into calls for papers or funding proposals. We thought it would be helpful to have experts in the field talk about their experiences and provide advice about doing work in this area. Our panelists—Terry Benzel (USC/ISI), Eric Eide (Utah), Jeremy Epstein (NSF), Simson Garfinkel (US Census Bureau), and Laura S. Tinnel (SRI)—and several members of the audience had a lively discussion on that subject. A full summary of that discussion is not possible here, but a key observation supported by many was that research of this type can often be funded as part of a larger project, where the tool in question is necessary for the research. Similarly, interesting and useful CSET-style papers can often be created by taking a previously created research system and polishing it or extending it to make it useful for others. In any case, papers should not simply be whitepapers describing the system or its creation saga, but should include an evaluation of some kind; this can include new results produced by the system, user feedback, or a validation of the system.

Eric Eide chaired the session on Shiny New Testbeds. Vitaly Ford got things started by telling us about "AMIsim: Application-Layer Advanced Metering Infrastructure Simulation

Framework for Secure Communication Protocol Performance Evaluation." Advanced Metering Infrastructure, or AMI, refers to "Smart Grid" infrastructure devices, such as smart meters, that allow for two-way communication using ZigBee or other methods. Because this wireless communication includes sensitive information about power consumption, security and privacy are of utmost importance. At the same time, the Smart Grid has fairly rigid communication and computation constraints. Building your own mini Smart Grid for research is too expensive for most researchers. Several Smart Grid simulation frameworks exist, but these focus largely on modeling electricity flow and power distribution hardware, not on the communication and computation resources necessary to test security and privacy-preserving Smart Grid protocols. AMIsim fills this gap. Based on OMNet++, AMIsim's performance is timed on a modern PC and then scaled to represent the approximate time a smart meter would take to perform the same computation. In this way, AMIsim opens the door for researchers to design and evaluate security-focused Smart Grid communication protocols in a way that was not previously available. AMIsim is under continuing development.

Our next new testbed was described in the paper "Galaxy: A Network Emulation Framework for Cybersecurity." Kevin Schoonover and Eric Michalak presented this work, which describes Galaxy, a new high-fidelity, emulation-based network testbed supporting quick, flexible, and parallel experimentation. Galaxy includes built-in logging to store results, and strongly isolates experiments so that state from one experiment does not affect another. While Galaxy could be useful for many purposes, it was specifically designed for evolutionary experiments requiring many iterations in succession, something that would be very time-consuming on a reconfigurable physical testbed, like Emulab or DeterLab. Galaxy takes configuration files describing a topology, and instantiates the network using bridging and virtual nodes using *vmbetter*. The snapshot feature of *libvirt* is used to ensure that no state persists between experiments, and the Ansible automation tool is used to instantiate topologies in parallel on a set of distributed computers. With these tools, topologies can be reverted and restarted very quickly and have high fidelity to real network behavior (two properties essential for valid evolutionary algorithm use). Their paper includes a case study using Galaxy as part of the CEADS-LIN project for developing attacker enumeration strategies through evolutionary algorithms, in addition to various enhancements planned for the tool.

The final session of CSET '18, Testbed Enhancements, was chaired by David Balenson. The first paper, "Supporting Docker in Emulab-Based Network Testbeds," was presented by Eric Eide. Given the popularity of Docker, it makes sense for Emulab to support Docker containers as "first class" nodes; it means that popular Docker-based tools used in research and industry can be easily integrated into Emulab experiments. A major challenge for this project was preserving both the Docker and Emulab experiences—testbed users should have the sense that their Docker containers and Emulab experiments "just work." This includes being able to use containers without manual intervention, but with the traffic shaping capabilities, logging, and command-line access common to traditional Emulab nodes. Emulab supports these features by automatically modifying Docker containers (in real time if necessary) to support essential capabilities. This process works well; in their experiments, 52 of the 60 most popular Docker containers could be automatically modified for use with Emulab. These enhancements are available now on Emulab.

The final paper at CSET '18, "High Performance Tor Experimentation from the Magic of Dynamic ELFs," was presented by Justin Tracey. For a variety of good reasons, experimentation on the live Tor network is discouraged. Instead, the recommended approach is to use a testbed of some kind, be it simulated, emulated, or physical. Shadow is a popular discrete-event simulator used for Tor and other types of research. It is popular, in part, because it runs real application code rather than models of the application being tested, the latter of which can lead to mistakes due to user error or when the documentation of a system is inconsistent with the software artifact in actual use. Unfortunately, discrete-event simulators often have the drawback of significant overhead. Tracey et al.'s work eliminated two major bottlenecks from Shadow by doing away with a global logging lock and by creating a new loader that enables more efficient use of CPU resources, in addition to removing restrictions on library and compiler use. In an evaluation simulating Tor networks, the enhancements reduced test time by about half. These enhancements are already part of the current version of Shadow.

Special thanks to the incredible USENIX staff, our panelists, program and steering committee, and the presenters for reviewing these summaries. The 12th CSET will again be co-located with USENIX Security 2019, with papers due in Spring 2019. Please consider submitting to or attending this unique and interesting workshop.

The Atlas Cluster Trace Repository

GEORGE AMVROSIADIS, MICHAEL KUCHNIK, JUN WOO PARK, CHUCK CRANOR, GREGORY R. GANGER, ELISABETH MOORE, AND NATHAN DEBARDELEBEN

George Amvrosiadis is a Professor of Electrical and Computer Engineering at Carnegie Mellon University and a member of the Parallel Data Lab. His current research focuses on scalable storage, distributed systems, and data analytics. He co-teaches courses on cloud computing and storage systems, and holds a PhD from the University of Toronto. gamvrosi@cmu.edu

Michael Kuchnik is a third-year PhD student in the Computer Science Department at Carnegie Mellon University and a member of the Parallel Data Lab. His research interests are in the design and analysis of computer systems, specifically projects incorporating elements of high performance computing or machine learning. Before coming to CMU, he earned his BS in computer engineering from the Georgia Institute of Technology. mkuchnik@andrew.cmu.edu

Jun Woo Park is a sixth-year PhD student in the Computer Science Department at Carnegie Mellon University and a member of the Parallel Data Lab. His research interests are in cluster scheduling and cloud computing, with a focus on leveraging the history of the jobs run in the past to make better scheduling decisions. Before coming back to CMU, he worked at Korea Asset Pricing and KulCloud. junwoop@andrew.cmu.edu

Many researchers evaluating cluster management designs today rely primarily on a trace released by Google [8] due to a scarcity of other sufficiently diverse data sources. We have gathered several longer and more varied traces and have shown that overreliance on the Google trace workload is leading researchers to overfit their results [1]. We have created the Atlas cluster trace repository [7] to aid researchers in avoiding this problem. This article explains the value of using and contributing to Atlas.

As a community of researchers and practitioners, we value systems work evaluated against real workloads. However, anyone who has attempted to find data to perform such an evaluation knows that there is a scarcity of publicly available workload traces. This scarcity is often due to legal and cultural obstacles associated with releasing data. Even when data sets get publicly released, they follow noncanonical formats, omit features useful to researchers, and get published individually on websites that eventually go offline. This has led to the creation of repositories such as SNIA IOTTA [9] for I/O traces, USENIX CFDR [10] for failure data, and the Parallel Workloads Archive [5] for HPC job logs. However, few of these repositories contain recent cluster traces, and their trace formats may vary considerably. More importantly, despite current research focusing on vertical optimizations spanning multiple hardware and software layers, none of the traces cover more than one system layer.

We have shown that this scarcity of data can lead to research that overfits to existing workloads [1]. To keep future research universally relevant, it is time we come together as a community and address this issue. This requires organizations with workloads not represented in existing public data sets to come forward and researchers to accept the responsibility of evaluating their artifacts with a variety of workloads. By having both sides come together, we can combat overfitting in systems research.

We are attempting to make this process easier through Project Atlas [7], a partnership initiated by Carnegie Mellon University and the Los Alamos National Laboratory (LANL). LANL has a variety of science and data analytics clusters, and it daily collects terabytes of log data from the operating system, job scheduler, hardware sensors, and other sources. Our goal is to analyze, model, and publicly release such logs so other researchers may use them. Since traces vary across platforms, we have created a common format and will release a version of each data set in it. This lowers the effort required to work with multiple data sets. Our common format ensures all jobs have user information, scheduler events, node and task allocations, and job outcomes. To lower the cost of releasing a data set, we will help organizations evaluate, anonymize, and host data. By making traces public, organizations can ensure their workloads are represented in future research. Two Sigma, a private hedge fund with datacenters in New York and Pittsburgh, has recently joined this effort by contributing data. *Analysis of our existing workloads shows that the LANL and Two Sigma traces differ significantly from the Google cluster trace that is most often used in literature today [8], a result that emphasizes the need for data diversity we are trying to foster through Atlas.*

Chuck Cranor is a Senior Systems Scientist in the Parallel Data Lab at Carnegie Mellon University working on high performance computing storage systems. His research interests include operating systems, storage systems, networking, and computer architecture. He is also a contributor to the *BSD open source operating systems projects. He has a DSc in computer science from Washington University at St. Louis. chuck@ece.cmu.edu

Greg Ganger is the Jatras Professor of ECE at Carnegie Mellon University and Director of the Parallel Data Lab (www.pdl.cmu.edu). He has broad research interests, with current projects exploring system support for large-scale ML (Big Learning), resource management in cloud computing, and software systems for heterogeneous storage clusters, HPC storage, and NVM. His PhD in CS&E is from the University of Michigan. ganger@ece.cmu.edu

Elisabeth Moore (Lissa) is a Research Scientist in the High Performance Computing Division at Los Alamos National Laboratory and at the Ultrascale Systems Research Center. Her research focuses on machine learning within the high performance computing space, as well as methods for explainable machine learning, and computational social science. Lissa has previously held positions in LANL's Center for Nonlinear Studies and MIT Lincoln Laboratory's Human Language Technology group. lissa@lanl.gov

Platform	Nodes	Node CPUs	Node RAM	Length
LANL Mustang	1600	24	64GB	5 years
LANL Trinity	9408	32	128GB	3 months
Two Sigma A	872	24	256GB	9 months
Two Sigma B	441	24	256GB	
Google B	6732	0.50*	0.50*	29 days
Google B	3863	0.50*	0.25*	
Google B	1001	0.50*	0.75*	
Google C	795	1.00*	1.00*	
Google A	126	0.25*	0.25*	
Google B	52	0.50*	0.12*	
Google B	5	0.50*	0.03*	
Google B	5	0.50*	0.97*	
Google C	3	1.00*	0.50*	
Google B	1	0.50*	0.06*	

Table 1: Hardware characteristics of the clusters with traces in the Atlas repository. This also includes the Google trace for reference [8]; (*) signifies resources normalized to the largest node, which is how that trace is constructed.

The Atlas Trace Repository

The Atlas cluster trace repository (http://www.project-atlas.org) hosts cluster traces from a variety of organizations, representing workloads from Internet services to high performance computing. Our immediate goal with Atlas is to help create a diverse corpus of real workload traces for researchers and practitioners. Long term, we plan to collect and host multi-layer cluster traces that combine data from several layers of systems (e.g., job scheduler and file system logs) to aid in the design of future, vertically optimized systems.

To start, we have released four sets of job scheduler logs to the Atlas trace repository. The logs are from a general-purpose LANL cluster, a cutting-edge LANL supercomputer, and from two of Two Sigma's datacenters. The hardware configuration for each cluster is shown in Table 1, and the corresponding Google trace information is included for reference.

Users typically interact with the job scheduler in these clusters by submitting jobs as scripts that spawn and distribute multiple processes or tasks across cluster nodes to perform computations. In the LANL HPC clusters, resources are allocated at the granularity of physical nodes, so tasks from different jobs are never scheduled on the same node. This is not necessarily true in private clusters like Two Sigma.

LANL Mustang Cluster

Mustang was an HPC cluster used for capacity computing at LANL from 2011 to 2016. Capacity clusters are architected as cost-effective, general-purpose resources for a large number of users. Mustang consisted of 1600 identical compute nodes, with a total of 38,400 AMD Opteron 6176 2.3 GHz cores and 102 TB RAM, and was mainly used by scientists, engineers, and software developers at LANL. Computing resources on Mustang were allocated to users at the granularity of physical nodes.

Nathan DeBardeleben is a Senior Research Scientist at Los Alamos National Laboratory and is the Co-Executive Director for Technical Operations of the Ultrascale Systems Research Center. His research focuses on resilience and reliability of supercomputers, particularly from a hardware and systems perspective. Nathan joined LANL in 2004 after completing his PhD in computer engineering from Clemson University with a focus on parallel computing.
ndebard@lanl.gov

Mustang was in operation from October 2011 to November 2016, and our Mustang data set covers the entire 61 months of the machine's lifetime. This makes the Mustang data set *the longest publicly available cluster trace to date*. The data set consists of 2.1 million multi-node jobs submitted by 565 users. Collected data include: timestamps for job stages from submission to termination, job properties such as size and owner, the job's exit status, and a time budget field per job that, if exceeded, causes the job to be killed.

LANL Trinity Supercomputer

Trinity is currently (in 2018) the largest supercomputer at LANL and is used for capability computing. Capability clusters are large-scale, high-demand resources that include novel hardware technologies that aid in achieving crucial computing milestones such as higher-resolution climate and astrophysics models. Trinity's hardware was deployed in two pre-production phases before being put into full production. Our trace was collected before the second phase completed. At the time of data collection, Trinity consisted of 9408 identical compute nodes with a total of 301,056 Intel Xeon E5-2698v3 2.3 GHz cores and 1.2 PB RAM, making this the *largest cluster with a publicly available trace by number of CPU cores*.

Our Trinity data set covers three months, from February to April 2017. During that time, Trinity was in beta testing and operating in OpenScience mode and thus was available to a wider number of users than it is expected to have after it receives its final security classification. OpenScience workloads are representative of a capability supercomputer's workload, as they occur roughly every 18 months when a new machine is introduced or before an older machine is decommissioned. We refer to Trinity's OpenScience workload trace as *OpenTrinity*. This data set consists of 25,237 multi-node jobs issued by 88 users. The information available in the trace is a superset of those available in the Mustang trace; additional scheduler information such as hosts allocated and QoS is also exposed.

Two Sigma Clusters

Our Two Sigma traces originated from two of their datacenters. The workload consists of data analytics jobs processing financial data. A fraction of these jobs are handled by an Apache Spark installation, while the rest are serviced by home-grown data analytics frameworks. The data set spans nine months of the two datacenters' operation starting in January 2016, covering a total of 1313 identical compute nodes with 31,512 CPU cores and 328 TB RAM. The logs contain 3.2 million jobs and 78.5 million tasks, collected by an internally developed job scheduler running on top of Mesos.

Unlike the LANL data sets, job runtime is not budgeted strictly in these clusters; users of the hedge fund clusters do not have to specify a time limit when submitting a job. Users can also allocate individual cores, as opposed to entire physical nodes allocated at LANL. Collected data include the same information as the LANL Mustang and Trinity traces, excluding the time budget field.

Overfitting to Existing Traces in Literature

Six years ago, Google released an invaluable set of scheduler logs, which currently have been used in more than 450 publications. Using traces we made available through Atlas, we found that the scarcity of other data sources is leading researchers to overfit their work to Google's data-set characteristics [1]. For example, both the Google trace and the Two Sigma cluster workloads in Atlas consist of data analytics jobs, but the characteristics of the Two Sigma workload display more similarity to LANL's HPC cluster workloads than to the Google workload. A summary of the results of our analysis is shown in Table 2 (the full analysis is in our recent USENIX ATC paper [1]). This observation suggests that additional traces should be considered when evaluating the generality of new research. An excerpt of our analysis that

Section	Characteristic	Google	Two Sigma	Mustang	OpenTrinity
Job Characteristics	Majority of jobs are small	✔	✘	✘	✘
	Majority of jobs are short	✔	✘	✘	✘
Workload Heterogeneity	Diurnal patterns in job submissions	✘	✔	✔	✔
	High job submission rate	✔	✔	✘	✘
Resource Utilization	Resource over-commitment	✔	✘	✘	✘
	Sub-second job interarrival periods	✔	✔	✔	✔
	User request variability	✘	✔	✔	✔
Failure Analysis	High fraction of unsuccessful job outcomes	✔	✔	✘	✔
	Jobs with unsuccessful outcomes consume significant fraction of resources	✔	✔	✘	✘
	Longer/larger jobs often terminate unsuccessfully	✔	✘	✘	✘

Table 2: Summary of the characteristics of each trace, derived from our analysis [1]. Note that the Google workload appears to be an outlier.

focuses on job characteristics, workload heterogeneity, and trace length is presented below. We also further identify work in the literature that has overfitted to characteristics of the Google trace.

Google Cluster

In 2012 Google released a 29-day trace of long-running and batch service jobs that ran in one of their compute clusters in May 2011 [8]. The trace consists of 672,074 jobs with 48 million tasks running on 12,583 heterogeneous nodes. Google has not released the exact hardware specifications of the nodes. Instead, as shown in Table 1, nodes are presented through anonymized platform names representing machines with different combinations of microarchitectures and chipsets. Note that the number of CPU cores and amount of RAM for each node in the trace has been normalized to the most powerful node in the cluster. Google's most popular server node type in 2011 is believed to be a dual-socket quad-core system with AMD Barcelona CPUs. If this is accurate, we estimate the total number of cores in the Google cluster to be 106,544. Google allows jobs to allocate fractions of a CPU core, so more than one job can be running on a node.

Analysis of Job Characteristics

Many instances of prior work in the literature rely on the assumption of heavy-tailed distributions to describe the size and duration of individual jobs. In the LANL and Two Sigma workloads these tails appear significantly lighter.

On average, jobs in the Two Sigma and LANL traces request 3–406 times more CPU cores than Google trace jobs.

Figure 1 shows the cumulative distribution functions (CDFs) of job requests for CPU cores across all traces, with the x-axis in logarithmic scale. We find that the 90% of smallest jobs in the Google trace request 16 CPU cores or fewer. The same fraction of Two Sigma and LANL jobs request 108 cores and 1–16K cores, respectively. Very large jobs are also more common outside Google. This is unsurprising for the LANL HPC clusters, where allocating thousands of CPU cores to a single job is common since the clusters' primary use is to run massively parallel scientific applications. However, it is interesting to note that while the Two Sigma clusters contain fewer cores than the other clusters we examined (one-third of those in the Google cluster), its median job is more than an order of magnitude larger than jobs in the Google trace. An analysis of allocated memory yields similar trends.

The median job in the Google trace is 4–5 times shorter than in the LANL or Two Sigma traces.

Figure 2 shows the CDFs of job durations for all traces. We find that in the Google trace, 80% of jobs last less than 12 minutes *each*. In the LANL and Two Sigma traces, jobs are at least an order of magnitude longer. In Two Sigma, the same fraction of jobs lasts up to two hours, and in LANL they last up to three hours for Mustang and six hours for OpenTrinity. Surprisingly, the tail end of the distribution is slightly shorter for the LANL clusters than for the Google and Two Sigma clusters. The longest job is hours in the Atlas traces and is days in the Google traces. For LANL, this is due to hard job time limits. For Google, the distribution's long tail is likely attributed to long-running services.

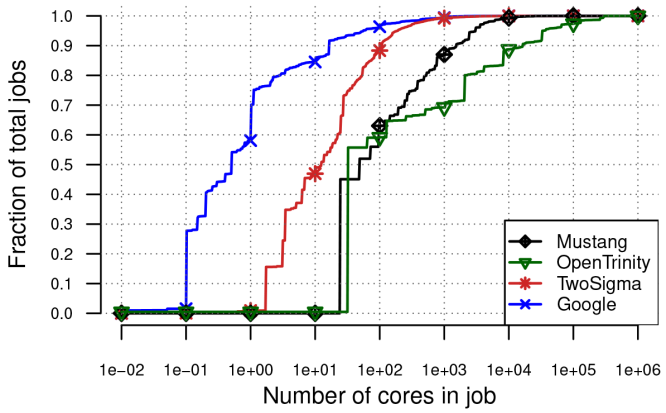


Figure 1: CDF of job sizes based on allocated CPU cores. Jobs at Two Sigma and LANL use 3–406 times more CPU cores than Google trace jobs, which challenges existing work that relies on the assumption that small jobs are prevalent in a typical cluster.

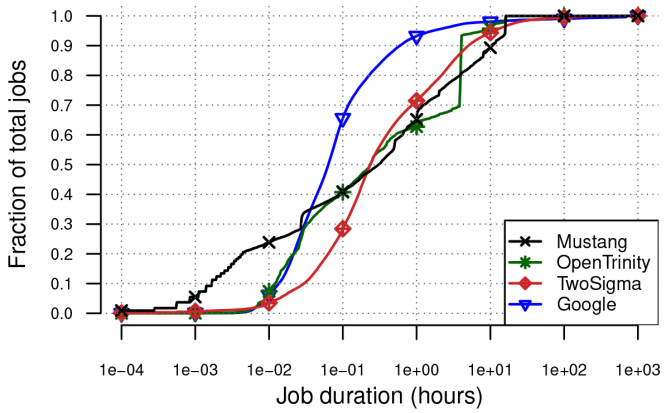


Figure 2: CDF of the durations of individual jobs. The median job in the Google trace is 4–5 times shorter than in the LANL or Two Sigma traces, urging us to reevaluate the feasibility of scheduling approaches that have been designed with the Google trace workload in mind.

Implications. These observations impact the applicability of job scheduling approaches whose efficiency relies on the assumption that the vast majority of jobs' durations are on the order of minutes, and job sizes are insignificant compared to the size of the cluster. For example, Ananthanarayanan et al. [2] propose to mitigate the effect of stragglers by duplicating tasks of smaller jobs. This is an effective approach for Internet service workloads because the vast majority of jobs can benefit from it without significantly increasing the overall cluster utilization. For the Google trace, 90% of jobs request fewer than 0.01% of the cluster each, so duplicating them only slightly increases cluster utilization. On the other hand, 25–55% of jobs in the LANL and Two Sigma traces *each* request *more than* 0.1% of the cluster's cores, suggesting that replication should be used judiciously. Also note that LANL tasks are tightly coupled, so entire jobs would have to be duplicated. Another example is the work by Delgado et al. [3], which improves the efficiency of distributed schedulers for short jobs by dedicating them to a fraction of the cluster. For the Two Sigma and LANL traces, we have shown that jobs are longer than for the Google trace (Figure 2), so larger partitions will likely be necessary to achieve similar efficiency. At the same time, jobs running in the Two Sigma and LANL clusters are also larger (Figure 1), so service times for long jobs are expected to increase unless the partition is shrunk.

Analysis of Workload Heterogeneity

Another common assumption about cloud workloads is that they run on heterogeneous compute nodes and have job interarrival times on the order of seconds. However, the LANL and Two Sigma clusters consist of homogeneous hardware (see Table 1) and have a scheduling rate that varies significantly across clusters.

Scheduling request rates differ by up to three orders of magnitude across clusters. Sub-second scheduling decisions seem necessary in order to keep up with the workload.

In Figure 3 we show the number of job scheduling requests for every hour of the day. Similar to prior work, diurnal patterns are evident in every trace, and user activity is concentrated in the daytime (7 a.m. to 7 p.m.). An exception to this is the Google trace, which is most active from midnight to 4 a.m., presumably due to batch jobs leveraging available resources. Figure 3 also shows that the rate of scheduling requests can differ significantly across clusters. For the Google and Two Sigma traces, hundreds to thousands of jobs are submitted every hour. On the other hand, LANL schedulers never receive more than tens of requests on any given hour. This could be related to the workload or to the number of users in the system, as the private clusters serve 2–9 times as many user IDs as the LANL clusters.

Implications. As cluster sizes increase, so does the rate of scheduling requests, urging us to reexamine prior work. Quincy [6] represents scheduling as a Min-Cost Max-Flow (MCMF) optimization problem over a task-node graph and continuously refines task placement. However, the complexity of this approach becomes a drawback for large-scale clusters. Gog et al. [4] find that Quincy requires 66 seconds (on average) to converge to a placement decision in a 10,000-node cluster. The Google and LANL clusters we study already operate on that scale. Note that when discussing scheduling so far we refer to *jobs*, since HPC jobs have a gang scheduling requirement. Placement algorithms such as Quincy, however, focus on *task* placement. An improvement to Quincy is Firmament [4], a centralized scheduler employing a generalized approach based on a combination of MCMF optimization techniques to achieve sub-second task placement latency on average. Sub-second latency is paramount, since the rate of task placement requests in the Google and Two

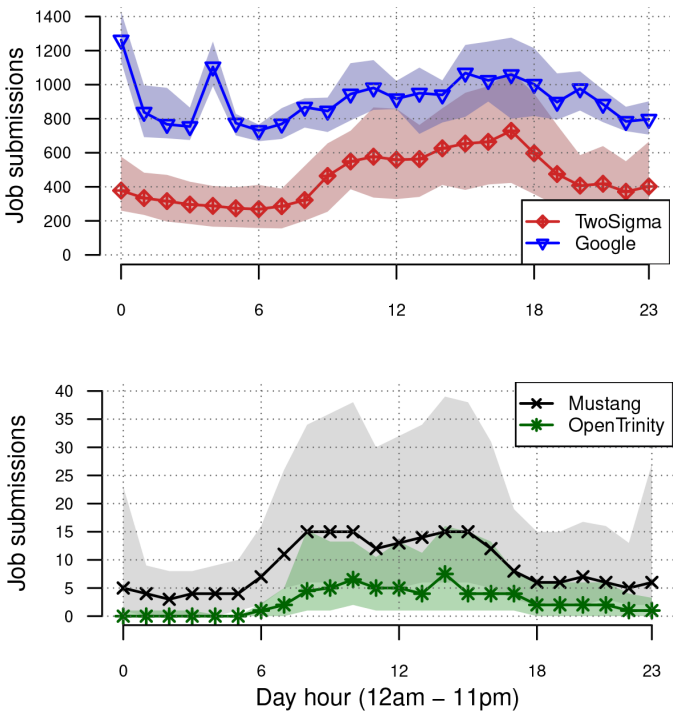


Figure 3: Hourly job submission rates for a given day. Lines represent the median, while the shaded region for each line outlines the span from the 25th (under) to the 75th percentile (over). LANL traces show lower rates of job submission, and the diurnal patterns for each trace appear at different times.

Sigma traces can be as high as 100K requests per hour, i.e., one task every 36 ms. However, Firmament's placement latency increases to several seconds as cluster utilization increases. For the Two Sigma and Google traces this can be problematic.

The Importance of Trace Length

Working with traces often forces researchers to make key assumptions as they interpret the data in order to cope with missing information. A common (unwritten) assumption is that traces represent the workload of the environment where they were collected. While the Google trace spans only 29 days, our Atlas traces are 3–60 times longer and in the case of Mustang cover the entire cluster lifetime. Thus, we decided to examine how representative individual 29-day periods are of the overall workload.

Our experiment consisted of dividing our traces in 29-day periods. For each such month we then compared the distributions of individual metrics against the overall distribution for the full trace. The metrics we considered were: job sizes, durations, and interarrival periods. Overall, we found consecutive months' distributions to vary wildly for all these metrics. More specifically, the average job interarrival of a given month can be 20–2400% the value of the overall average. Average job durations can fluctuate 10–6900% of the average job duration.

Call for Traces

In order to guarantee that researchers and practitioners design and develop systems that will be truly universally relevant, we need to make a collective effort as a community to ensure that the workloads we care about are represented with publicly available traces. This way we will be able to both gain a better understanding of trends and pain points that span industries and create software and hardware that affect a wider population. Through Project Atlas, we urge and welcome members of this community to come forward with cluster traces collected at any layer of their systems: scheduler logs, file system logs, application profiling data, operating system logs, etc. We further look forward to contributions of multi-layer traces that stitch together multiple such data sources.

To aid in the process of releasing new data sets, we will be happy to help by sharing our experiences and software for data collection, analysis, and anonymization. We also offer to host new data sets in the Atlas repository, which is accessible through www.project-atlas.org. Please feel free to direct any communication to info@project-atlas.org.

References

[1] G. Amvrosiadis, J. W. Park, G. R. Ganger, G. A. Gibson, E. Baseman, and N. DeBardeleben, "On the Diversity of Cluster Workloads and Its Impact on Research Results," in *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, pp. 533–546: https://www.usenix.org/system/files/conference/atc18/atc18-amvrosiadis.pdf.

[2] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective Straggler Mitigation: Attack of the Clones," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, pp. 185–198: https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final231.pdf.

[3] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid Datacenter Scheduling," in *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC '15)*, pp. 499–510: https://www.usenix.org/system/files/conference/atc15/atc15-paper-delgado.pdf.

[4] I. Gog, M. Schwarzkopf, A. Gleave, R. N. M. Watson, and S. Hand, "Firmament: Fast, Centralized Cluster Scheduling at Scale," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pp. 99–115: https://www.usenix.org/system/files/conference/osdi16/osdi16-gog.pdf.

[5] Hebrew University of Jerusalem, Parallel Workloads Archive: http://www.cs.huji.ac.il/labs/parallel/workload/.

[6] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair Scheduling for Distributed Computing Clusters," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pp. 261–276: http://www.sigops.org/sosp/sosp09/papers/isard-sosp09.pdf.

[7] Parallel Data Laboratory, Carnegie Mellon University, Atlas Repository: Traces: http://www.project-atlas.org/.

[8] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC '12)*, pp. 7:1–7:13: http://www.pdl.cmu.edu/PDL-FTP/CloudComputing/googletrace-socc2012.pdf.

[9] Storage Networking Industry Association, I/O traces, tools, and analysis repository: http://iotta.snia.org/.

[10] USENIX Association, The Computer Failure Data Repository (CFDR): https://www.usenix.org/cfdr.

The Modern Data Architecture

The Deconstructed Database

AMANDEEP KHURANA AND JULIEN LE DEM

Amandeep Khurana is cofounder and CEO at Okera, a company focused on solving data management challenges in modern data platforms. Previously, he was a Principal Architect at Cloudera where he supported customer initiatives and oversaw some of the industry's largest big data implementations. Prior to that, he was at AWS on the Elastic MapReduce engineering team. Amandeep is passionate about distributed systems, big data, and everything cloud. Amandeep is also the coauthor of *HBase in Action*, a book on building applications with HBase. Amandeep holds an MS in computer science from the University of California, Santa Cruz. amansk@gmail.com

Julien Le Dem is the coauthor of Apache Parquet and the PMC chair of the project. He is also a committer and PMC Member on Apache Pig, Apache Arrow, and a few others. Julien is a Principal Engineer at WeWork working on data platform, and was previously Architect at Dremio and Tech Lead for Twitter's data processing tools, where he also obtained a two-character Twitter handle (@J_). Prior to Twitter, Julien was a Principal Engineer and Tech Lead working on content platforms at Yahoo, where he received his Hadoop initiation. His French accent makes his talks particularly attractive. julien@ledem.net

Mainframes evolved into the relational database in the 1970s with the core tenet of providing users with an easier-to-use abstraction, an expressive query language, and a vertically integrated system. With the explosion of data in the early 2000s, we created the big data stack and decoupled storage from compute. Since then the community has gone on to build the modern data platform that looks like a deconstructed database. We survey the different technologies that have been built to support big data and what a modern data platform looks like, especially in the era of the cloud.

Modern data platform architectures are spurring a wave of innovation and intelligence by enabling new workloads that weren't possible before. We will review three main phases of technology evolution to highlight how the user experience of working with data has changed over time. The article concludes with a review of the current state of data architectures and how they are changing to better meet demand.

From Mainframe to Database—A Brief Review

Mainframes were among the early platforms for applications and analytics done in a programmatic way, using what we know as modern computing systems. In the world of mainframes, users had to write code to interact with data structures as stored on disk. Users had to know the details of the data storage with which they were working, including its location and storage format. These details had to be coded as a part of the application. You could still write arbitrarily complex logic to work with the data, but the paradigm was not very accessible or easy to understand by mainstream users. The technical complexity was a hurdle users had to overcome, thus limiting the adoption of this paradigm.

Fortunately, in the 1970s, the relational database was born. It was created based on a few core tenets:

- Simplify the data abstraction for the end user and make it more intuitive.
- Provide a rich language to facilitate the expression of computational logic.
- Hide the complexity of the underlying systems from end users.

These goals are clearly articulated in the first paragraph of Codd's 1970 paper on relational models [1], one of the first papers on relational databases. You don't have to read much past the first three sentences of his paper:

> *Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.*

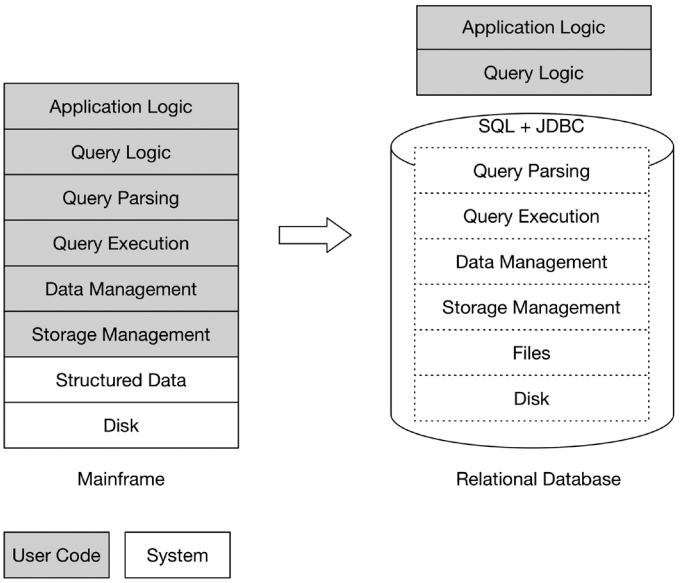


Figure 1: The evolution of the technology stack from the mainframe to the database

This means:

- Users of database systems should not have to worry about the underlying layouts, how and where data is stored, formats, etc.
- If changes need to be made to underlying files and structures, the applications should not be affected.
- Anything that provides more and better information about the underlying data structure doesn't necessarily reduce the technical complexity.

One could argue that data analytics, as we know it today, was made possible by the relational database. For the first time, companies could leverage their data and extract value from it. The relational database employed SQL (created in the early 1970s at IBM) as the language to express computation, and 1986 saw the first SQL standard, which has been updated several times since, eventually becoming a global standard. SQL has some excellent properties, including a strong separation of the query logic from the execution details. The table abstraction and how it is stored or indexed is opaque to the user, who can concentrate on the data logic rather than the storage implementation. An optimizer is charged with finding the best way to produce the requested data and exploit the properties of the underlying storage (column-oriented, indexed, sorted, partitioned). Additionally, ACID guarantees, integrity constraints, and transactions help to ensure certain properties of the data.

In addition to a standard language to express data-processing logic, Sun Microsystems released JDBC as a standard API, which further abstracted the underlying SQL implementation from the user (see Figure 1). An entire ecosystem of technologies and applications was created around the relational database.

At the very core, it was ease of use, the accessibility and the simplicity of the database, that led to its broad adoption. You no longer needed to be an engineer to work with data.

The Birth of the Big Data Stack

In late 1990s and early 2000s, the relational database struggled to keep up with the explosion of data. Technologists who worked with large amounts of data re-evaluated data platform architectures. The reasons for this included scalability limitations, the increasingly heterogeneous nature of data, and the types of workloads people wanted to run. The database's architecture constrained these capabilities. SQL, as a language, was not expressive enough, and the database wasn't flexible and scalable enough to support different workloads.

This reconsideration of data storage and processing was the genesis of the big data stack and, later on, the concept of the data-lake. The Apache Hadoop project was at the core of this. Hadoop started in 2006 as a spin-off from Apache Nutch, a web crawler that stemmed from Apache Lucene, the famous open source search engine. The inspiration for this project came from two Google papers describing the Google File System [2] and a distributed processing framework called MapReduce [3]. These two components combined the extreme flexibility and scalability necessary to develop distributed batch applications in a simple way.

The Hadoop Distributed File System (HDFS)

HDFS provides a file system abstraction over a cluster of mainstream servers. It also provides metadata on data placement, which is exploited by MapReduce to process data where it is stored. Back when network I/O was much more constrained than disk I/O, this innovation was significant. HDFS files are free form; there are no constraints on the format or any kind of schema. We started to call this concept *schema on read* (as opposed to *schema on write* in the world of databases).

MapReduce

MapReduce provides a simple framework to build distributed batch applications on top of HDFS. Usually a job is defined by scanning a data set in parallel, applying a Map function to the content, and emitting key-value pairs. All values with the same key are sent to the same machine, independent of where they were produced, in a step called "the shuffle." The key and its corresponding list of values are then passed to the Reduce function. This simple framework allows us to build powerful distributed algorithms. One example is the famous PageRank algorithm, originally used by Google to rank websites based on how many incoming links they get.

MapReduce is a very flexible paradigm. MapReduce simply edits key-value pairs, and it is also composable, allowing its users to

realize complex algorithms by orchestrating multiple MapReduce steps. For example, PageRank converges to a result after a number of iterations. The inherent limitations of MapReduce, however, come from the same attributes that make it strong. The flexibility in file formats and the code used to process them offer no support for optimizing data access. In that respect, the MapReduce paradigm returns us to the world of mainframes at a much larger scale. The MapReduce programmer must in fact know quite a bit about storage details to write a successful program.

"MapReduce, a Major Step Backwards"

The database community eventually became annoyed by this new wave of open source people re-inventing the wheel. Turing Award winner Michael Stonebraker, of PostgreSQL fame and recent co-founder of the distributed analytical database Vertica, famously declared in 2008 [4] that MapReduce was "a major step backwards." Compared to the nice abstractions of the relational model, this new model was too low level and complex.

Evolution of the Big Data Stack

Ten years later, the entire Hadoop ecosystem is much larger than the two components it originally included. People argued about where the boundary of that ecosystem really stopped. In the cloud, you can even use a significant portion of the ecosystem without Hadoop itself. New functional categories beyond storage and compute have emerged: execution engines, streaming ingest, resource management, and, of course, a long list of SQL-on-Hadoop distributed query engines: Impala, Hive, SparkSQL, Drill, Phoenix, Presto, Tajo, Kylin, etc. The ecosystem can be broken down into the following categories:

Storage Systems

HDFS, S3, and Google Cloud Storage are the distributed file system/object stores where data of all kinds can be stored. Apache Parquet has become a standard file format for immutable columnar storage at rest.

Apache Kudu and Apache HBase provide mutable storage layers with similar abstractions, enabling projection and predicate pushdown to minimize I/O by retrieving only the data needed from disk. These projects require explicit schema, and getting that right is critical to efficient access.

Streaming Systems

Kafka is the most popular stream persistence system in the data platform world today. It's open source and is widely used for streaming data at scale. Kinesis, an AWS service, is the most popular hosted and managed streaming framework but is not available outside the AWS environment. GCP provides a similar service called PubSub. Another noteworthy platform is Pulsar. Pulsar has interesting features for multi-tenancy and performance of concurrent consumers on the same stream. The project is more of a challenger that has yet to reach wide adoption.

Query Engines

Since MapReduce, many other query engines have developed. Originally, they were often layered on top of MapReduce, which introduced a lot of latency; MapReduce is designed for web-scale indexing and is optimized for fault tolerance, not quick response. Its goal is to optimize for running very large, long-running jobs, during which a physical failure of at least one component is likely.

For example, Hive (an SQL implementation) and Pig (a functional DSL with similar capabilities) both originally compiled to a sequence of MapReduce jobs.

As more people wanted interactive capability for data analytics, Hadoop-compatible data-processing engines evolved and MapReduce became less relevant. Spark has become a popular alternative, with its richer set of primitives that can be combined to form Data Availability Groups (DAGs) of operators. It includes an in-memory cache feature that allows fast iterations on a data set during a session of work. Tez is a lower level DAG of operator APIs aimed at optimizing similar types of work. SparkSQL is a SQL implementation on top of Spark. Hive and Pig both can now run on either MapReduce, Spark, or Tez.

There are several SQL engines that provide their own runtime, all with the goal of minimizing query latency. Apache Drill and Apache Presto generate optimized Java bytecode for their operators, while Apache Impala uses LLVM to generate native code. Apache Phoenix supports a SQL interface to Apache HBase and takes advantage of HBase's capabilities to reduce I/O by running code in the database.

Tools such as Python, with the help of libraries like NumPy and pandas and R, are also very popular for data processing and can cater to a large variety of use cases that don't need the scale that MapReduce or Spark supports.

In addition to these, we're seeing a variety of machine-learning frameworks being created, such as Tensorflow, DL4J, Spark MLLib, and H2O. Each of these is specialized for certain workloads, so we are going to see more of these emerge over the next few years.

Query Optimizer

There is a query parser and optimizer framework in the Calcite project, one of the lesser known but most important projects in the ecosystem. Calcite powers the query execution in projects such as Hive, Drill, Phoenix, and Kylin. Calcite is also used in several streaming SQL engines such as Apex, Flink, SamzaSQL, and StormSQL. It can be customized in multiple ways. Notably, one would provide an execution engine, schema, and connectors implementations as well as optimization rules either relevant to

the execution engine, the storage layer, or both. Spark, Impala, and Presto have their own query optimizers and don't use an external engine.

Serialization

Apache Arrow is a standard in-memory columnar representation that combines efficient in-memory query evaluation, allowing for zero-overhead serialization, with standard simplifying integration, removing unnecessary and costly conversion layers. It also allows fast in-memory processing by enabling vectorized execution.

Security

Access control policies can be put in policy stores like Apache Sentry and Apache Ranger. In addition, there are proprietary tools such as BlueTalon that enable access control on SQL engines.

Cataloging and Governance

There are a few offerings in this realm as well, but none that truly solve the problems of today. The Hive Metastore is the dominant schema registry. Apache Atlas is an open source framework that's focused on governance. Proprietary tools such as AWS Glue, Cloudera Navigator, Alation, Waterline, and Collibra are solving different aspects of the problem.

Towards a Modern Data Platform—The Deconstructed Database

In parallel to the evolution of the data-lake concept and the big-data stack, the world of cloud computing continues to redefine technology architectures. Cloud computing normalizes variants of infrastructure, platform, and applications as a service. We are now seeing the emergence of Data-as-a-Service (DaaS). All these trends constitute a significant paradigm shift from the world of datacenters, in which enterprises had to either build their own datacenters or buy capacity from a provider. The kind of data platform that people want to build today, especially in the cloud, looks very different from what we have seen so far. At the same time, the modern data platform borrows many of the core tenets we valued in previous generations. The core tenets of the cloud include:

1. Allowing **choice** between multiple analytics frameworks for data consumers so they can pick the best tool for the workload.
2. **Flexibility** in the underlying data source systems but a consistent means to enable and govern new workloads and users.
3. A **self-service experience** for the end user: no waiting on IT and engineering teams to catch up and deliver on all the asks from all the constituents they have to serve.

Agility and self-service require components to be loosely coupled, easily available as a service or open source software, and usable in different contexts. Systems that are loosely coupled need to have common, standard abstractions in order to work together. Many of these are missing today, which makes building a true modern data platform with the core tenets articulated above challenging.

Given where the ecosystem is headed, new developments are enabling the capabilities that people want. Key areas that are experiencing significant innovation include:

1. **Improved metadata repository and better table abstractions**. There are many promising projects maturing in the open source ecosystem. For example, the Iceberg project from Netflix defines table abstractions to provide snapshot isolation and serialization semantics (at a high level, not row by row) to update data in a distributed file system. Iceberg abstracts away formats and file layouts while enabling predicate and projection push downs. Marquez is also a project that defines a metadata repository to take advantage of this work.
2. **Access control and governance across different engines and storage systems**. Current methodologies are fragmented and not fully featured. In the wake of GDPR and other privacy acts, security and privacy are important aspects of the data platform. Labeling private data appropriately to track its use across the entire platform, and enabling only approved use cases, has become a key requirement. The current ecosystem does not deliver on this, and there are new developments that will take place to fill this gap.
3. **Unifying push-down logic**. A great help toward more consistent performance of query engines on top of Parquet and other columnar storage would be unifying push-down logic. Current implementations are very fragmented and duplicate effort. The same concepts apply to streaming.
4. **Arrow project adoption to enable better interoperability between components**. This would enable simpler and more general interoperability between systems but, more importantly, would do so without sacrificing performance as lowest common denominator type integrations often do.
5. **A common access layer**. A unified access layer that allows push-downs (projection, predicate, aggregation) and retrieves the data in a standard format efficiently will advance modern data architectures. We need this capability whether or not data storage is mutable (HBase, Cassandra, Kudu), batch-oriented (HDFS, S3), or streaming-oriented (Kafka, Kinesis). This unified access layer will improve interoperability and performance, reduce duplication, and support more consistent behavior across engines. A lot of other data management problems can be solved at this layer. This is also in line with Codd's core tenet of databases: users of large data banks should not have to deal with the internal semantics of data storage.

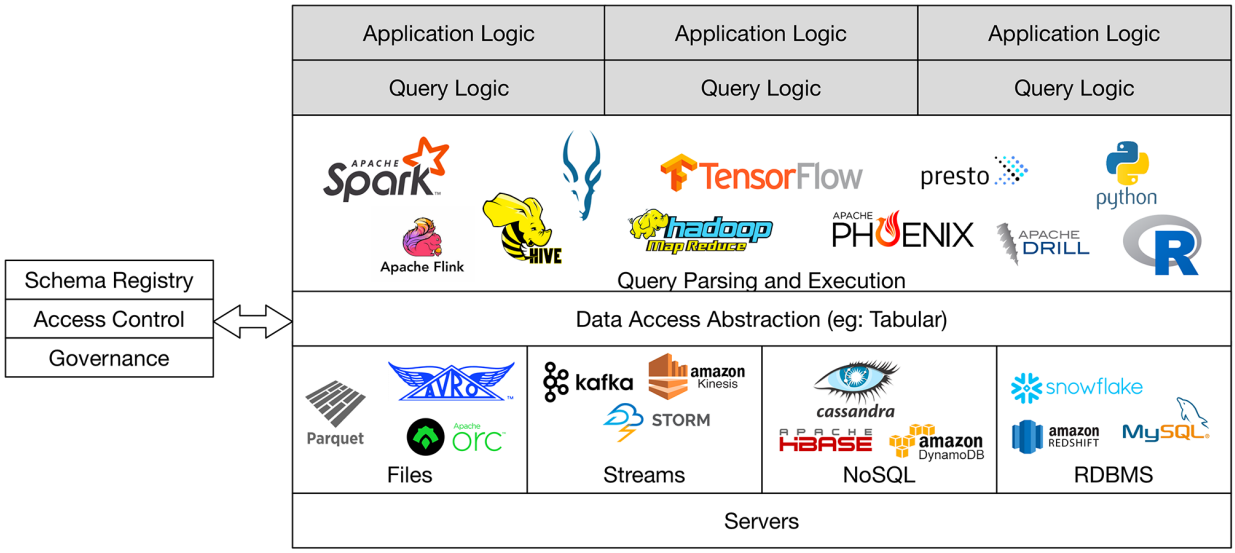


Figure 2: The modern data platform stack

Conclusion

Bringing it all together, a modern data platform will look similar to the stack shown in Figure 2. It will have a subset of these components integrated as independent, specialized services. The figure shows a few examples of the technologies at various levels of the stack and is not an exhaustive list.

A typical deployment may not always consist of all the components shown. Platform owners will be able to pick and choose the most appropriate ones and create their own stack, giving end users the flexibility and scale they need to run new workloads in the enterprise. This modular approach is a powerful paradigm that will further enable new capabilities for enterprises. This will drive more innovation and disruption in the industry, making businesses data-driven by shortening time to market of applications that take advantage of the large volumes of data that are defining the modern enterprise.

References

[1] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, vol. 13, no. 6 (June 1970), pp. 377–387: https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*: https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1 (January 2008), pp. 107–113: https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf.

[4] D. J. DeWitt and M. Stonebraker, "MapReduce: A Major Step Backwards," The Database Column, 2008: http://db.cs.berkeley.edu/cs286/papers/backwards-vertica2008.pdf.

And Now for Something Completely Different

PETER NORTON

Peter works on automating cloud environments. He loves using Python to solve problems. He has contributed to books on Linux and Python, helped with the New York Linux Users Group, and helped to organize past DevOpsDays NYC events. In addition to Python, Peter is slowly improving his knowledge of Rust, Clojure, and maybe other fun things. Even though he is a native New Yorker, he is currently living in and working from home in the northeast of Brazil. pcnorton@rbox.co.

I use Python in my day-to-day work, and I aspire to being able to write about things that I would want to know more about if I were reading this column instead of writing it. I use Python with Saltstack for writing internal APIs, for templating, for writing one-off tools, and for trying out ideas. It's my first choice as a go-to tool for almost anything at this point. I've come to realize that it's the lens that I view my computer and my job through.

Between writing my first column and this one, there was the announcement that Python is changing in a fundamental way. So I feel the need to take this opportunity to reflect on the extraordinary nature of this change: on July 12, 2018, Guido van Rossum elected to step down as the BDFL of the Python language [1].

A lot has been written about the circumstances, and I am not able to add useful commentary or knowledge about Guido's decision to retire from his title and his position in the community. I just want to add my own voice to those who have thanked him for shepherding the language for as long as he has done.

This is also a great chance to give props to all of Python's maintainers who will be guiding the language to its next phase of governance and to discuss what that may mean for those of us who mainly use the language. So while this article will be non-technical, I hope it will at least be informative, interesting, and useful.

Conway's Law

Conway's Law [2] is often invoked when asking how some piece of software developed into its current state. The version at Wikipedia attributed to Melvin Conway says, "organizations which design systems…are constrained to produce designs which are copies of the communication structures of these organizations." If you've ever wondered why a system is written in a byzantine-seeming way including paths in and out of various modules and dependencies that don't appear to make sense to you, it is often because of Conway's Law: the software needed to be worked on is under the constraints imposed by the organization writing it, not just based on the needs of the software.

However, the law also describes systems organized in a clear and sensible, easy-to-follow manner, which often doesn't get noted in describing positive aspects of software.

Python

So let's take a step back and think about Python. The core of the model for changes to Python is most typical of a new programming language: someone wrote it, and that person is in charge. It makes intuitive sense on almost every level. When Python was new, it was mostly simple to defer all decisions to Guido, since he clearly cared and was willing to shoulder the burden. As it developed, in order to accommodate the wishes of its growing community, Python developed a PEP (Python Enhancements Proposal) process for proposing changes to the language, its core modules, the C API, and to make clear what was "standard" (e.g., what other implementations needed to do in order to be considered "an implementation of Python" that could use code written for other Python implementations and not merely be considered "like Python") and what was specific to the C implementation.

The PEP process was there to provide feedback to language maintainers, and Guido was given the jocular title of the BDFL, "Benevolent Dictator for Life." This title has always been conditioned on Guido actually wanting it, and the flexibility given to him and to the language by that nuance has meant that even though Python made that acronym popular, it has become a fairly common term to give to maintainers in programming communities ever since it was coined.

Most of the languages I've seen other people enjoy using have been governed by an involved benevolent leader. Most of these have also been dynamic scripting languages: Ruby, Python, Tcl, and Perl are all somewhat similar as languages, and all follow a similar model: they each have a large audience, core developers, and a single leader whom they flourish/flourished under at some point.

Outside of "scripting languages," some other languages that have the same broad leadership model are Clojure and Scala. When I stopped to think about it, most of the hot languages of the past decade benefitted from having an undisputed leadership and support from the core group of users and maintainers.

In addition to these examples, there is evidence that the BDFL leadership model isn't a critical part of the success of a language—successful languages curated by a company or a committee include Java, C, C++, Haskell, and OCaml, among others. In addition, in a "similar to Python" vein, node.js, for one, is clearly successful, and its governance is managed quite differently. So even though there are many successful models, it's not a stretch to think that the languages that have thrived under a leader have done well solely because of that leadership.

If you use Python as a nontrivial codebase, you've probably considered how Python's organization around a minimal core with many modules matches what enables central language maintainers to do the best job they can. The fun and interesting question is how and whether it has affected the structure and the development of your software.

Going forward, the Python core maintainers and broader committer community have begun the difficult and admirable process of describing what they need in order to feel like they can make good decisions. This means that they are creating documentation on the process and also the context of the decisions that are being made. As you might expect, a new series of PEPs have been produced in order to describe the future of Python governance. Starting at PEP 8000 [3] a series of decisions will be made, and in the end PEP 13 [4] will get filled in with the decisions that are reached.

A deliberate part of the outcome of this will be documentation and data about how other software projects and companies are managed. I understand that they are seeking a common understanding so that everyone participating can make an informed decision towards a common goal of helping Python thrive. This is being acted on as an opportunity to provide future maintainers—themselves and others—with the guidelines and knowledge of how and why they made their decisions. If it's ever necessary to change the governance model again, this will probably make the process easier.

PEP 8002 [5, 6] is absolutely fascinating—the Python community is reaching out to other communities and is asking questions about their governance, which may not be documented clearly enough for outsiders to simply comprehend, and the resulting survey provides material for the Python community to understand where they—where *we*—fit in the broader community of software users. Looking at the Git log of the text of this PEP, I see more and more information being added to it weekly, and each addition is fascinating.

A notable point is that the communities in PEP 8002 are not just other languages. As of this writing, it does include Rust and Typescript, but it also includes Jupyter, Openstack, and Django, as well as Microsoft to add a significantly different and contrasting perspective.

Speculation

I'm now going to put out some very unreliable and probably baseless speculation about what will be done to the language in the future.

First, it is uncontroversial that there is an industry trend that CPU speeds have leveled off. Even though special purpose compute units like GPUs are taking over some workloads, threading that isn't bound to a single CPU is becoming more important, not less. I hope that something new could come to Python to improve its story here, even though it's unlikely considering the current and past state of the language.

In recent 3.x releases, however, the addition of *async* features and libraries emphasize how important it is to have some way of scaling that gets closer to true parallel multithreading. In the long term, could a change in the governance model prioritize multithreaded scheduling?

Another recent change in 3.6+ is type hints and their use for static type checks, even though one of the great things about Python is that the usage of types is very beginner-friendly: that is, flexible and forgiving (as they are in Ruby, Perl, and many other languages!). They are also very expert-friendly! If you know what you are doing, the thinking goes, the lack of compile-time type checking lets you get through prototyping faster.

However, in spite of how friendly Python and similar languages are, it's clear that in many cases strict compile-time checks are a huge benefit. An example of this is the development of HHVM

(HipHop Virtual Machine). In case you're not aware, PHP is also a very flexible dynamically typed language. It is the underpinning of a huge enterprise, and that enterprise created a version of PHP for its own use where they added static type annotations. This feature then made its way to mainstream PHP 7 and above.

I feel that the needs of the business fundamentally altered how they perceived the benefits and difficulties of the language they were using, to the degree that they changed fundamental aspects of that language, trading away some ease of use for what I understand to be a huge benefit. They did this by creating a slightly different language, and while communicating with the maintainer of PHP, and the benefit became a part of mainstream PHP.

If you view this progression as an extension of Conway's Law, that could tell us something about some of the potential directions that Python could go in, and also could perhaps indicate some of the benefits along with the costs. A lot of the benefit of HHVM and PHP derive from the type hints being provided to a JIT, though, and that sounds like something that is closer to **PyPy** than to standard C-Python. But as long as I'm speculating wildly: there you have it.

Changes

I am not trying to predict anything here and now except the obvious: there is potential for huge changes in Python in the long term if the community of maintainers and users come together and agree on the inherent benefits. I am not hoping that anyone try to burn down the amazing system that we have and love! My message is that it will be important to have civil conversations as the maintainers peer into their crystal balls, predict the future, and try to guide the language—but there may be some things that were considered unstoppable, immovable, or invariant that could be called into question now!

It simply seems more possible that there will be a chance to accommodate experiments that haven't been getting done because the opinions of the BDFL were known and would make some suggestions dead on arrival. For the most part, it seems unlikely that the maintainers of Python will want to change the language drastically, but looking at the possibilities with an open mind will benefit everyone greatly.

To follow past, present, and future developments, go to the PEP index at https://www.python.org/dev/peps/, where you will find:

- PEP 8002 describing the governance models of other software projects
- PEP 8010 describing the BDFL governance model
- PEP 8011 describing the council governance model
- PEP 8012 describing the community governance model
- PEP 8013 describing the external council governance model

Ongoing meta-discussion in the community is forming the PEPs above. It's also important to pay attention to the python-committers mailing list (https://mail.python.org/pipermail/python-committers/). At this time there have been discussions about how to time box the discussion so that a decision can be made, though I'm not clear on whether there is an agreement about an actual date just yet.

Decisions are being made in large and small ways constantly, and they always have been. Python sprints (https://python-sprints.github.io/) are places where developers get together and discuss Python in addition to hacking on it. Obviously, Python's past, present, and future are discussed at the sprints and will continue to be discussed there.

Conclusion

For anyone who is considering picking up or becoming involved with Python or a Python-based project, the change in leadership shouldn't discourage you—in fact, the process so far should encourage all of us to understand more about how this language has been governed and how it will be in the future.

This column is being written months before its publication, so when you finally read this, a lot more progress should have been made towards describing how Python's future may be guided, but the process will still be alive and dynamic and in motion. So this is a great opportunity to alert those of you who may not be aware that this is happening, and to invite those of you who may have filed this under "look at how this is going later" to see how things are going now.

References

[1] Guido van Rossum, "Transfer of Power," python-committers list: https://mail.python.org/pipermail/python-committers/2018-July/005664.html.

[2] Conway's Law: https://en.wikipedia.org/wiki/Conway%27s_law.

[3] B. Warsaw, Python Language Governance Proposal Overview: https://www.python.org/dev/peps/pep-8000/.

[4] B. Warsaw, Python Language Governance: https://www.python.org/dev/peps/pep-0013/.

[5] B. Warsaw, L. Langa, A. Pitrou, D. Hellmann, C. Willing, Open Source Governance Survey: https://www.python.org/dev/peps/pep-8002/.

[6] History for PEP 8002: https://github.com/python/peps/commits/master/pep-8002.rst.

Custom Binaries to Ease Onboarding Using 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

It recently came up that I needed to release a helper tool for our work environment. I was limited with regards to my distribution methods since much of the user base is BYOD based. Having Go in my toolbox, I knew that I could use Go to ease the distribution. One of Go's selling points is its ability to package up all of its dependencies and runtime at compile time, so that you can avoid the runtime dependencies management issues that often arise.

In this case, the executable I built required a lot of configuration information—our list of compute clusters, the authentication endpoint, and an authentication client ID (OAuth 2 based)—so I wrote up and released the documentation on how a user can configure the tool for our environment.

After several weeks with supporting users and the binary, I observed two key behaviors:

1. Every user eventually used the same configuration file, and
2. I needed to regularly update the configuration file as we built, deleted, or moved clusters.

Every time there was a change in the latter, I had to inform the users, publish a new set of documentation, and ask everyone to update. This had mixed success. The extra amount of work, the amount of internal works that were exposed to every user, and the limit of effectiveness of the updates made me look for an easier way to accomplish this.

I started to compare it to another rising situation: mobile device application management. Mobile devices operate under similar circumstances. They tend to be dominated by BYOD. Applications are distributed as large single installs that similarly embed the runtime. The one big difference that I noticed is that with mobile devices, the users are limited with some configuration items. Most configuration items are either compiled into the binary or fetched and cached on the device. Some of those configuration items include secrets such as application identifiers and client tokens.

In an attempt to make life easier, I decided to try moving the configuration around with my helper executable.

In this column, we're going to explore moving the configuration for organizational applications out of configuration files. Along the way, we're going to use this as an opportunity to pick up the AWS object storage, S3, to help us out. We're going to store our basic configuration in an S3 bucket, and we're going to provide access to that bucket by hard coding the access values into the executable.

The code for this example can be found at https://github.com/cmceniry/login/ in the "hardcode" directory. hardcode contains a customizer directory, which is our example application without any organizational-specific configuration.

For this example, you'll need:

1. An AWS account.
2. To create a bucket and upload a sample. If you are new to S3, use this guide: https://docs.aws.amazon.com/AmazonS3/latest/gsg/GetStartedWithS3.html.
3. Set up an IAM user with access keys. This user should have the AmazonS3ReadOnlyAccess policy applied to it (or a more restrictive one if you are familiar with IAM). If you are new to AWS access management, see https://docs.aws.amazon.com/IAM/latest/UserGuide/id_users_create.html#id_users_create_console.

You can also choose to use a different, off-site storage technique, but a point of this article is to learn how to use the AWS Go interface for S3.

Storing Remote Configuration

The first part is to move the bulk of configuration into an external store.

Amazon's Simple Storage Service (S3) is a household name for many working in cloud environments. It provides an authenticated and globally available storage location for static contents. We're fetching our configuration from an S3 bucket. In this example, that configuration is going to be a simple string message, but it could easily be a block of structured data to hold various values.

We're dependent on the AWS Go SDK. You can obtain this with go get github.com/aws-sdk-go/... or using a Go dependency management tool.

Since we're only accessing S3 in one place, we're going to wrap all of that in a single function. It expects four inputs: the access key, its secret key, and the bucket name and path. Instead of returning a value, it will set a global configuration—we'll return to this in the next section.

fetch.go: fetch.

```
func fetch(ak, sk, bucket, path string) error {
```

Inside of `fetch`, we start by configuring an AWS session. This is used for all of the interactions with the AWS APIs. We're going to give it our authentication keys and provide a region for the profile to use. Outside of the static credentials, the AWS SDK does not operate directly on Go types, so we'll need to wrap these with the AWS SDK types.

fetch.go: session.

```
sess, err := session.NewSession(&aws.Config{
    Region: aws.String("us-west-2"),
    Credentials: credentials.NewStaticCredentials(
        ak, sk, "",
    ),
})
```

Before we download, we need a place to download to. Since the application will be using this configuration, we want to keep this configuration in a memory buffer. Underneath the hood, S3 may download across multiple streams and data segments. This means that an ordinary buffer, specifically one that expects just to append to the end, will not work. AWS provides a buffer—aws.WriteAtBuffer—that can be written to in multiple locations at the same time so we can use that.

fetch.go: writeatbuf.

```
writeAtBuf := aws.NewWriteAtBuffer([]byte{})
```

Next we construct the downloader and run the download. The `s3manager.Downloader` is an intelligent transfer manager and capable of downloading many different objects in parallel. In this case, we're just downloading the one object, but we still funnel everything through it. When creating it, we need to tell it our AWS API session so that it has the proper authentication information. `Download` requires a destination—our `writeAtBuf` buffer—and a source—the `aws.String` wrapped bucket name and path or key name.

One thing to note: the `writeAtBuf` parameter passed into download is an `io.WriteAt` interface. This means anything that has a `WriteAt` member method can be used there. For instance, if you were downloading straight to a file, then `os.File` can be used directly since it has the `WriteAt` member method. This is an excellent example of using Go interfaces for flexibility.

fetch.go: download.

```
downloader := s3manager.NewDownloader(sess)
_, err = downloader.Download(
    writeAtBuf,
    &s3.GetObjectInput{
        Bucket: aws.String(bucket),
        Key:    aws.String(path),
    },
)
```

Once we're complete on the download, we then convert that into a string we can use in our configuration. For presentation purposes, we strip leading and trailing whitespace from our value.

fetch.go: config.

```
globalConfig = strings.TrimSpace(
    string(
        writeAtBuf.Bytes(),
    ),
)
```

Again, we stored the configuration in a `customizer`-level variable instead of returning it from the function. As we'll see next, that will help us with our custom application configuration.

Packaging the Configuration Access

Also inside of the `customizer` directory is a `main.go` containing a Main method. This is not a standard Go `main`—it is not in the main package, and it is exported. It is, however, meant to be the entry point for execution of our application. It lacks specific organizational customizations and only has variables to allow for this.

To simplify naming, it takes a `customizer.Options` type. In this example, we just mirror the four items we need to access our S3 bucket. In other situations, this could also include authentication endpoint URLs, specific DNS names, or any other generally unchanging values.

main.go: options.

```
type Options struct {
   AccessKeyID     string
   SecretAccessKey string
   BucketName      string
   BucketPath      string
}
```

This is instantiated as a `customizer`-level variable so that any function inside of `customizer` has access to it. For this same reason, we put our `globalConfig` value from the S3 bucket at the same level. This mirrors how many Go command line tools operate—especially ones that use the Standard Library `flag` or Steve Francia's `pflag` libraries.

main.go: vars.

```
var opt Options
var globalConfig string
```

Once the inputs and variables are established, we can define our pseudo-`Main`. It should be passed an `Options` parameter, which is what will be provided to make an organization-specific application build. The `customizer`-level `opt` parameter is set to the provided `Options` parameter for these values to take effect.

main.go: mainopt.

```
func Main(o Options) {
   opt = o
```

Beyond that, it behaves akin to any `main`, including parts such as command line argument parsing. Since we're pulling additional configuration from S3, we also want to ensure that we perform that as part of this `Main`.

main.go: mainfetch.

```
   err := fetch(
      opt.AccessKeyID,
      opt.SecretAccessKey,
      opt.BucketName,
      opt.BucketPath,
   )
```

Since this is an example, it does not do anything other than print the value of the retrieved configuration file from S3.

main.go: mainprint.

```
   fmt.Printf("Using Configuration: %s\n", globalConfig)
```

Creating a Custom Binary

The `customizer` library can't execute on its own. We need to call it from our own `main.main` method where we pass the specific organizational `Options` values to it.

```
package main

import "github.com/cmceniry/login/hardcode/customizer"

func main() {
   customizer.Run(
      customizer.Options{
         AccessKeyID: "appspecific1",
         SecretKeyID: "orgspecific2",
         BucketName:  "orgspecific3",
         BucketPath:  "orgspecific4",
      },
   )
}
```

Any number of these organization-specific builds can be done, and all end up being approximately the same number of lines of code (depends on the number of options). The marginal effort to create organization-specific builds is limited to ensuring that the configuration items are specified.

Considerations

While this approach certainly aids in the ease-of-use department, there are several considerations and tradeoffs to at least look at. Many exist, but here are some of the more pressing ones.

All of the configuration items in the binary or remote storage should be limited to low-risk items. Low risk is relative, but the rule of thumb is that there is not anything more disclosed than could be available to anyone inside of the organization. Typically, this limits it to coarse-level information disclosures. Conversely, if this opens to arbitrary code execution—e.g., download a binary and run it—you should ensure that the code is signed or validated. It's arguable that no secret should even be hard coded into a binary, especially one that is expected to be widely distributed in an organization. The worry is that this is a slippery slope and encourages bad practices. The balance of security and usability is a constant navigation of slippery slopes.

Any time you use hard-coded values in user applications, you need to account for the fact that you'll have multiple applications in the wild at a time. This means that you'll need to ensure that the use of these configuration items could exist at the same time.

For instance, in this example, AWS supports two API keys per user. This allows you to rotate the key, and both the old and new values are valid while you rotate it.

This is not limited to the server side. If there are validation keys, your application will need to support an array of keys so that the old and new can exist at the same time.

```
customizer.Options{
   VerifyKeys: []string{"abcd", "efgh"},
}
```

For remote configuration, your application will need to support the configuration format of the future. In practice, this means that your application will probably rely on non-strict validation of the configuration data and reasonable defaults when the configuration is unspecified on the remote storage.

You have to decide what goes in the application and what is stored in remote configuration storage. This will largely come down to a question of flexibility. If you expect something to remain largely static or static over a longer period of time, you can put it into the binary. If you expect it to change on a regular basis—at least more often than you want to release binary updates—put it into the configuration repository.

Conclusion

I long held the belief that you should not hard code anything into your binary. If you did, it was a sign that your infrastructure lacked good distribution mechanisms. Best practice was to build and distribute them separately using strong central configuration tools.

Those assumptions came from a specific perspective. That perspective was common, but, with the rise of decentralizing practices such as BYOD and remote work, it has become less so.

Sometimes you have to question your assumptions about best practices. When best practices are established, they are done so in a certain environment. If the environment of the nature of the problem has changed, then the practices need to adjust with them. We're seeing more and more environments where controlling the end device is a very different prospect than it used to be.

Don't be afraid to question the assumptions that you've held. Sometimes you'll find that you're not held to the same constraints that you used to be or that you're not enabled by the same capabilities that you used to be. When this happens, you have to adjust and come up with new best practices.

USENIX Supporters

USENIX Patrons
Facebook • Google • Microsoft • NetApp • Private Internet Access

USENIX Benefactors
Amazon • Bloomberg • Oracle • Squarespace • VMware

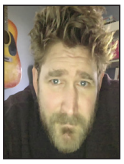
USENIX Partners
BestVPN.com • Booking.com • CanStockPhoto • Cisco Meraki
Fotosearch • Teradactyl • thebestvpn.com

Open Access Publishing Partner
PeerJ

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

iVoyeur
Flow, Part II

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

The bells of Basilika Sankt Kastor clang—a nagging reminder behind me that I should be in Cochem right now, exploring castles like a proper tourist. But my imagination has been hijacked, and so I sit in Koblenz, having failed to switch trains when I realized—looking at the railway map—that this was the city of Deutsches Eck, where the Mosel empties into the Rhine.

The Rhine is the second longest river in Europe (behind the Danube), and yesterday, 100 miles north of here, I watched as a long, low jalopy-looking riverboat meandered up to its bank in Dusseldorf and launched, like a fanout-algorithm, a small flock of half-a-dozen bicycles—mother and children—toward the farmers market, their baskets full of empty shopping bags.

The wide flat deck of the boat was laden with the typical boat-crap-trappings that you would expect to see on the deck of a riverboat, but there were also things foreign to that environment, like a large wooden dining room table with seven chairs, an Iron Man Big Wheel, and lush green potted plants. Through the window of the wheelhouse I could see crayon art and action figures adorning every sill as if on the lookout for inclement weather.

It was love at first sight.

And so here I sit, watching the riverboats navigate the confluence of these two great rivers, most of them laden with cargo or tourists, but some—about one in twenty—serving as someone's home afloat, headed who knows where. I imagine them unhurriedly drifting from town to town, suffering the world to move around them until they come spilling out into the North Sea, and maybe then turning right to explore Amsterdam's maze of canals, or perhaps left, hugging the coastline as far as Le Havre and the mouth of the Seine. I can't help but wonder how the 4G reception is along the great rivers of Europe.

In the US we don't really have any rivers like the Rhine anymore—unencumbered by hydroelectric necessity. Our closest analog is probably a thing called the "Great Loop" [1], which is more of a bucket-list, check-box-excursion sort of thing than a place to live. People who navigate it are called *loopers*, and they traverse a 6000-mile circular "system of waterways" (many of which are man-made) with soulless hyphenesque names like *The Atlantic Intercostal Waterway* and the *Tennessee-Tombigbee Canal*.

I will spare you my rant on the absurd irony of the pork-barrel excavation of navigable waterways in America, in the aftermath of our insane spree of pork-barrel river-damming, and confine myself to pointing out that despite, or maybe *because of* its lack of poetic romance, the American Great Loop, with its overabundance of locks, too-low bridges, VHF signaling, and flood control, is actually far better suited a metaphor to streaming data pipelines and data engineering than rivers like the Rhine.

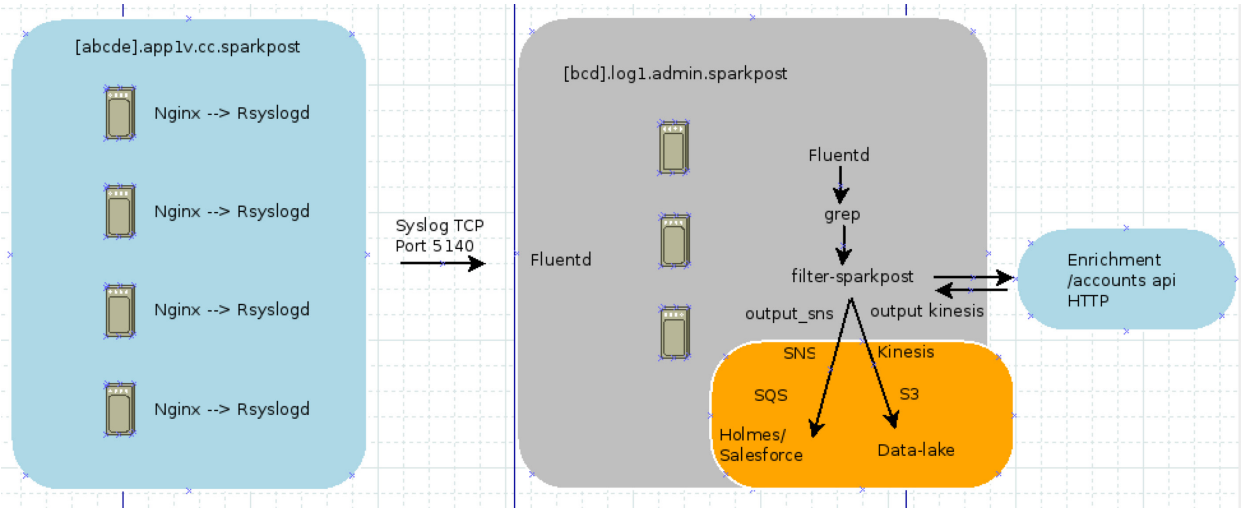


Figure 1: The IEH pipeline architecture

The Loop is a messy, complicated, and artificial route whose waters do not always flow smoothly, in the expected direction, or even at all. Indeed, if data engineering were as pleasant as the great rivers of Europe, STEM graduates would be starving in the streets. But as it is, few people outside of those demented few who yearn for nautical headaches have ever heard of the Great Loop, in just exactly the same way as you, my dear demented reader, are still working your way through this intro in anticipation of my getting to the juicy data engineering bits.

The Flow, Part Two

In my last column, I introduced you to our *Internal Event Hose* ingestion pipeline and *Data Lake* projects at Sparkpost, which together, provide an SQL-like query interface into various types of cherished log data. We covered how schema-at-read systems obviate the need for a proper database and enable us to query any at-rest data set, and we learned about columnar data stores, which make our SQL query engines practical and inexpensive to use (in both the computational and pocket-book sense).

In this article we'll look at what the Great Loop navigators would call *the first leg*, where our log data gets reaped from the instances and undergoes its first transformation into structured data. One of the many well-spring sources of critically important data-flow for us is that of our API servers, whose Nginx processes are configured to send their access logs not to a text file or a syslog server but, rather, to a local UNIX socket like so:

```
access_log syslog:server=unix:/var/run/msys-nginx.sock,facility
=local0 api;
```

Listening to the other side of that FIFO is the local `rsyslogd` process, which is carefully configured to disable all limits, and forward all messages via `syslog/TCP` to an environment-specific logging cluster, which resides behind an Elastic Load Balancer at the split-horizon DNS name: `log.sparkpost`.

```
$SystemLogRateLimitInterval 0
$SystemLogRateLimitBurst 0
$IMUXSockRateLimitBurst 0
$IMUXSockRateLimitInterval 0
$ModLoad imuxsock
$ModLoad imklog
$AddUnixListenSocket /var/run/sys-nginx.sock
local0.* @@log:5140
& ~
```

The day we turned on IEH (Internal Event Hose) in production was (not at all coincidentally) the same day we learned the practical limitations of Nginx's `syslog` outputter, `rsyslogd`, and the UDP `syslog` protocol itself. We service around 11,000 API calls per second in our production environment, a number too great for each of the aforementioned technologies in their original configurations. So Nginx was moved from `syslog`-direct logging to UNIX socket, `rsyslogd` had all of its annoying rate-limits disabled, and `udp/syslog` transport to the logging servers was replaced with `tcp/syslog`.

Listening on port 5140/tcp on the logging cluster is a log-processing framework called Fluentd. You may think of Fluentd as an event-router. You provide routing targets and addressing, and Fluentd routes incoming events accordingly. Our high-level architecture looks like the diagram in Figure 1.

In the configuration, a source block defines a listening port. Routing instructions in Fluentd-land are called *tags*, so the following source block listens for `syslog` protocol on `tcp/5140` and tags everything that arrives as routable to *firehose*.

```
<source>
  @type syslog
  port 5140
  protocol_type tcp
  bind 0.0.0.0
  tag firehose
</source>
```

The *firehose* tagged event's first stop is to a built-in Fluentd plugin called *parser*. The parser plugin's job is, predictably, to parse each plaintext line into a JSON blob of named fields. To do this, it needs a Ruby-syntax regular expression with named fields. For our particular Nginx log format, the Fluentd config looks like this:

```
<match firehose.**>
  type parser
  key_name message
  format /^(?<ts>.*?) "(?<remote_addr>.*?)" (?<response_code>\d+)
    "(?<request>(?<method>.*?) (?<path>.*?) (?<version>.*?))"
    "(?<key>.*?)" (?<key_type>.*?) (?<customer>.*?)
    (?<username>.*?) (?<response_time>.*?) (?<bytes_sent>.*?)
    (?<length>.*?) (?<tenant_id>.*?) (?<subaccount_id>.*?)
    "(?<upstream>.*?)" "(?<user_agent>.*?)" (?<cache_status>.*?)
    "(?<entity_id>.*?)"$/
  tag firehose_parsed
</match>
```

Every plugin begins with a `match` or `filter` parameter that names tagged `fluentd` should route to it. At this point, given the big hairy regex, you might be wondering about the computational overhead of Fluentd, and my answer would be that the system is internally threaded, partially implemented in C, and surprisingly resource-efficient. Although we initially had problems getting the traffic load stable across the network boundaries (as mentioned), we've had no problem running our workload on a set of three (one-per-AZ) modest instances.

Once a given log line has traversed the parser plugin, it exists in a parsed state to the rest of the plugins in the chain. In other words, we can now refer to the individual fields of our log lines using the names we assigned them in the regex we provided to the parser plugin. For example, I can see a given event's response code by specifying `event['response_code']`.

You'll notice the events are now tagged *firehose_parsed*. These get routed to the next filter in our config, which is a custom filter that we wrote ourselves in Ruby (all custom Fluentd filters are Ruby).

```
<filter firehose_parsed.**>
  @type sparkpost
</filter>
```

A simple enough configuration, since Fluentd doesn't know anything about it other than to import our Ruby script and provide it with events via the pre-ordained `filter` function, as described in the Fluentd documentation on writing custom plugins [2].

Our custom filter performs a slew of business-oriented tasks on the event flow. First, we use it to scrub the Nginx data, deriving new attributes from existing ones. For example, the `event['path']` contains the entire path from Nginx, including things like CGI query parameters. In our custom Fluentd filter, we can split these out like so:

```
fixed_path = String(message['path'].split('?')[0]),
query_string = String(message['path'].split('?')[1]),
```

Some events represent API calls that are *special* in a business-sensitivity context, like new-user sign-ups or account deletions. Our custom plugin extracts these events, *enriches* them with data derived from follow-up API calls, and then forwards them to Salesforce and other internal tools by way of AWS SNS (Simple Notification Service).

```
<match ieh_enriched.**>
  @type amazon_sns
  flush_interval 5
  num_threads 20
  buffer_type file
  buffer_path /tmp/td-agent/amazon_sns
  topic_name internal-event-firehose-prd
  aws_region us-west-2
  sns_message_attributes_keys {"enriched":"enriched", "event_
type":"type"}
  add_time_key true
</match>
```

These *special* events are tagged `ieh_enriched` from within our custom plugin. You might notice that there is quite a bit of buffer configuration in this SNS output block. Although we haven't had scalability problems with Fluentd itself, we have found that handoffs to external services like AWS SNS and Kinesis can be fragile. It's taken some time to get the buffer settings locked in for our particular workload, and you should expect a similar experience.

You might be curious about the `sns_message_attributes_keys` parameter. This parameter implements AWS-side SNS filtering [3]. I bring it up because there are two widely used third-party Fluentd filters today. One of them is an unbuffered (read: dangerous) plugin that supports SNS filtering, and the other is a buffered plugin that does not support filtering. What the world in fact needs, is a single, buffered, SNS plugin that supports AWS-side filtering.

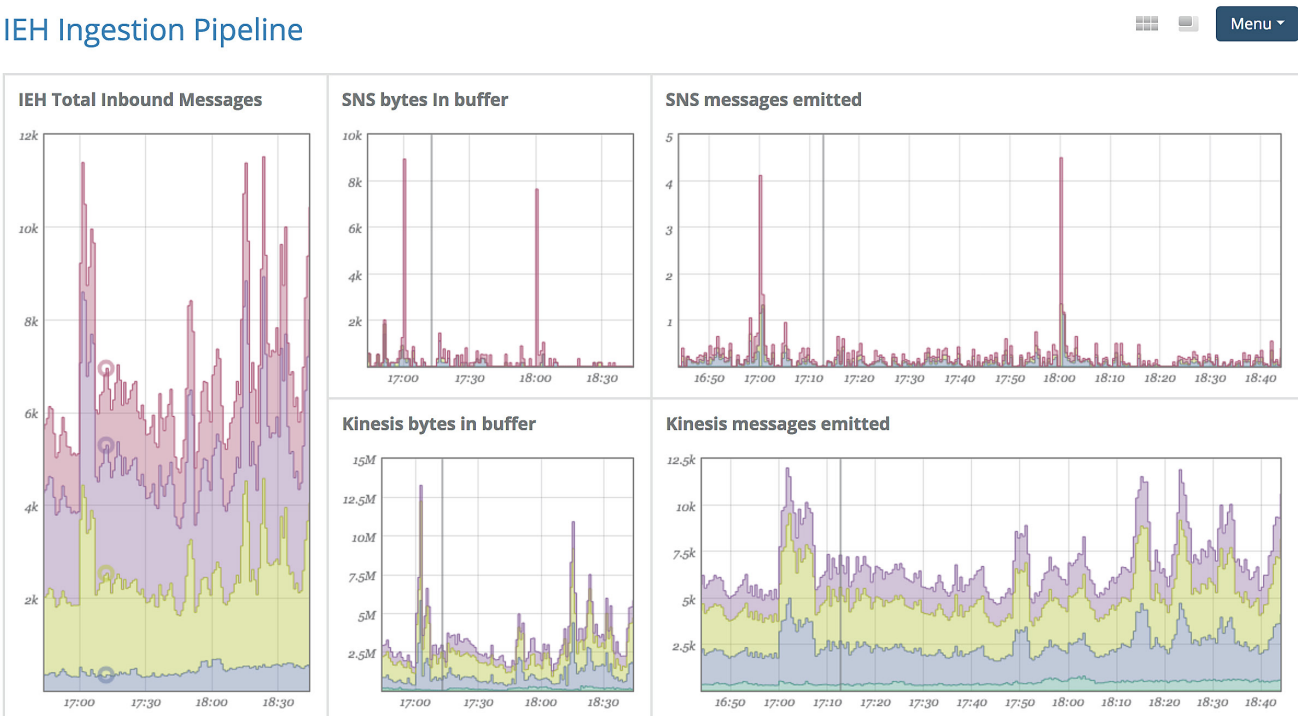


Figure 2: IEH metrics, derived from the Fluentd-Prometheus plugin

To that end, I've forked and extended the buffered plugin with filter support for our workload at Sparkpost and have PR'd the result back upstream [4]. Hopefully, by the time you actually care about this it'll be merged in.

Finally, every event, regardless of whether it is special or not, gets forwarded to the data-lake via AWS Kinesis Firehose. These events keep the `firehose_parsed fluentd` tag and are routed to this output config block:

```
<match firehose_parsed.**>
  @type kinesis_firehose
  region us-west-2
  delivery_stream_name internal-event-firehose-prd
  append_new_line true
  num_threads 64
  flush_interval 1
  buffer_type file
  buffer_path /tmp/td-agent/kinesis_firehose
  buffer_chunk 8388608
  buffer_queue_limit 512
</match>
```

This plugin is an AWS-supported plugin for Fluentd [5] and works very well. We still needed to carefully balance Fluentd's buffer behavior to our workload. Some things to point out here are the `append_new_line` feature, which places a new line between each event rather than just firing a huge incomprehensible JSON blob of 100 smushed-together events into Kinesis, which subsequent data tooling like Glue will not be able to parse. I point this out to you as someone who had to perform a manual retroactive data-reload on several weeks' worth of incomprehensible JSON data.

A few words about Fluentd buffers: first, study the diagram in the Fluentd documentation [6]. There are buffers, and there is a queue, and they are different entities with unique behaviors, log-errors, and configurations. In production you want to use file-based buffers. They are fast enough for sub-second data flushes (remember, buffered output is threaded) and survive better in the event of an instance/server failure. Finally, consider your buffer chunk sizes carefully, especially how long it takes to fill a chunk, because internal buffering can be a source of massive time-delay for low-frequency events.

I'm out of space for this issue, but we've gotten through pretty much all of *the first leg* save for our favorite subject: monitoring. The easiest means of introspecting Fluentd's behavior is using the Prometheus plugin for Fluentd. You don't need to be using Prometheus to use the plugin—in fact, I'm currently using Circonus to visualize metrics from it as you can see in Figure 2.

Next time, I'll walk through the (quite excellent) Prometheus plugin's configuration and start you out on leg two of our journey, where we'll stream our parsed event data into S3 and use Apache Spark to transform it into Parquet format.

Take it easy.

References

[1] The Great Loop: https://en.wikipedia.org/wiki/Great_Loop.

[2] Fluentd plugin docs: https://docs.fluentd.org/v1.0/articles/api-plugin-filter.

[3] AWS SNS filtering: https://docs.aws.amazon.com/sns/latest/dg/message-filtering.html.

[4] Fully buffered SNS plugin: https://github.com/miyagawa/fluent-plugin-amazon_sns/pull/11.

[5] Fluent plugin for Amazon Kinesis: https://github.com/awslabs/aws-fluent-plugin-kinesis.

[6] Fluent buffers: https://docs.fluentd.org/v0.12/articles/buffer-plugin-overview.

Statement of Ownership, Management, and Circulation, 10/01/2018

Title: *;login:* Pub. No. 0008-334. Frequency: Quarterly. Number of issues published annually: 4. Subscription price: $90.

Office of publication: USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710.

Headquarters of General Business Office of Publisher: Same. Publisher: Same.

Editor: Rik Farrow; Managing Editor: Michele Nelson, located at office of publication.

Owner: USENIX Association. Mailing address: As above.

Known bondholders, mortgagees, and other security holders owning or holding 1 percent or more of total amount of bonds, mortgages, or other securities: None.

The purpose, function, and nonprofit status of this organization and the exempt status for federal income tax purposes have not changed during the preceding 12 months.

Extent and Nature of Circulation			**Average No. Copies Each Issue During Preceding 12 Months**	**No. Copies of Single Issue (Fall 2018) Published Nearest to Filing Date**
a. Total Number of Copies *(Net press run)*			2436	2775
b. Paid Circulation *(By Mail and Outside the Mail)*	(1)	Mailed Outside-County Paid Subscriptions	974	888
	(2)	Mailed In-County Paid Subscriptions	0	0
	(3)	Paid Distribution Outside the Mails	693	652
	(4)	Paid Distribution by Other Classes of Mail	0	0
c. Total Paid Distribution			**1667**	**1540**
d. Free or Nominal Rate Distribution *(By Mail and Outside the Mail)*	(1)	Free or Nominal Rate Outside-County Copies	77	77
	(2)	Free or Nominal Rate In-County Copies	0	0
	(3)	Free or Nominal Rate Copies Mailed at Other Classes	18	17
	(4)	Free or Nominal Rate Distribution Outside the Mail	374	330
e. Total Free or Nominal Rate Distribution			**469**	**424**
f. Total Distribution			**2136**	**1964**
g. Copies Not Distributed			300	811
h. Total			2436	2775
i. Percent Paid			78%	78%
Electronic Copy Circulation				
a. Paid Electronic Copies			468	466
b. Total Paid Print Copies			**2135**	**2006**
c. Total Print Distribution			**2604**	**2430**
Percent Paid (Both Print and Electronic Copies)			82%	83%

I certify that the statements made by me above are correct and complete.

Michele Nelson, Managing Editor 9/28/18

For Good Measure
Nameless Dread

DAN GEER AND PAUL VIXIE

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, includi ng officer positions, program committees, etc. dan@geer.org

Dr. Paul Vixie is an Internet pioneer. Currently, he is the Chairman, CEO, and cofounder of Farsight Security, Inc. He was inducted into the Internet Hall of Fame in 2014 for his work related to DNS. Dr. Vixie began his career as a programmer (Cron, RTTY, BIND) before co-authoring *Sendmail: Theory and Practice* and more than a dozen RFCs, and contributing a chapter to *Open Sources: Voices from the Open Source Revolution*. More recently, he has become a serial entrepreneur (ISC, MAPS, PAIX, MIBH, DNS-OARC, Farsight). He was a member of the ARIN Board from 2004–2013. He completed his PhD in 2010 at Keio University. vixie@fsi.io

> What's in a name? That which we call a rose
> By any other word would smell as sweet;
>
> *Romeo and Juliet*, Act 2, Scene 2

Each generation of global commerce and culture has to decide for itself what the Internet "means" to them. Some of that meaning will depend on how large the Internet is at that time. Delightfully, the unit of measure of that largeness will also change with every era.

There was a time when to be "on the Internet" meant that your host's name was published in a central registry called HOSTS.TXT—and then the wheels came off. The original text-only terminals were replaced by graphical workstations, later by personal computers, then by virtual servers, followed by smartphones, and, eventually, smart devices. But whereas the time-shared minicomputers that once serviced text-only terminals had names, the workstations and personal computers that came later were given names mostly out of habit: we wanted to know where connections to our time-shared computers and servers were coming from, but we would rarely have any reason to try to connect back to those origins.

There was also a time when to be "on the Internet" meant that your IP address block was present in the global routing table. Those wheels also came off pretty early: network address translation (NAT), whether deployed as a security measure or due to a real or perceived shortage of address space, meant that only a small island of a university or enterprise network would use so-called "global addresses," and these would act as gateways to private networks that serviced a much larger population of possible endpoints hidden behind such gateways.

In 2018 ("now") the fashion is to measure the number of connected people and not the number of connected devices. We round this number to the nearest billion, as if we neither know nor require any further accuracy.

Because it's hard to secure something we don't understand, it's necessary that we *fathom* the Internet in some way, so that we can account for and predict the risks it poses and the risks it experiences, and ultimately make some plan as to how to manage some risks and how to cope with others we cannot manage.

Scale

The Domain Survey, operated since 1987 by Network Wizards, Internet Systems Consortium, and 3Way Labs, gives us a general baseline of one measure of Internet size: the population of endpoints that have names. Notably, not all of these names are actually used—many are assigned by network operators from a pool of machine-generated and meaningless names with no expectation that any of these names will ever be used to initiate a connection. This is due to ancient prejudices whereby a service might reject as "low value" any connection from an endpoint lacking a name. Even though this prejudice is wrong, the optics generated by its adherents have helped chart the growth of the Internet to a population size just over

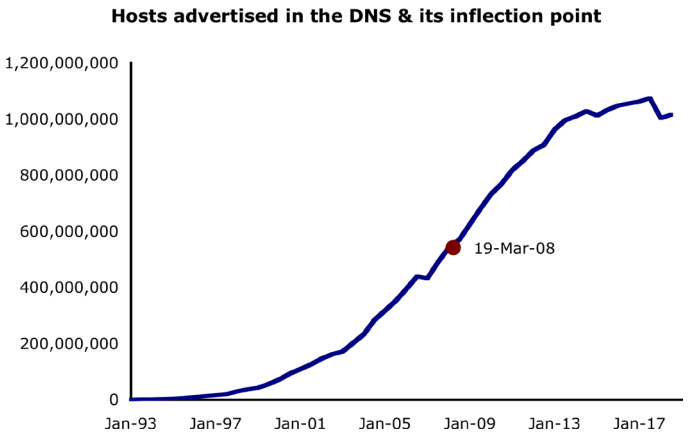


Figure 1: Hosts advertised in the DNS, and its inflection point [1]

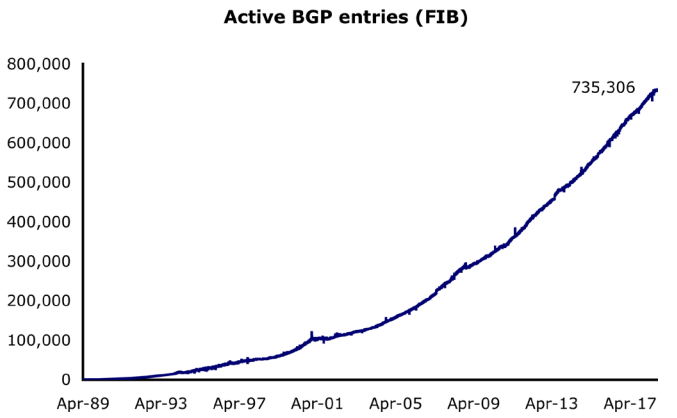


Figure 2: Active BGP entries [2]

one billion "endpoints having names," as seen in Figure 1. We highlight the logistic inflection point, 19-Mar-08, the point in time at which the rate of growth in advertised names changed from accelerating to decelerating. As of today, 66% of the total IPv4 space is advertised as compared to 0.0026% of the total IPv6 space.

Measurement of the Border Gateway Protocol (BGP) global routing table is another proxy for some measurement of the Internet's size. A single entry in this table can contain as few as 256 potential endpoint addresses or as many as 16 million. We can constrain our estimate of the average number of potential endpoint addresses in a routing table entry by noting that about three billion endpoint addresses are globally reachable, there is no new IP version 4 address space remaining in the free pool, and the global routing table contains about 750,000 entries. So a routing table entry represents, on average, perhaps 4000 potential endpoints. In CIDR terms that's a "/20." Notably, many of the smallest routing table entries are just NAT gateways, and so each might represent a vast population of endpoints that could reach outward or accept inbound transactions (see Figure 2).

In 2018, mobile Internet devices such as smartphones began to reach a saturation point—most humans who want or need and can afford a mobile Internet device already have several of them, which means device sales are now principally for upgrades and replacements (see Figure 3). The market is still strong with vigorous competition between handset and platform makers, but the decade of Internet growth driven by new mobile Internet devices may be reaching a plateau. Notably, the vast majority of mobile Internet devices do not have resolvable names since they are only outbound traffic sources and not also inbound traffic sinks. Most do not have fixed addresses and will make their outbound requests from a new address every few minutes due to mobility, roaming, or virtual network grooming.

The fastest source of Internet growth since 2015 is the Internet of Things (IoT), and this is expected to continue, more or less forever (see Figure 4). A "thing" in this context can be a home appliance, an embedded device, or a component in some system like home audio. These devices are cheap to build and cheap to buy, such that very little thought goes into life-cycle management either by producers or consumers of these tiny and plentiful devices. Many such devices are shipped with known or discoverable security vulnerabilities, and many will never be patched whether because of supply chain churn or because the resulting software engineering economics would drive unprofitability. Most importantly, precisely none of these devices have names.

Internet Protocol version 6 (IPv6) has come a long way in a short time, and now represents about a quarter of all inbound traffic seen by Google's network [5]. This fraction is characteristic of other cloud, search, Application as a Service, and social network providers. There is no confident estimate of the relative size of the IPv6 vs. IPv4 endpoint populations due to technical differences in the format and allocation of endpoint addresses between the v6 and v4 systems. Generally speaking, it's easier to enable IPv6 in a new device or online system than to add

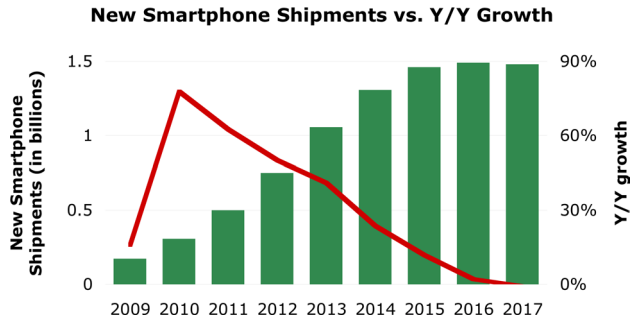


Figure 3: New smartphone shipments vs. year-to-year growth [3]

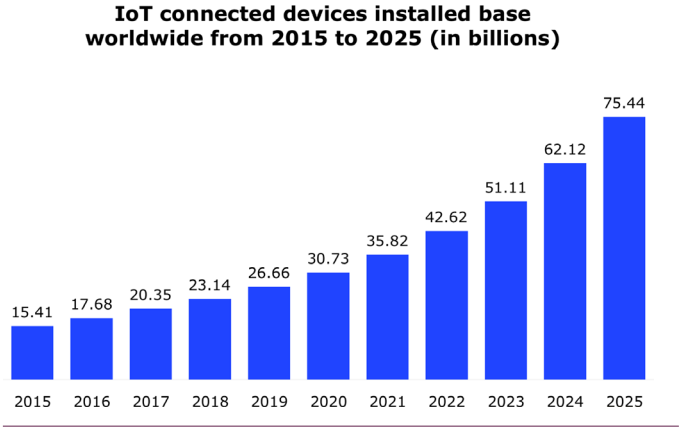


Figure 4: IoT-connected devices installed base worldwide from 2015 to 2025 [4]

IPv6 to an existing IPv4-only system, which dovetails with the trend toward seamless automation without end-user configuration or awareness. While many endpoints from the two largest populations (mobile Internet and IoT) are now using IPv6, most of their traffic is outbound-only, and these devices rarely have or require names. One hopeful difference between the IPv4 and IPv6 systems is that IPv4 addresses are dense enough to permit brute force automatic network scanning by an attacker, so even an endpoint that never advertises its presence and has no name might still be attacked. The sparse addressing of IPv6 makes this kind of attack far more expensive in terms of brute force than for dense IPv4. Of course, security regimes that walk the corporate address space to discover what addresses exist on "their" network are similarly disabled by IPv6's sparseness.

Implications

Security risk is a function of defects and vulnerabilities, exposure, opportunity, and motivation. Factors like the relative motivations and skills of defenders vs. attackers can often be more decisive than the number of defects or the overall reachability of a victim endpoint. However, when other things are equal, as they tend to become in a maturing market with established equilibriums, the best predictors of risk are exposure and reachability. A device that never receives inbound messages from any other device can contribute very little risk. Of course, outbound-only means that it is infeasible to push messages to that device—an auto-update process has to be initiated by the remote device asking to be updated, for example, or a reserve channel has to be secretly designed-in.

We have placed special focus on names because, for security analysis, a name makes a device more reachable, thus increasing its exposure. If successfully attacked, a device will often become a beachhead by which other more private and less reachable devices can be probed and perhaps also successfully attacked, thus increasing the risk posed by the exposed device. Having a name is a risk factor, just as being reachable from outside the local network due to firewall weaknesses or misconfiguration is a risk factor.

Mobile devices can and do join botnets. But the initial vector for a successful attack on such devices will invariably be that it was induced to make an outbound transaction whose results were damaging in some way and against which the device had no working defense. The same will be true of IoT devices for the most part, although in this class of victim, inbound transactions either from the local network or from selected parts of the outside world are part of the product design, and in that case a name, either in the domain name system (DNS) or some other less public naming scheme, will contribute to reachability and therefore to overall risk.

Defenders should consider a mostly closed reachability policy. Nothing should be externally reachable unless there is a hard requirement. This includes both giving an endpoint a globally resolvable DNS name and giving it any kind of reachability in the firewall configuration. But more than this, internal firewalls have to be deployed so that a successful attack on one part of the network does not necessarily create a beachhead for attacks on the rest of the network. This kind of internal segmentation is costly, but at least it's an up-front cost that defenders can budget for—much cheaper than answering questions from the press, customers, shareholders, or regulators after a successful attack—plus whatever damage was actually caused.

There are far-reaching design questions here. One involves the resurrection of a 20-year-old debate: assuming that myriad, nameless devices will need to be able to cryptographically protect their messages, where is the key for that looked up? Will each device have one of its own? Will internal firewalls include a key-centric, rather than a name-centric, PKI of sorts [6]? Does a MAC address or UUID-in-ROM distinguish keys in a nameless world and thus imply an identity-based PKI? Either way, is the key-management job going to be harder or easier absent names? Will we not bother with keys at all and trust that the internal firewalls are resilient to lax operation? Perhaps especially interesting, what would a name mean when the end user has a half-dozen devices that mutually self-synchronize?

Evolution

Because small nameless devices tend to be cloud-associated but typically do not accept inbound transactions or connections, they will (by design) make long-running outbound connections to their maker's command and control infrastructure and simply wait to be told over that connection what action or report they should make next. The identity of the device might be encoded as a client-side TLS certificate or some hardware serial number.

The command and control service will associate the device's identity with a subscriber, and when the subscriber also connects in, this elbow-shaped pair of connections will allow the subscriber to apparently but indirectly control their device. This synchronization-design language is both the result and supporter of the trend toward namelessness in modern Internet-connected devices. Even where direct LAN-based connectivity is used to connect a subscriber to a device, it will as often be negotiated through the maker's command and control network, as discovered locally by some broadcast or multicast protocol along the lines of mDNS or UPnP. Whatever the motive or method, the universal consensus among system designers is that using names to reach Internet-connected devices is considered a legacy. Services need names; servers who provide those services need names; devices which are not servers, will be reached in other ways.

We are at a fork in the road. The choices to be made will be expensive to later reverse in either dollars or clock-ticks. Momentum says that, soon, the majority of Internet endpoints will not be describable by name or discoverable by scanning. Another layer of indirection will, as ever, solve some problems and create others. Provenance and forensics will all but surely be affected. The CAP theorem [7] is licking at our heels.

References

[1] Internet Domain Survey, July 2018: ftp.isc.org/www/survey/reports/2018/07/index.html.

[2] Active BGP entries: http://bgp.potaroo.net/as2.0/bgp-active.txt.

[3] New smartphone shipments vs. growth: https://www.slideshare.net/kleinerperkins/internet-trends-report-2018-99574140, slide 6.

[4] IoT-connected devices: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/.

[5] Proportion of IPv6 traffic: https://www.internetsociety.org/wp-content/uploads/2018/06/IPv6-infographic.pdf; https://www.google.com/intl/en/ipv6/statistics.html.

[6] Name-Centric PKI (Ellison & Metzger) vs. Key-Centric PKI (Ford & Kent): http://static.usenix.org/publications/library/proceedings/ec98/pki.html.

[7] Simon S. Y. Shim, "The CAP Theorem's Growing Impact," *IEEE Computer*, vol. 45, no. 2, February 2012, pp. 21–22: https://www.computer.org/csdl/mags/co/2012/02/mco2012020021.pdf.

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

Kurt Andersen, *LinkedIn*
kurta@usenix.org

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

Amy Rich, *Nuna Inc.*
arr@usenix.org

EXECUTIVE DIRECTOR
Casey Henderson
casey@usenix.org

/dev/random
Simulation Station

ROBERT G. FERRELL

Robert G. Ferrell, author of *The Toi Chronicles*, spends most of his time writing humor, fantasy, and science fiction. rgferrell@gmail.com.

I was planning to discuss SRE this time, but as I was looking up that abbreviation I realized I don't actually know anything about it. In my day, "site reliability engineering" meant you used four empty heavily caffeinated soda cans to support the particle board shelf with the server on it instead of the customary three. Fortunately for the admittedly tiny segment of my audience who labor under the misapprehension that I'm an authority on anything outside the realm of sarcastic goblin detectives, I stumbled across the Joe Rogan interview with the Muskmeister and fell headlong into his metaphysical rabbit hole of cosmological solipsism.

The idea that we are all in a simulation is hardly novel. It was around long before the Wachowskis spun it into great glittering mounds of platinum. Descartes' 1641 brain-in-a-vat idea, for example. Most science fiction writers worth their salt have taken a crack at it over the past five or six decades, William Gibson's matrix in *Neuromancer* and Neal Stephenson's metaverse from *Snow Crash* being perhaps the most famous examples, although technically neither of those were fully immersive worlds that constitute a separate reality. Since *The Matrix*, two of my favorites have been the Framework (*Agents of SHIELD*) and virtual *Eureka* (http://eureka.wikia.com/wiki/Season_5). There are no doubt others I have yet to discover because I'm about 10 to 15 years behind everyone else in my media consumption. I mean, I just recently finished my first time all the way through both *Smallville* and *Buffy*.

As a science fiction and fantasy author, I've noticed that the immersive simulation concept is one that carries my imagination along like Class 5 philosophical rapids. The versions I've seen in fiction assume that some higher life form is pulling the strings that keep some other life form(s) trapped in a simulated reality, but that reality is always limited in geographic scope to, at most, a single city. They also usually predicate their narratives on the idea that, given the right circumstances, the victims can break free and return to the "real" world.

In video gaming, there is a concept called "clipping." It means that objects have no inherent persistence: the game's engine only needs to render whatever is in the players' current fields of vision. It tracks where they are looking and calculates what they can observe from a given perspective. Similarly, our own simulation engine only has to render what we can see right now. When we look away from something, it just disappears. In the "real world," objects are effectively only photons bouncing off a surface and impinging on our optic nerve via the retina. Whether or not they have mass and occupy space when we aren't looking is a moot point.

If the universe has been in existence for over 13 billion years, there's been plenty of time for a civilization to arise and reach the point where it can create simulations powerful enough to be indistinguishable from reality. But it doesn't have to create an entire objective universe, with all the pieces in place—not by a very long shot. It only needs to plant an illusion of this in our putatively sentient brains. The objective perception of this shared universe doesn't even have to be truly identical: all that matters is that those brains believe the consensus exists. Solipsism is the ultimate restriction on external truth. So long as we perceive shared

consistency in the application of universal "constants," we will believe ourselves to comprehend them, no matter how far from fact that belief may stray.

This consistency of perception is trivial for an extremely advanced civilization to engineer. The evidence we can see suggests that the universe is 13.8 billion years old, yes, but that evidence—or the perception thereof—might just be another part of the simulation. Perhaps after all this time there is only one sentient species left in the entire multiverse, and just for grins it controls a large number of simulations full of simulated people who only think they exist in objective reality. The motivation for this might be some far-reaching sinister purpose involving harvesting us or thriving on our triggered endocrine secretions, or it could be as simple as entertainment. I mean, once you've mastered fabricating then enslaving another entire species by manipulating their very perception, you might be tempted to allow others to tune in and watch the show, for a fee. This of course takes "reality programming" to an entirely new level.

Or we could just remove the overseer from the picture completely and envision a scenario where the species that designed and built the simulations has long since gone extinct, leaving the self-perpetuating virtual multiverse ticking away all on its own. The existential question then becomes, does it matter? If instead of being a collection of cells producing proteins, splitting ATP into ADP to generate energy, and transporting various ions back and forth across membranes, I am nothing more than a parent process with a bunch of subroutines running on some vast CPU, does that really make any difference? I require petaflops; therefore, I am.

Whether a "red pill" could even exist depends, then, upon whether there is any existence outside the simulation to which to exit. If we're all floating in some tank with electrodes taped to our foreheads that's one thing, but if we're merely computational avatars, that's quite another. Maybe we already destroyed our planet utterly in a nuclear holocaust, or by abusing the environment to its breaking point, and an alien species came across our dead civilization. They analyzed our culture from the archives we left behind and reconstructed it as a simulation with variables they control in order to learn about our society in a laboratory setting. They're watching to see where we went wrong so they can warn other similar civilizations, again perhaps for a fee. Maybe we keep getting reset to some point in the past to play out the same doomed self-destructive track as a cautionary tale for each new class that comes though an alien social psych course.

Being in a cosmic simulation also raises interesting questions about death. I mean, is the end just "kill -9," or is our thread diverted to another core and we continue in a new program fork? Would we even be aware of this change? If every aspect of our existence is hard-coded, that means predestination is real. On the other hand, if our code is heuristic and adaptive, maybe we can determine our own destiny, at least within the greater programming context. Are we closed loops or fractal sub-threads? Is our will truly free, or are we prisoners of our mallocs? If we could read our own header files, what would they tell us? Would they be like lines on a palm, laying out our futures? Is a massive heart attack merely a divide-by-zero error? Maybe in searching for the meaning of life, the universe, and everything, 101010 is just what you get when you de-reference the pointer to human existence.

Personally, I suspect that rather than some high and noble pragma, human society is nothing more than a streaming event for an interstellar entertainment network, like watching *Civilization VI* on Twitch. Trillions of beings across the local cluster are laughing right now as I type this. "The monkey figured it out!" a viewer howls between bites of whatever snack food appeals to hyper-intelligent liquid methane-based squamoids.

"Hey," interjects another, "this is boring. Change the channel to that white hole cluster cam. I love scoping the bizarre crap that slides out of those things." Because, you know, that's how hipster superior beings talk.

If I am just the result of compiled code running on some processor, all I ask is that there be decent garbage collection. Corrupted memory does not lead to sexy fun times.

Homo barada nikto, y'all.

Book Reviews

MARK LAMOURINE AND RIK FARROW

Groovy in Action, 2nd ed.

Dierk König and Paul King
Manning Publications, 2015, 912 pages
ISBN 978-1-93518-244-3

Reviewed by Mark Lamourine

I have been learning Groovy as part of some recent work using Jenkins pipelines for a build-and-test system. The Jenkins pipeline plugin implements a Domain Specific Language (DSL) for job control that extends (and, to some degree, limits) the Groovy language. I started off just reading examples and using search engines to answer my questions. When it became clear to me that this was going to be a regular part of my work, I decided I needed to do some proper reading.

Groovy is a scripting language and execution environment based on Java that first appeared in 2003. The first edition of *Groovy in Action* was released three years later. The second edition covers Groovy 2.4. The current version of Groovy, as of this writing, is 2.5, and a version 3 is in development. The Jenkins pipeline plugin appears to be based on Groovy 2.4.

To be clear, *Groovy in Action* is strictly concerned with the Groovy language and does not treat Jenkins or the pipeline plugin at all. Understanding Groovy is required but not sufficient to write Jenkins pipeline jobs. There's also a lot more to Groovy than Jenkins.

Groovy in Action feels smooth. The progression of topics is clear and logical. König and King know their audience and write well to them. There are no tutorial digressions into theory or foundations. They do make note of things that might be unexpected or unfamiliar, like the concept of optional typing. Groovy is tightly coupled to Java, and the authors assume a level of comfort with Java and with modern programming practice and terminology.

The first section covers the Groovy language proper. I like the grouping of data types into Simple and Collective. Placing a complete chapter on closures before the chapters on control structures and objects breaks the ordering I'd expect, but it makes sense when seen in this context. Closures are no longer an exotic idea, and they are a big part of idiomatic Groovy.

The language section closes with a couple of chapters on dynamic programming and static typing in Groovy. Groovy is an "optionally typed" language: you can specify the types of variables and functions, or you can let the compiler infer them. It is not a dynamically typed language. That is, once you or the compiler have determined the type of a variable, it cannot be changed. Specifying the type allows the compiler to flag mismatches, but you'll get a runtime error if you try to change the type of an inferred object.

In the second section the authors start making some real work possible. The chapters cover how to work with databases, structured data, and web services. Groovy includes something I haven't seen before, called "builders," which are used to create hierarchical data structures without a lot of the boilerplate. Builders can create in-memory trees or structured data like XML or JSON or even HTML. I'm going to be giving this a closer look.

König and King finish up with a section of practicum. This includes chapters on unit-testing, and interacting with the underlying ecosystem. I group the appendices in this as well. These cover installation of Groovy and some great cheat sheets for language constructs. It also includes a small section on the operational steps of the Groovy compiler and a list of compiler annotations, which I find useful and unusual. I like knowing one layer beneath where I'm working.

Groovy in Action, 2nd ed. introduced me to a new way to interact with the Java environment that felt smooth, seamless, and comfortable. Groovy might prove useful for prototyping Java. I don't know if it is a reasonable replacement for Java, but it's certainly easier to use and debug. If you need to work with Jenkins pipelines, I'd consider it invaluable and am using it daily.

Improving Agile Retrospectives: Helping Teams Become More Efficient

Marc Loeffler
Addison-Wesley Professional, 2018, 272 pages
ISBN 978-0-13-467834-4

Reviewed by Mark Lamourine

For most people the Agile retrospective is the least appealing part of the development process. The idea of re-hashing all the things that have gone wrong can lead people to find fault and lay blame. Few people are totally comfortable with having others put their mistakes under a microscope. In *Improving Agile Retrospectives*, Loeffler shows that there is another way (several others, in fact) to draw out experience and make use of it without resorting to finger pointing and personal judgment.

You might not think there's that much to running a meeting that shouldn't last more than an hour, but unlike most meetings, the retrospective has the potential to go very wrong. Loeffler devotes the first half of the book to describing and understanding the purpose of the retrospective and the role of the facilitator. The

retrospective will often require some preparation and thought to create the right atmosphere for candid discussion. Most won't need the full treatment Loeffler offers, but these chapters offer a good toolbox.

The purpose of a retrospective is to understand what happened in the recent past; how reality matched and differed from the plan. It should also be a time to recognize and celebrate what went well. There is always something that either didn't go as planned, or was harder than expected. Chapter 1 sketches a skeleton for the meeting, creating an arc from opening to closing. Loeffler provides an agenda of five phases and a timeline for each. An hour isn't very long, and it's important to keep the process moving.

The next three chapters frame the job of the facilitator. Chapter 2 covers preparation and planning. Chapter 3 walks through the first retrospective, touching on the process of guiding and moderating the meeting and the goals for each phase. Chapter 4, the longest in the book at 35 pages, details the role of the facilitator in the process: how to prompt and guide the discussion while keeping the focus on the team and on constructive participation. A good facilitator is central to the process, but should never become the focus of the meeting. There are goals beyond the production of recommendations and action items. At the end of the ideal meeting, all of the participants should feel that they have been heard and that the results represent the consensus of the team. Items that can't be resolved can be tabled but should not be dismissed. Nothing ever ends up ideal, but the result can still be satisfying.

The real surprise to me was the second half of the book. It turns out that there are quite a number of ways to frame the process of reflection. Retrospectives can have different scopes and goals, and these may require different techniques. A single-team end-of-sprint meeting doesn't compare with a project-management-level post-release review. Some of the metaphors offered here may not work for all audiences. The props needed for the "kitchen retrospective " might not set the right tone for a director-level quarterly meeting but could be just the thing to open a team session with a light, fun tone. It's easy when running a retro every two to three weeks to fall into mental ruts, and a change-up of the format can freshen people's engagement and interest.

Whether you've been asked to moderate a retro meeting or are a participant, *Improving Agile Retrospectives* will give you a better handle on how to participate and get the most from the process. Hopefully, it will encourage both doubters and advocates for the benefits of the retrospective process to participate with an open mind.

The Manga Guide to Cryptography
Masaaki Mitani, Shinichi Sato, Idero Minoki, and Vert Corp.
Omsha Ltd. and No Starch Press, 2018, 234 pages
ISBN: 978-1-59327-7

Reviewed by Mark Lamourine

You might be forgiven for assuming that *TMGTC* is aimed at middle schoolers or teen students. They might be drawn in by the cover and the artwork in the first few pages, but it's not long before the real math and logic come out to play.

There are characters and a story line that interlace with the exposition, but they really just frame the real lessons. If the framing entices a reader to start and continue, then they've done their job, but they don't add much to understanding the content.

The contents are actually a good treatment of the basics of modern encryption. The four sections cover foundational ideas, symmetric and public-key ciphers, and close with practical applications. These last should be familiar to most Internet users today: user identification and authentication, content encryption and validation, and e-commerce.

Another surprise is the breadth and depth of the coverage. The first chapter introduces the mandatory Caesar cipher and transposition ciphers with a page apiece, but doesn't wait there at all. The rest of the section goes into fairly deep exposition of the construction (and deconstruction) of polyalphabetic ciphers like Enigma. The authors close with the introduction of Vernam ciphers and one-time pads. They go on to show how these actually demonstrate the ultimate vulnerabilities of any polyalphabetic cipher system, and set up the next chapter, symmetric key systems.

The symmetric chapter is probably the best one: I've never seen a good explanation of the internals of a block cipher until now. The authors lay out the data flow of DES on a single page and explain triple DES. They show how these are now vulnerable to brute force attacks and have been replaced by AES. While they don't detail the data flow of AES, they've left the reader with an understanding of how block ciphers, as a class, work.

The public-key section is just as complete. Mitani et al. explain in a matter of 75 pages the key exchange problem, trap-door functions, modulo arithmetic, Euler functions, and prime factoring. They finish the chapter by demonstrating a walkthrough of RSA: key generation, encryption, and decryption.

The closing section touches on the ways most people today use encryption even if they don't know it.

This book is one of a series of manga books from Omsha and No Starch. The series includes treatments of physics, relativity, statistics, and calculus. The originals are in Japanese, and No Starch is in the process of bringing them all to English-speaking audiences. Cryptography is the latest release.

The only problem with the translation is that in the opening stories, the authors use several jokes and puns that have to be called out and explained because they only make sense in Japanese. While these stood out, they only happen at the beginning of the book and none of them are critical to understanding.

While *The Manga Guide to Cryptography* is presented as a pop-media book, it would be a disappointment to someone looking for a casual pop treatment of cryptography. There's no code here, either. You won't learn how to use crypto libraries in your app. This book is best suited to the self-learner who wants to be conversant with the underlying ideas and perhaps understand some of what's going on behind-the-scenes every day when they use their web browser.

Millions, Billions, Zillions: Defending Yourself in a World of Too Many Numbers

Brian Kernighan
Princeton University Press, 2018, 174 pages
ISBN 978-0-691-18277-3

Reviewed by Rik Farrow

Kernighan begins his preface by quoting three geniuses, W. E. B. DuBois, John Nash, and Nate Silver, sharing their words about numbers. Of the three, Silver comes closest to the mark when I think about this book: "On average, people should be more skeptical when they see numbers." Kernighan wants us not just to be skeptical, but to use techniques, like estimation, to determine whether the numbers we see in print, from our friends, or in the news are actually correct.

Fortunately, Kernighan doesn't expect the readers of this book to be geniuses. Instead, he starts out by writing that all you need are grade-school arithmetic skills to follow along and learn from this work. I found that was true when I started reading portions of the book to my wife, who had been math-averse ever since some teacher criticized her for not being quick with numbers.

Throughout *Numbers*, my nickname for this book, Kernighan focuses on the use of estimation and rounding. In every example, he starts by using these techniques, allowing him to quickly come up with a decision about some numbers taken from the press: instead of using 330 million for the population of the US, try 300 million as a starting point, for example.

There are 13 chapters in *Numbers*, each focusing on a different aspect of accidental, or occasionally malicious, innumeracy. He covers mistakes involving names for large numbers or units, mis-mixing units, dimensionality, milestones, specious precision, statistics, bias, and arithmetic.

Kernighan begins by explaining how to make large numbers easier to understand. Words like million, billion, and trillion have no intuitive meaning to most people, myself included, and thus we tend to treat them as synonyms for "big," "really big," and "really really big." Kernighan suggests scaling down numbers: for example, if the US budget were $3.9 trillion, each person's share of that would be about $13,000.

He also suggests more use of scientific notation. This is one of the parts of the book I read to my wife as she painted, and she told me she finally understood scientific notation. Kernighan's explanations, and use of examples, really do make this book easy to understand.

Throughout *Numbers*, Kernighan provides shortcuts for quick calculations. Some were ones I already used, while others were new to me, like Little's Law, an easy method for figuring approximately how long it will take for an amount to double with different compound interest rates. Or the relationship of some powers of two to powers of ten that comes in very handy.

The only weak point in the book is the chapter on statistics. *Numbers* isn't a statistics book, and Kernighan does explain with a very clear example the common failing of the usage of the mean when the median would be more appropriate. He also makes many references to Darrell Hull's *How to Lie with Statistics* (1954), a book that he considers worth reading.

I liked reading *Numbers*. Kernighan's style could be said to be didactic, but it's never boring. I recommend *Numbers* to anyone who encounters potentially dubious numbers during their day—that is, everyone.

USENIX NOTES

Meet the Board: Kurt Andersen

Liz Markel, Community Engagement Manager

Kurt Andersen is one of two new board members elected in 2018 and was one of the first members of the USENIX community I met after joining the staff. He shared some details about his current professional activities, how he became involved with USENIX, as well as some interesting personal facts. Here are a few highlights:

Liz Markel: Tell me about your professional role and what kinds of problems you're working on solving now in that role.

Kurt Andersen: I'm currently a Senior Staff Site Reliability Engineer at LinkedIn working in the Product-SRE arm of our SRE organization (one arm of four—the others are data, infrastructure, and security). Currently, my primary focus is on enhancing the continuing education environment across the entire SRE and Security organizations. I've also led the implementation of our annual organization-wide internal conference for the last three years, and I work widely in reviewing projects and initiatives for reliability and scaling concerns.

I also work with the Messaging, Malware, Mobile Anti-Abuse Working Group (m3aawg.org) on their board and as a program co-chair, and I am active with the Internet Engineering Task Force (IETF) in the development and refinement of various standards related to messaging, security, and privacy.

LM: How were you first introduced to USENIX?

KA: I had occasionally encountered USENIX earlier in my career mainly through its role in publishing interesting papers in areas that related to professional concerns in the moment. With my interest and concern around security and privacy, I think the first time I attended a USENIX-associated event was SOUPS (Symposium on Usable Privacy and Security), which is now co-located with the USENIX Security Symposium. I missed the first SREcon (2014—Santa Clara) but have been an avid participant ever since.

LM: Why did you decide to pursue a seat on the board?

KA: I think that there is a huge potential opportunity for USENIX to strengthen the academic/industry connections betwixt our participating communities to the advancement of both. On the academic side, there is the potential for more relevant, interesting problems by interacting with professional practitioners. There is also the obvious benefit to students in having both experience and connections with industry.

For the professionals, we can benefit by the unique time and resources which academic researchers can bring to bear on problems, and we can also contribute toward the successful "heritage" by having students who are better prepared to move from the academy to industry.

USENIX has had a long tradition of inclusiveness and diversity, and I'm delighted to contribute to helping overcome the systemic issues that have limited the diversity in the computing field.

LM: Why should someone consider becoming involved in USENIX?

KA: I think that it goes very much to one of LinkedIn's core values and one in which I have found increasing truth and depth over time: relationships matter. As a participant in any role, you will have opportunities to interact with people who have both common concerns and divergent approaches to address those concerns. The more you can learn, the better you will be as a practitioner.

LM: Do you have one unique fact about yourself you can share with us?

KA: I've been an amateur radio (ham) operator since 5th grade.

LM: What's your favorite board game?

KA: Any one that I can win; so my family sticks to games for fun only.

LM: Tell me a bit about the region of the country you live in.

Kurt Andersen co-chaired the SREcon18 Americas conference with Betsy Beyer (Google).

KA: We live in a rural part of Northern Idaho, within 100 miles of the Canadian border. Being on the western slope of the Rockies, we avoid the arctic blast of Canadian air that runs down into the Midwest, but we have a wonderful four-seasons climate and a Fall color show in the trees that is the equal of anything I've seen in the Northeast. The nearest towns of Coeur d'Alene and Sandpoint are wonderful tourist destinations, with tons of outdoor recreation activities from golf to hunting and fishing or skiing, hiking, and mountain biking; but too many people are moving here, and the infrastructure is experiencing significant congestion problems.

LM: Anything else you'd like to share?

KA: I'd say that I am an amateur (in the original Latin sense—one who loves) design aficionado, ranging from Donald Norman's *The Design of Everyday Things* to Edward Tufte's *The Visual Display of Quantitative Information*. Finding ways to communicate effectively and simply in spite of complexity is the pervasive theme. I also appreciate good wine, whisk[e]y, and espresso.

There's more to read! Visit the USENIX Blog at www.usenix.org/blog for Kurt's full-length interview.

USENIX Member Benefits

For information regarding membership or benefits, please see www.usenix.org/membership/, or contact us via email at membership@usenix.org, or telephone +1 510.528.8649.

USENIX ASSOCIATION FINANCIAL STATEMENTS FOR 2017

The following information is provided as the annual report of the USENIX Association's finances. The accompanying statements have been prepared by BHLF LLP, CPAs, in accordance with Statements on Standards for Accounting and Review Services issued by the American Institute of Certified Public Accountants. The 2017 financial statements were also audited by BHLF LLP. Accompanying the statements are charts that illustrate the breakdown of the following: operating expenses, program expenses, and general and administrative expenses. The Association's operating expenses consist of its program, management and general, and fundraising expenses, as illustrated in Chart 1.

These operating expenses include the general and administrative expenses allocated across all of the Association's activities. Chart 2 shows USENIX's program expenses, a subset of its operating expenses. The individual portions shown represent expenses for conferences and workshops; membership (including *;login:* magazine); and project, program, and good works. Chart 3 shows the details of what makes up USENIX's general, administrative, and management expenses. The Association's complete financial statements for the fiscal year ended December 31, 2017, are available on request.

Casey Henderson, Executive Director

USENIX ASSOCIATION

Statements of Financial Position
December 31, 2017 and 2016

	2017	2016
ASSETS		
Current assets		
Cash and equivalents	$ 641,026	$ 742,910
Accounts receivable	118,733	94,535
Prepaid expenses	219,894	215,002
Investments	6,365,034	5,803,274
Total current assets	7,344,687	6,855,721
Property and equipment, net	85,137	137,795
Total assets	$ 7,429,824	$ 6,993,516
LIABILITIES AND NET ASSETS		
Current liabilities		
Accounts payable and accrued expenses	$ 49,859	$ 744,335
Accrued compensation	72,363	59,811
Deferred revenue	739,440	443,275
Total current liabilities	861,662	1,247,421
Deferred revenue, net of current portion	187,500	337,500
Total liabilities	1,049,162	1,584,921
Net assets		
Undrestricted net assets	6,380,662	5,408,595
Total net assets	6,380,662	5,408,595
Total liabilities and net assets	$ 7,429,824	$ 6,993,516

USENIX ASSOCIATION

Statements of Activities
Years Ended December 31, 2017 and 2016

	2017	2016
REVENUES		
Conference and workshop revenue	$ 5,064,417	$ 4,857,484
Membership dues	228,793	257,295
General sponsorship	129,000	60,000
Product sales	5,146	7,067
Event services and projects	4,000	4,750
LISA SIG dues and other	-	7,946
Total revenues	5,431,356	5,194,542
EXPENSES		
Program services		
Conferences and workshops	4,554,459	4,537,398
Projects, programs and membership	307,626	377,969
Total program services	4,862,085	4,915,367
Management and general	445,114	621,006
Fundraising	50,849	84,715
Total expenses	5,358,048	5,621,088
CHANGE IN NET ASSETS FROM OPERATIONS	73,308	(426,546)
OTHER INCOME (EXPENSES)		
Donations	24,470	23,660
Investment income	925,730	432,343
Investment fees	(51,441)	(45,897)
Other income	-	150
Total other income (expenses)	898,759	410,256
Change in net assets	972,067	(16,290)
NET ASSETS - unrestricted		
Beginning of year	5,408,595	5,424,885
End of year	$ 6,380,662	$ 5,408,595

USENIX ASSOCIATION FINANCIAL STATEMENTS FOR 2017

Chart 1: USENIX 2017 Operating Expenses

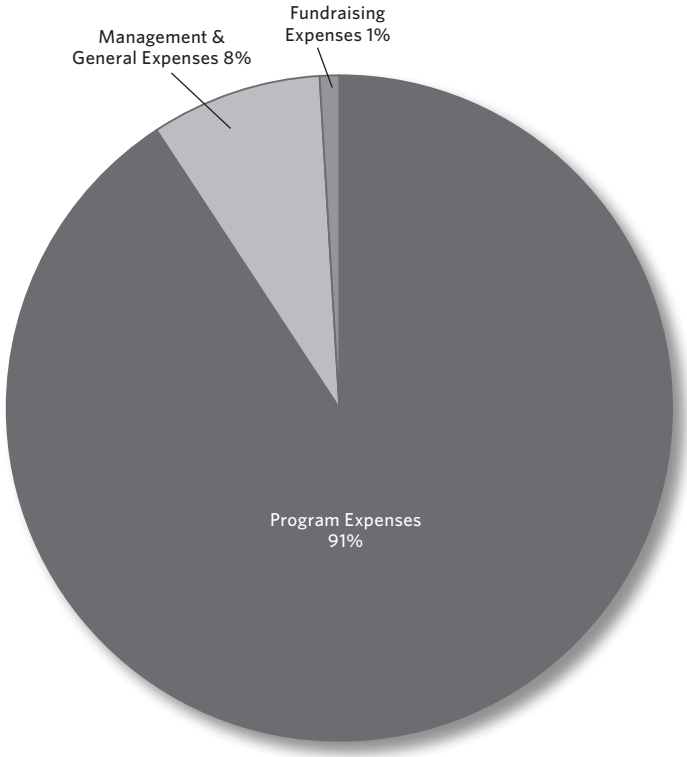


Chart 2: USENIX 2017 Program Expenses

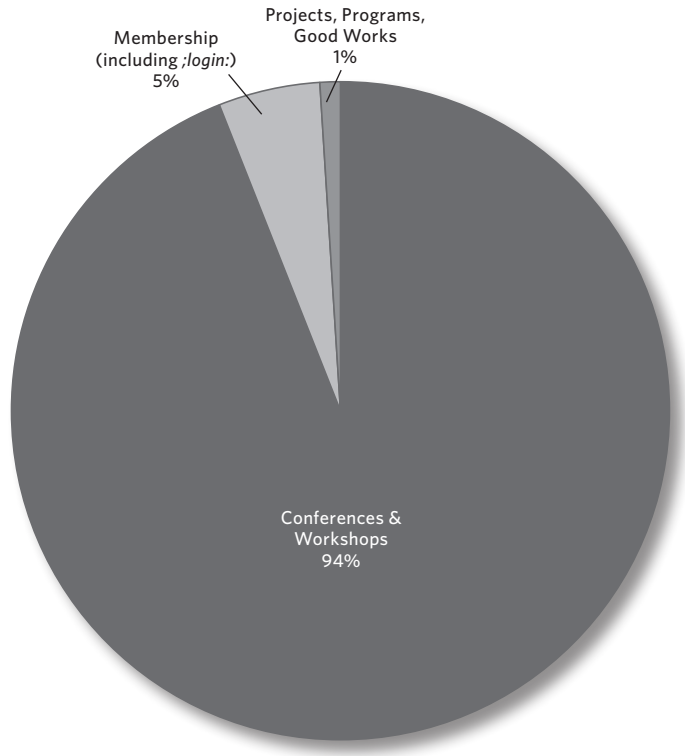
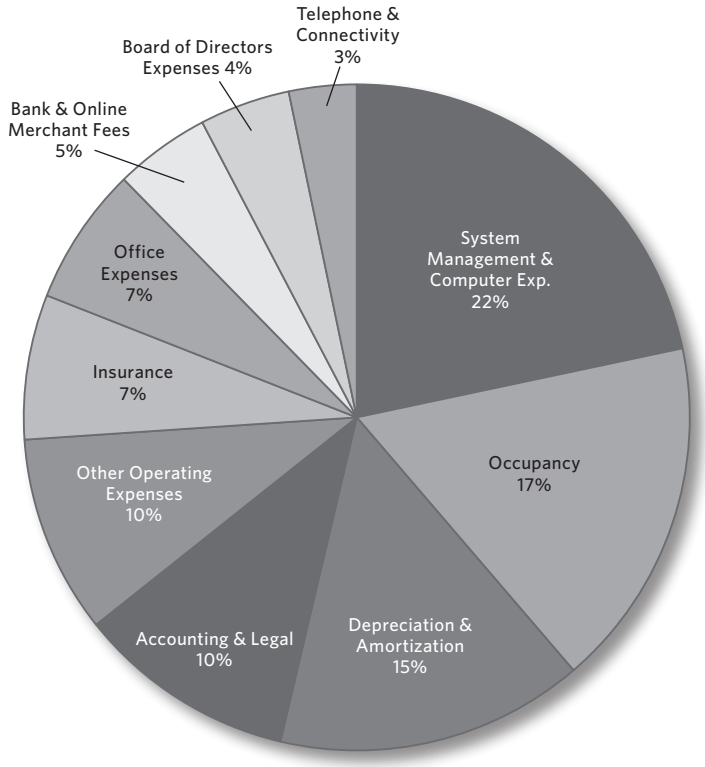


Chart 3: USENIX 2017 General & Administrative Expenses



Join us in Boston!

FAST'19

17th USENIX Conference on File and Storage Technologies

February 25–28, 2019
www.usenix.org/fast19

FAST brings together storage-system researchers and practitioners to explore new directions in the design, implementation, evaluation, and deployment of storage systems.

VAULT'19

2019 Linux Storage and Filesystems Conference

February 25–26, 2019
www.usenix.org/vault19

After a one-year hiatus, Vault returns in 2019, under the sponsorship and organization of the USENIX Association, and will bring together practitioners, implementers, users, and researchers working on storage in open source and related projects.

nsdi'19

16th USENIX Symposium on Networked Systems Design and Implementation

February 26–28, 2019
www.usenix.org/nsdi19

NSDI focuses on the design principles, implementation, and practical evaluation of networked and distributed systems. Our goal is to bring together researchers from across the networking and systems community to foster a broad approach to addressing overlapping research challenges.

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

ENIGMA

A USENIX CONFERENCE

Security and Privacy Ideas that Matter

FEATURED SPEAKERS


Denelle Dixon
Mozilla


Ashkan Soltani
Independent Researcher
and Consultant


Emily Stark
Google


Joe Kiniry
Galois and Free & Fair


Daniela Oliveira
University of Florida


Bob Lord
Democratic National
Committee

The full program and registration are available now.

enigma.usenix.org

JAN 28–30, 2019

BURLINGAME, CA, USA

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION