;login:
WINTER 2019 VOL. 44, NO. 4

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

UPCOMING EVENTS

Enigma 2020
January 27–29, 2020, San Francisco, CA, USA
www.usenix.org/enigma2020

FAST '20: 18th USENIX Conference on File and Storage Technologies
February 24–27, 2020, Santa Clara, CA, USA
Sponsored by USENIX in cooperation with ACM SIGOPS
Co-located with NSDI '20
www.usenix.org/fast20

Vault '20: 2020 Linux Storage and Filesystems Conference
Feburary 24–25, 2020, Santa Clara, CA, USA
Co-located with FAST '20
www.usenix.org/vault20

NSDI '20: 17th USENIX Symposium on Networked Systems Design and Implementation
February 25–27, 2020, Santa Clara, CA, USA
Sponsored by USENIX in cooperation with ACM SIGCOMM and ACM SIGOPS
Co-located with FAST '20
www.usenix.org/nsdi20

SREcon20 Americas West
March 24–26, 2020, Santa Clara, CA, USA
www.usenix.org/srecon20americaswest

HotEdge '20: 3rd USENIX Workshop on Hot Topics in Edge Computing
April 30, 2020, Santa Clara, CA, USA
Paper submissions due February 20, 2020
www.usenix.org/hotedge20

OpML '20: 2020 USENIX Conference on Operational Machine Learning
May 1, 2020, Santa Clara, CA, USA

SREcon20 Asia/Pacific
June 15–17, 2020, Sydney, Australia

2020 USENIX Annual Technical Conference
July 15–17, 2020, Boston, MA, USA
Paper submissions due January 15, 2020
www.usenix.org/atc20

SOUPS 2020: Sixteenth Symposium on Usable Privacy and Security
August 9–11, 2020, Boston, MA, USA
Co-located with USENIX Security '20

29th USENIX Security Symposium
August 12–14, 2020, Boston, MA, USA
Winter Quarter paper submissions due February 15, 2020
www.usenix.org/sec20

SREcon20 Europe/Middle East/Africa
October 27–29, 2020, Amsterdam, Netherlands

OSDI '20: 14th USENIX Symposium on Operating Systems Design and Implementation
November 4–6, 2020, Banff, Alberta, Canada
Sponsored by USENIX in cooperation with ACM SIGOPS
Abstract registrations due May 5, 2020
www.usenix.org/osdi20

LISA20
December 7–9, 2020, Boston, MA, USA

SREcon20 Americas East
December 7–9, 2020, Boston, MA, 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. Please help us support open access by becoming a USENIX member and asking your colleagues to do the same!

www.usenix.org/membership

;login:
WINTER 2019 VOL. 44, NO. 4

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
Ann Heron
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.

Musings

RIK FARROW

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

While I was attending my very first FAST conference, in 2006, a journalist told me that he had a question for a file system expert. I walked him toward registration and quickly spotted someone who I thought could answer his question. After all, I didn't think it would be that hard.

"Why are there so many file systems?" he asked. I was floored. At the time, I thought this was a naive question and felt embarrassed for the journalist. But I later realized I shouldn't have been, because while I had accepted the fact that there were many file systems, multiple ones for every popular operating system, there actually are many reasons why there are so many.

On the surface, you might think that each group of operating systems designers wants to write their very own file system. And you'd be right, up to a point. If you look at the Wikipedia page that compares file systems [1], all of the most commonly used file systems share features. But this misses two important points.

First, the underlying operating systems cannot support dropping in existing file systems. The interfaces to key support functions—for example, allocation of memory—will be different. These differences go even deeper than interfaces. Windows NT, now known as Windows, has a very different design philosophy from any UNIX-related operating system. One manifestation of this is that the NTFS keeps file and directory names, attributes, and block lists in the Master File Table whenever possible. This allows for much faster filename searches and file openings but at the price of having a single point-of-failure—yes, the MFT is a file. I'm exaggerating slightly, as there is a copy of the MFT, and the locations of each are stored in the boot block. UNIX-style systems have superblocks for file-system metadata, inodes for the attributes and blocks in files, and directories themselves are files. I'm skipping over how large files and extended attributes are handled for simplicity, but UNIX and NT file systems handle key features very differently.

The second reason for the variety found in file systems is that the science behind building fast and reliable file systems advances. Fragmentation was a huge problem for the Version 7 UNIX file system, which continued to be used in System V. The Fast File System solved this through the use of cylinder groups, something used in Linux ext and the NTFS as well. FFS also introduced the notion of having multiple copies of the superblock. But ext2 experimented with a dangerous technique that has turned out to work: it dispensed with synchronous writes of metadata, which block any further writing to the file system. Unpacking a tar archive on the same hardware was 10 times faster using ext2 than FFS. I know, because I experimented with both BSDI and Linux in 1992.

Finally, file systems exist to support applications, and not all applications require the same style of operations. Some applications create large sequential reads or writes, some create lots of small files (mail systems), high performance computing systems create thousands of checkpoint files nearly simultaneously, and so on. So while ext4 works fine for most Linux installs, some applications demand the use of XFS for handling large files.

The NSF Visioning Workshop [2], with a report published in 2019, focuses on the potential future design requirements for storage and file systems. Some things in the report just seem like common sense: for example, creating support for commonly used lower level file system tasks that can be reused. Other facets of this report cover newer applications, such as edge computing and machine learning, that have different needs than older applications.

The five lead authors of the workshop report (you can find a list of the workshop participants at [2]), wrote about some of the report's findings for this issue of *;login:*.

The Lineup

Given that build-up, you might expect that this issue will be all about file systems and storage, but you'd be wrong. We actually start out with several articles related to security.

Jean Camp writes about her experience monitoring BGP mistakes and attacks. BGP was designed when there was little concern for security, and Internet operators were happy enough to have a routing advertisement protocol that just worked. There have been many attempts to secure routing updates that have not succeeded in the decades since BGP was adopted. Camp explains how important it is to consider the geopolitical aspects of running BGP safely, and describes a tool that permits blocking access to networks affected by mistaken or malicious route updates.

Reardon et al. share their work on finding side and covert channels in Android. They built upon their prior work in improving the usability of the Android permission system and creating a monitoring tool to check how well the permissions systems worked. While testing the accuracy of their tools, they uncovered lots of malfeasance, including in libraries specifically for apps for children, a violation of the law in many countries.

Kulandaivel et al. talk about the CAN bus scanner they developed, called CANvas. While you can purchase information about replacing your car's Electronic Control Units, that tells you only about the ECUs the manufacturer installed in your car, not anything added later. The authors explain why a scanner is useful but also difficult to implement for the CAN bus.

Peter Peterson and Rob Jansen have produced a summary of the CSET '19 workshop. CSET had a broader focus this year, and the chairs describe how this new focus and a larger program committee worked out. They also include summaries of all the presentations from CSET '19.

Kirill Levchenko is interested in aviation, but his interest goes deeper than just being a passenger. Levchenko has been working with a multi-institution group to create testbeds for the communication buses used in large passenger aircraft, like the Boeing 737. Some of what they have discovered turns out to be comforting rather than frightening, as I discovered while interviewing him.

Amvrosiadis et al. share some of the results of the NSF-sponsored future of file systems and storage 2025 workshop mentioned in the opening. The idea behind the workshop was to provide directions for future research and development in these areas. In particular, the workshop participants determined important new areas that have, or are expected to have, unusual storage requirements, such as machine learning and the Internet of Things.

Terence Kelly has been designing programming paradigms for persistent memory for many years. In this article, Kelly demonstrates two different techniques, one a programming style for traditional storage-backed memory, and the second, a mechanism for making changes to the backing-store atomic.

Effie Mouzeli shares her perspective on "Ask-Me-Anything" engineering. Not all system engineers (SEs) work at huge companies. Mouzeli provides useful advice for people working alone or in small teams of SREs in the often chaotic environments of startups and at smaller organizations, where the SE must be able to solve almost any problem—thus the AMA designation. Mouzeli writes from her own life experiences, including about the five stages of technical debt, with humor and honesty.

Amit Gud explains the benefits of multi-tenancy in microservice environments. Multi-tenancy means that data in flight and when stored include labels that are used to control the flow and usage of data in these environments. Uses include testing code or configuration changes and designing more modularity into systems.

Laura Nolan writes about the pitfalls of dynamic control systems. Ever wonder why the servers you once had in your racks were more reliable than the complex systems run by gigantic cloud services? Wonder no more.

Peter Norton wants to teach you how to add useful profiling to your Python scripts with the goal of looking at different visualizations. The default Python profiler doesn't produce as useful results as newer tools, so Norton demonstrates other tooling.

Dave Josephsen tells us about distributed tracing. There have been two popular projects, OpenTracing and OpenCensus, which are being merged into one. And the IETF has been working on a way to use HTTP headers to do this called OpenTelemetry. Dave explains the differences between these approaches.

Dan Geer and Jason Crabtree challenge us to get clear about our security metrics. If everyone creates and uses their own set of in-house metrics, we cannot share measurable information about attacks, risk, and the success or failure of defenses.

Robert Ferrell muses about possible solutions to the wave of ransomware affecting systems throughout the world. I pointed out several obvious weaknesses in most of his approaches, and he reproached me. It *is* a humor column, Robert reminded me.

Mark Lamourine has reviewed cookbooks about Kubernetes, Ansible, and OpenStack in this issue, and I cover Randall Munroe's cookbook for how to solve common problems using absurd scientific advice.

While considering Robert Ferrell's absurd solutions to ransomware, I wondered why victims almost never have good backups prepared. We all know about the importance of backing up systems and testing these systems before we need them. Could it be that most IT departments cannot handle this most basic of tasks, one that comes right after user management?

The failure of so many organizations tells us volumes about the world we live in. The real world is prone to failure and is not full of people eager to do the repetitive work of having to create routine backups—even though the occasional consequences are far worse than the boredom entailed while spot-checking backups for correct functionality—or the more difficult task of setting up and enforcing a site-wise backup system or policy.

Sometimes I think that I live in a world populated by teenagers, all in revolt against everything that has stood the test of time and willing to risk everything just because they can. I was glad when my teenagers outgrew that period of their lives, and I can't wait for the rest of the world to get there, too.

References

[1] Comparison of File Systems: https://en.wikipedia.org/wiki/Comparison_of_file_systems.

[2] G. Amvrosiadis, A. R. Butt, V. Tarasov, E. Zadok, and M. Zhao, Data Storage Research Vision 2025 Report on NSF Visioning Workshop, 2018: https://dl.acm.org/citation.cfm?id=3316807.

Letter to the Editor

Thanks for the enthusiastic intro to our "Not So Fast" work in the Musings editorial in the Fall 2019 issue.

We did want to highlight a few unfortunate misconceptions about Browsix-Wasm—the biggest being that Browsix is most emphatically not a browser plugin, and that it does not provide access to the host's file system!

Browsix works as a JavaScript/Wasm library that runs entirely within the browser, without the need to install plugins. The file system that Browsix exposes to programs (like SPEC) is entirely independent of the host OS's file system. For SPEC, all the files come from an HTTP server, and a writeable "overlay" file system provides ephemeral storage for the duration that a tab is alive (a very similar approach to how Docker provides layered file systems).

Like the file system, all of the operating system services that Browsix-Wasm provides (like processes and pipes) are built on top of standard browser APIs (like WebWorkers) within the confines of the browser sandbox. This approach is what enables us to work across all major browsers.

Given that safety, you and everyone else should feel just fine about running Browsix-Wasm on your daily driver browser, and indeed you might visit websites that use Browsix without you even knowing it! (Components of it help power the emulators on archive.org, like the Oregon Trail: https://archive.org/details/msdos_Oregon_Trail_The_1990.)

Many thanks in advance,

—Abhinav, Bobby, Emery, and Arjun

Correction

In the book review of *Concurrency in Go* (Summer 2019 issue, page 59), C. A. R. Hoare was incorrectly listed as C. Anthony and R. Hoare in the reference at the end of the review. We apologize for the error.

Save the Dates!

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

FAST '20 — 18th USENIX Conference on File and Storage Technologies

February 24–27, 2020 | Santa Clara, CA, USA

Sponsored by USENIX in cooperation with ACM SIGOPS
Co-located with NSDI '20
www.usenix.org/fast20

The 18th USENIX Conference on File and Storage Technologies (FAST '20) brings together storage-system researchers and practitioners to explore new directions in the design, implementation, evaluation, and deployment of storage systems.

The program committee will interpret "storage systems" broadly; papers on low-level storage devices, distributed storage systems, and information management are all of interest. The conference will consist of technical presentations including refereed papers, Work-in-Progress (WiP) reports, poster sessions, and tutorials.

---

nsdi '20 — 17th USENIX Symposium on Networked Systems Design and Implementation

February 25–27, 2020 | Santa Clara, CA, USA

Sponsored by USENIX in cooperation with ACM SIGCOMM and ACM SIGOPS
Co-located with FAST '20
www.usenix.org/nsdi20

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.

NSDI provides a high-quality, single-track forum for presenting results and discussing ideas that further the knowledge and understanding of the networked systems community as a whole, continue a significant research dialog, or push the architectural boundaries of network services.

The full programs and registration will be available in December.

Using ML to Block BGP Hijacking

L. JEAN CAMP

L. Jean Camp is a Professor in the Luddy School of Informatics, Computing, and Engineering at Indiana University. She is currently a visiting scholar is at the University of California at Berkeley's Center for Long-Term Cybersecurity. She joined Indiana after eight years at Harvard's Kennedy School where her courses were also listed in Harvard Law, Harvard Business, and the Engineering Systems Division of MIT. After earning her doctorate from Carnegie Mellon, she spent one year as a senior member of the Technical Staff at Sandia National Laboratories. She began her career as an engineer at Catawba Nuclear Station with an MSEE from the University of North Carolina at Charlotte. She has authored more than 150 peer-reviewed publications on security and privacy, addressing trust on every layer of the OSI model. ljcamp@indiana.edu

Border Gateway Protocol (BGP) has proven to be resilient in the face of failures, attacks, and general maliciousness and incompetence. While there are no deployed mechanisms for automatically remediating BGP announcements that may be malicious, there have been many attempts at fixing this sorry state of affairs. In this article, I will describe some troublesome BGP events and how our tool, Bongo, uses machine learning (ML) and Layer 8 in the IP stack to detect malicious announcements and block traffic that would be diverted.

David Clark, in his book *Designing an Internet*, identified the Internet as a socio-technical system that could have been developed in many different ways. That the Internet is social, political, and economic is not contested in 2019. Yet the argument that BGP is social, political, and economic pushes this discussion to another level. In recent years several trends have emerged illustrating the vulnerability of the Internet's control plane. One of these trends is that, as the Internet expands, the expertise of individual operators is increasingly diverse, leading to more routing errors. The second is the increase in traffic redirection: in other words, route hijacking as an attack vector.

Consider first, misguided network configurations. China Telecom announced 15% of all IPv4 space in April of 2010, resulting in tremendous loss of traffic. This was represented as an error, and given that the traffic did not reach its intended destination, this would have been an extremely clumsy attack.

Another failure of routing was the misconfiguration of a small Australian ISP in 2012, one which took the continent "down under" down for hours. This is a most common failure of routing: a straightforward failure of human factors in configuration. These outages were immediately apparent and repaired within hours. However, the smaller customer ISP did in fact announce all routes from the two larger ISPs. The upstream ISP did not filter the route announcements appropriately and can be said to have both caused and suffered from the outage. Route leaks are a function of lack of technical competence, but this does not mean that they cannot diffuse quickly and cause harm. Previously proposed solutions have included customer route filtering, such as filtering all traffic from a client that will affect remote prefixes. The simultaneous emergence of untrustworthy behavior and the continuing need for connectivity illustrates the risk of such an approach. A different approach requires the increase of technical competence among not only ISPs but also larger end users, which appears optimistic [8].

In addition to errors and odd incidents, there has also been a wide range of political attacks. An early example originated from Pakistan in 2008, where Pakistan identified several YouTube videos as sufficiently problematic politically that the decision was made to block all of YouTube. An address internal to Pakistan for YouTube was announced within Pakistan Telecom; however, it was also broadcast across northern Africa and Europe, with a duration of hours. Arguably, while the intention to block YouTube within Pakistan was political, the leak itself was a human factors problem.

The efficacy of attacks of nation-states on the reachability of the Internet was further proven during Arab Spring, as rapid drop-offs for Egyptian and Libyan populations were easily observable but less easily repairable.

When an entity misrepresents its location in a routing path, rather than claiming to own a destination, the errors are less obvious. With this attack, traffic continues to be delivered and such a routing configuration could remain stable for long periods. Such a case occurred between China Telecom and AT&T, this time for some period of months: Facebook traffic was routed through China. Note that while the login to Facebook is protected by TLS, other uses of Facebook were not encrypted at that time. Thus a significant amount of global traffic was routed through a nation where Facebook adoption is remarkably low.

Since 2001, route hijacking has been turned into an attack [4]. Many of these attacks appear to remain undetected and unreported, creating a call for a ubiquitous cryptographic solution, RPKI [9]. Like many previous solutions, RPKI is incentive-misaligned and requires widespread adoption. BGPSEC is another example of a solution that was operationally, and economically, misaligned to the problem it was intended to solve. BGPSEC makes both requirements and benefits for early adopters that discourage innovators. Should BGPSEC be adopted it may not be resistant to political attacks nor misconfigurations. As the experience with certificate authorities issuing X509 certificates for the Web has demonstrated, political attacks are difficult to prevent and detect with an all-or-nothing cryptographic model of trust.

Sometimes malicious authorities are generally trusted for purposes of access, interoperability, and connectivity when populations on the network experience these authorities as malicious. Other solutions call for the creation of trusted third parties [10] or other changes in the infrastructure.

Addressing Geopolitical Dimensions

Our research directly addresses the geographical and political dimensions of BGP, reaching beyond purely technical dimensions to develop operational solutions. While specific threats to the control plane have included political interference, misguided network configurations, and miscellaneous mischief, much research on hardening the control plane has tended to be carefully neutral and apolitical—which is itself an ideological choice.

We contest this viewpoint by using a variety of data, including technical, rates of change, economic, and geopolitical, as network topology changes via BGP updates can offer probabilistic (not cryptographic) trust indicators. We understand that any ML approach needs to consider that attackers can and do adjust their strategies when defenses appear, and that any mechanism needs to do more than just provide temporary benefits.

At our lab, the application of machine learning to security has three basic principles. First, do not examine something that is easy for the attacker to change. Second, focus on the minimal requirements for a successful attack. Then leverage those minimal requirements to identify features that the attacker will have difficulty altering in a successful attack. And finally, use offline information or indicators of offline information when possible to better identify those features.

Using this as a starting point, we identified the fact that a requirement for BGP routing attacks is the ability to manipulate announcements from a single autonomous system (AS). Attacks, foolishness, and manipulations have arisen from single source ASes rather than coordinating across source networks. That the geography of the network reflects political geography was an underlying assumption for our work, one that has been more recently explored and validated [7].

What are the minimal requirements for a successful attack? The attacker's routing announcements must be distributed across the Internet without arousing suspicion so that the traffic is misdirected. Detecting this behavior means identifying patterns in attacks, so we began with offline features: jurisdiction and the data arising from a nation-state approach.

Incompetence or Attack

We did wonder whether we were simply seeing the rise of less qualified operators. An IETF member (who asked to remain anonymous) once argued, "As the number of people on the internet increases, the amount of cluefulness remains constant," meaning that more incidents will occur as participation in the network increases. Could cluelessness explain BGP anomalies? Is lack of technical expertise a significant explanatory variable for why some countries initiate more BGP incidences than others? Any given anomaly could be an accident, a crime, or an attack.

To understand the nature of routing anomalies, we empirically investigated the nations of origin of the events, using multiple regression and unsupervised learning techniques to analyze anomalies over a four-year period. If BGP anomalies are a result of limited technical competence, then countries with low levels of education, few technology exports, and less expertise should be overrepresented. If BGP anomalies are crime, leveraged by criminals for profit, then economic theories and analytical approaches from criminology should show statistical significance. Using macroeconomics by leveraging three theories from criminology and global measures of technology adoption, we examined whether anomalies were likely incompetence, potential e-crime, or intelligence operations.

For the issue of technical competence we included secure Internet services, IT exports, and third-party measures of network readiness. Our variable selection was also informed by different

theories of criminology. We found that exports of technology were not statistically significant, undermining the argument for incompetence. We also found support for the possibility that anomalies were driven by crime, specifically for the guardianship and relative deprivation theories of crime.

We used unsupervised learning on the sources of BGP hijacks and apparently malicious routing announcements; and the result was two categories of nations, one correlated with higher levels of corruption and one associated with conflict. This clustering indicated that civil conflict and surveillance were associated with the disproportionate origination of routing anomalies [5].

Choosing Safety over Availability

The lack of support for the hypothesis that BGP anomalies are generated by incompetents, the support for the role of crime, and the indication that political stability is an indicator provided a design guideline for our next step. If there is a threat from a particular jurisdiction, then announcements that change stable routing patterns to include these jurisdictions are suspicious.

Beyond the refusal to treat the Internet as if it were some third space, unmoored to political nor geography reality, our next decision made explicit the tradeoff between availability and confidentiality. Is there data where it is better not to send it, rather than sending it and risking exposure? We developed a proof of concept that generates firewall rules based on BGP changes according to the jurisdiction of the nation of the announcing AS.

We examined the feasibility of changing the calculus of the confidentiality, integrity, and availability model by providing the choice to select confidentiality and integrity over availability [3]. We named this package Bongo to fit with the ungulate method of naming in the open source route reflector tradition. We distinguish this from Quagga and Zebra in that bongos are the only spiral-horned antelopes (tragelaphid) where both male and females have horns. Our goal was to offer a pointed defense. Bongo uses the Routing Information Base (RIB) to identify changes not in routes but in AS, and then assigns each AS a risk parameter. Based on this parameter, Bongo can create a firewall rule. Alternatively, Bongo may simply issue an alert to the operator, if the change is worthy of note but not high risk.

The approaches for route filtering also would work with the next generation Internet architecture, software-defined networking (SDN). SDN is an established alternative architecture, where flows (rather than routes) define the paths of data using information from multiple layers (rather than IP and NAT information only). To say that SDN was not built with security as a design goal is a bit of an understatement. Yet the nature of flows inherently offers potential against BGP attacks. Changes in flows and delaying flows should be less prone to overlap and confusion than potentially quickly changing firewall rules. If this holds under further study, then SDN allows the promise of rejecting changes in the BGP topology.

Bongo also implements the creation of OpenFlow rules based on risk estimates, including jurisdiction. The key difference between Bongo and other works is the assumption that hijacks and leaks will happen upstream beyond the control of the end of the network. The goal is to enable what a BGP end user—that is, companies operating networks receiving BGP updates from their upstream ISPs—can do to react to these events.

Additionally, even if prefix hijacking and path-length attacks were to be completely eliminated, an organization may want to cease sending data in reaction to a topology change that would send sensitive data to an undesirable geographical region. Bongo enables prohibition of traffic from traversing certain jurisdictions. At some level, this seems as simple as the promulgation of simple yes or no rules. However, full path verification is clearly not feasible (upstream routers may lie about AS paths), so this detection inherently includes uncertainty.

A Pragmatic Approach

Bongo is designed to defend the network as it is currently operated, without assumptions about adoption of PKI or any other technology in the future. Bongo approaches route updates as risk decisions and can estimate the risk of adopting a route based on any filter the programmer deems appropriate. Bongo allows an organization to decide exactly how much and what kind of risk it will accept from the control plane.

In addition to publishing this in a traditional technical domain—an ACM workshop—we also sought feedback from experts in communications and Internet policy. An earlier version, before the ACM publication, was presented at the Telecommunications Policy Research Conference. This venue has a rich history of publishing visions before there is a reality. Software-defined radio was included in the call for papers before it was a respectable technical idea; and Internet commerce was a session at TPRC before the First Workshop on Electronic Commerce was hosted by one of the big three (ACM, IEEE, and, of course, USENIX) [1].

The feedback at that venue was that the reality of the politics of delay, misdirection, and black holing of traffic was already known in policy circles. The roles of criminals and intelligence agencies was much discussed at this Telecommunications Policy Research Conference, but rarely with engineers in the room.

To determine how disruptive such an approach might be, we examined the paths to the top global financial institutions. We found only the Bank of Tehran would have suffered any denial of service connecting from California while using Bongo, and that may be a result of route flapping. Other routes were extremely consistent in terms of jurisdictions traversed. We identify these as political and geographical decisions [2].

Specifically, we selected the top 50 banking websites from Alexa as a rough approximation for sensitive addresses to which an organization may want to apply transit restrictions. We examined the BGP state from a single ISP based in San Francisco over the course of April 2016 and identified every time the country associations changed in the AS path to each banking site. Out of the 50 financial services IP addresses analyzed over the one-month period, only four of them experienced country changes in the path to their destination and one experienced an outage. That means that for 46 of the IP addresses, an extremely strict exfiltration policy, restricting to no AS path country changes, would not have caused any outages during this time period from our vantage point in San Francisco.

Only four showed a change in jurisdiction along their paths. Online.citibank.com, during a period of 24 hours on April 5 to April 6, 2016, had routes to their network completely withdrawn from the routing table. The path for www.nbg.gr, the website for the National Bank of Greece, changed on April 9 to a direct peering between a US provider and a Greek (GR) provider, eliminating an intermediary peer in the EU. HSBC.com alternated between being advertised directly in the US and being advertised by an ISP in Great Britain (GB). For US-only paths, this would have caused an eight-day outage and then a three-day outage, but for US and GB paths, there would have been no impact.

The one likely impacted domain would have been Bsi.ir, Bank Saderat Iran, headquartered in Tehran. From the start of the period until April 17, the path traversed the US, Russia (RU), Azerbaijan (AZ), and Iran (IR). For the following eight days, it traversed Oman (OM), the US, and Iran. Then for four hours on April 25, the path traversed Germany (DE), the US, and Iran, after which it switched back to the Oman route. Thus, the use of Bongo for preventing connections through specified jurisdictions would have affected only Bank Saderat in its connections to the US were Germany, Russia, or Azerbaijan disallowed.

Another Approach

Given that it is feasible to defend against possible targeted events, can we detect larger-scale assaults on BGP? Again we began by asking the three core questions for applying machine learning to security: what is required, what is the goal, and what features can be identified with this understanding? Building on the requirements to defend against attacks that require large-scale redirections, we defined the goal of the attack as fast, undetected changes in topology.

Our analysis showed that large-scale attacks could be identified at an earlier stage than reported in the literature, again by focusing on the single-AS source and the necessary results for a successful attack. For a successful attack, attackers must issue a large number of route changes, leading to detectable bursts in their announcements. So we examined each AS as a data source and compared the interarrival times of announcements not only from the (malicious) AS but also to the neighboring ASes. BGP announcements that are associated with disruptive updates occurred in groups of relatively high frequency, often multiple standard deviations from normal rates, followed by periods of infrequent activity.

Conclusion

Together this set of analysis and open code illustrates that it is possible to quickly identify some BGP attacks and then mitigate the risks of hijacks when confidentiality is more valuable than availability [6].

Since a core problem with BGP resiliency is the concept of trust, then trust and risk must be a core of the solution. Understanding routing updates as a function of trust and risk enables approaching such updates as partially trusted. Cryptographic solutions attempt to provide perfectly trustworthy sources and paths. Yet certificate authority subversion in the TLS realm have shown that today's certificates are not themselves trustworthy; nor does this proposed solution address misconfiguration or malicious configurations.

Trust is not globally encompassing. The Internet is no less affected by the transit across geographical boundaries than were telephone calls, letters, or other communications infrastructures. Recognizing the connection between the political, the physical network, and the requirements for security in the sociopolitical reality of the Internet offers the potential for more robust defense and earlier indicators. Not taking advantage of this would be an ideological, not technical, choice.

References

[1] K. Benton and L. J. Camp, "Examining the Jurisdictions of Internet Routes to Prevent Data Exfiltration," 44th Research Conference on Communications, Information and Internet Policy (TPRC44), 2016.

[2] K. Benton and L. J. Camp, "Firewalling Scenic Routes: Preventing Data Exfiltration via Political and Geographic Routing Policies," in *Proceedings of the 2016 ACM Workshop on Automated Decision Making for Active Cyber Defense* (*SafeConfig '16),* ACM, 2016, pp. 31–36.

[3] K. Benton, L. J. Camp, and M. Swany, "Bongo: A BGP Speaker Built for Defending against Bad Routes," in *MILCOM 2016 IEEE Military Communications Conference*, IEEE, 2016, pp. 735–739.

[4] T. Mizuguchi and T. Yoshida, "Inter-Domain Routing Security ˜BGP Route Hijacking˜," Asia Pacific Regional Internet Conference on Operational Technologies (APRICOT '07), 2007.

[5] P. Moriano, S. Achar, and L. J. Camp, "Incompetents, Criminals, or Spies: Macroeconomic Analysis of Routing Anomalies," *Computers & Security*, vol. 70, 2017, pp. 319–334.

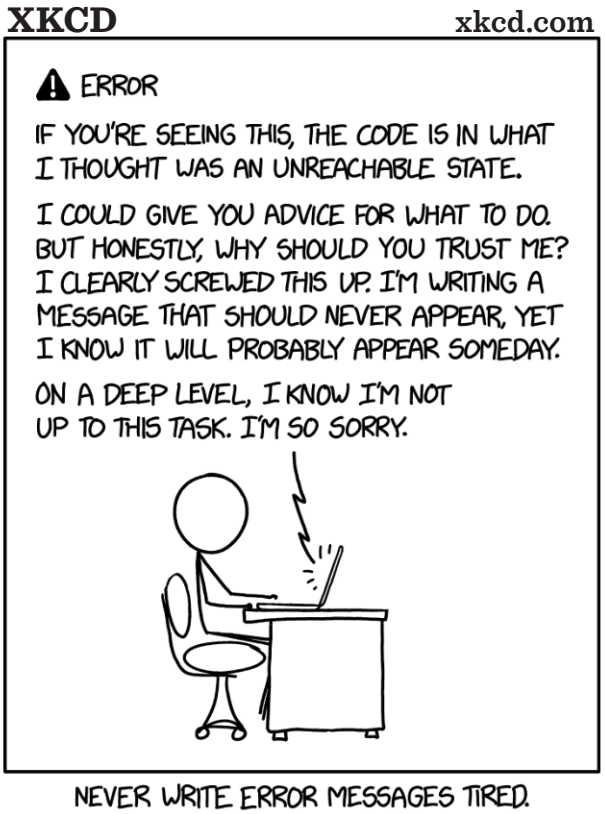
[6] P. Moriano, R. Hill, and L. J. Camp, "Using Bursty Announcements for Early Detection of BGP Routing Anomalies": https://arxiv.org/abs/1905.05835, 2019.

[7] L. Petiniaud and L. Salamatian, "Geopolitics of Routing," RIPE Network Coordination Center: https://labs.ripe.net/Members/louis_petiniaud/geopolitics-of-routing, 2019.

[8] V. Valancius, N. Feamster, J. Rexford, and A. Nakao, "Wide-Area Route Control for Distributed Services," in *Proceedings of the USENIX Annual Technical Conference (ATC '10)*, pp. 17–30: https://www.usenix.org/legacy/events/atc10/tech/full_papers/Valancius.pdf.

[9] M. Wählisch, O. Maennel, and T. C. Schmidt, "Towards Detecting BGP Route Hijacking Using the RPKI," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, 2012, pp. 103–104.

[10] Z. Zhang, Y. Zhang, Y. C. Hu, and Z. M. Mao, "Practical Defenses against BGP Prefix Hijacking," in Proceedings of the 2007 ACM Conference on Emerging Network Experiment and Technology (CoNEXT 2007), article 3.



50 Ways to Leak Your Data

An Exploration of Apps' Circumvention of the Android Permissions System

JOEL REARDON, ÁLVARO FEAL, PRIMAL WIJESEKERA, AMIT ELAZARI BAR ON, NARSEO VALLINA-RODRIGUEZ, AND SERGE EGELMAN

Joel Reardon is an Assistant Professor at the University of Calgary. He received his master's degree from the University of Waterloo, his doctoral degree from the ETH Zurich, and spent a postdoctoral year at UC Berkeley and the International Computer Science Institute (ICSI). His research interests relate to security and privacy, including storage compliance issues as well as systems to make it easier to use. He also loves mountains, bicycles, and writing poetry. joel@moosematch.com

Álvaro Feal is a second-year PhD student working at IMDEA Networks Institute. He focuses on analyzing privacy threats to web and mobile applications from a technical and regulatory perspective. Prior to working at IMDEA Networks, Alvaro interned at IMDEA Software, working in Android privacy and anonymous communication systems. He has published in peer-reviewed conferences such as ACM IMC, USENIX Security, and workshops like IEEE Consumer Protection (ConPro). Álvaro's work received a Distinguished Paper Award at USENIX Security 2019. alvaro.feal@imdea.org

Primal Wijesekera is a Research Scientist in the Usable Security and Privacy Group at the International Computer Science Institute at Berkeley and is a Postdoctoral Researcher in the Electrical Engineering and Computer Science Department at UC Berkeley. His prior work includes contextual permission models for Android, mobile app analysis for privacy and security violations. His recent work focuses on smart speakers for the home, vulnerability discovery mechanisms in the wild, and ecosystems surrounding fake news. primal@berkeley.edu

Smartphones are general-purpose computers that store a great deal of sensitive personal information. Apps are prevented from accessing this information at will through the use of a *permission system* at the operating-system level. These security mechanisms are reasonable because we carry our smartphones alongside us all day, and they can gain access to our intimate communications and social network, our web browsing history, our location at all times—even if the GPS is disabled. When apps are denied permissions, however, they still have options to cheat the permission system by using side and covert channels. In our research we found a small number of such channels being actively exploited when we tested Google Play Store apps.

Are Mobile Permission Models Bullet-Proof?

There are lots of valid criticisms for the current permissions system. Users cannot reliably understand what permissions mean or why they are needed. Apps request more permissions than necessary. Users don't have easy means to find alternate apps that request fewer permissions, or to omit search results for apps that request dangerous permissions, like being able to turn on your microphone at all times.

The increasing presence of third-party software development kits (SDKs) in mobile applications amplifies the dissemination of personal data from mobile applications to online services. Most developers use third-party SDKs in their apps for advertising, analytics, crash reporting, or social network integration [5]. Both Android and iOS permission models allow third-party SDKs to piggyback on the permissions that the user grants to the host app. Unfortunately, users cannot distinguish between a permission given to enable a feature in the app and one to be used by a data-hungry third-party SDK [5].

StartApp's official guidance for integrating its SDK into apps provides a perfect example of this problem. It tells developers that it will improve performance if they add extra permissions for location, Bluetooth, and silent starting on boot [4]—that is, it tells developers to add location access to their apps even though the apps would have no legitimate need for such access. Users aren't made aware when permissions have been requested by an advertising library that simply wants to track them and harvest their private information.

Additionally, mobile apps can circumvent the permission model and gain access to protected data without user consent by using both *covert* and *side channels*, attacks described in Figure 1. Side channels manifest through vulnerabilities present in the implementation of the OS permission system that allow apps to access protected data and system resources without permission. Covert channels manifest when inter-app communication, which may be legitimate, is leveraged for illegitimate purposes, such as having one app abuse its privileges by acting as a facade for another app's desire to access permission-protected data.

Amit Elazari Bar On is a Director of Global Cybersecurity Policy at Intel Corporation and a Lecturer at UC Berkeley's School of Information. She holds a JSD from UC Berkeley School of Law and graduated summa cum laude in gaining three prior degrees. Her research in cybersecurity law and policy has appeared in leading journals, has been presented at conferences such as RSA, Black Hat, and USENIX Security, and was featured at leading news sites such as the *New York Times*. She practiced law in Israel. amit.elazari@berkeley.edu

Narseo Vallina-Rodriguez is an Assistant Research Professor at IMDEA Networks and a Research Scientist at ICSI in Berkeley. His research interests fall in the area of network measurements, privacy, and security. Narseo's work has received distinguished paper awards at USENIX Security 2019, ACM IMC 2018, and ACM CoNEXT 2014, and was awarded the IETF Applied Networking Research Prize in 2016. Narseo's research in mobile privacy has influenced industry practices and regulation and has been covered by international media outlets. narseo.vallina@imdea.org

Serge Egelman is the Research Director of the Usable Security and Privacy Group at the International Computer Science Institute. He conducts research to help people make more informed online privacy and security decisions. He has received the USENIX Distinguished Paper Award, seven ACM CHI Honorable Mention Awards, and the SOUPS Impact and best paper awards; his research has been cited in numerous lawsuits and regulatory actions, as well as being featured in the *New York Times, Washington Post,* and *Wall Street Journal.* egelman@cs.berkeley

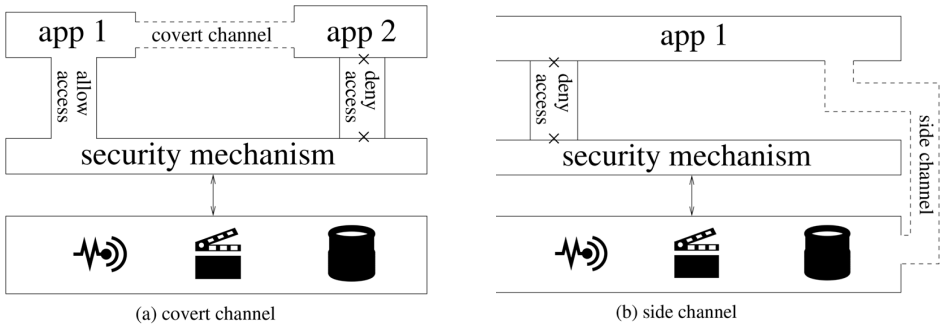


Figure 1: Covert and side channels. (a) A security mechanism allows app1 access to resources but denies app2 access; this is circumvented by app2 using app1 as a facade to obtain access over a communication channel not monitored by the security mechanism. (b) A security mechanism denies app1 access to resources; this is circumvented by accessing the resources through a side channel that bypasses the security mechanism.

Discovering Covert and Side Channels in the Wild

Previous research focused on understanding personal data collection using system-supported access control mechanisms (i.e., Android permissions). The research community has also explored the potential for covert channels in Android using local sockets, shared storage [2], and other unorthodox means, such as using vibrations to send data and accelerometers to receive it [1]. Accelerometer data can further act as a side channel to uniquely identify the user [9, 11] or infer demographic data such as gender [3]. However, there has been little research in detecting and measuring at scale the prevalence of both covert and side channels in apps that are available in the Google Play Store.

Instead of imagining new channels, our USENIX Security '19 paper focuses on collecting evidence of apps abusing side and covert channels in practice [7]. We leveraged our AppCensus app auditing platform to search for instances of Android applications disseminating permission-protected data over the network without requesting the permission to access it. We then reverse engineered the apps and third-party libraries responsible for this behavior to determine how the unauthorized access occurred.

It is important to note that AppCensus is not a regular security-oriented sandbox: detecting and analyzing both privacy abuses and regulatory violations require specific research methods. To that end, AppCensus implements mechanisms to exhaustively monitor apps' runtime behavior and personal data leaks at system and network levels, including a TLS man-in-the-middle proxy. Then, we leverage heuristics inspired by different regulatory frameworks to contextualize these observations and to hunt for potential abuses and violations.

Research Findings

We automatically executed over 88,000 Android apps in our AppCensus platform to see when permission-protected data was transmitted by the device, and scanned the permissions that apps requested to see which ones *should not even be able* to access the transmitted data in the first place (Figure 2). We focused on a subset of the *dangerous* permissions that prevent apps from accessing location data and unique identifiers. We grouped our findings by *where* on the Internet data was sent and *what* data type was sent, as this allows us to attribute the observations to the actual app developer or embedded third-party libraries. We then reverse engineered the responsible component to determine exactly how the data was accessed so that we could statically analyze our entire data set to measure the prevalence of each attack. We found the following side and covert channels being exploited in Google Play Store apps:

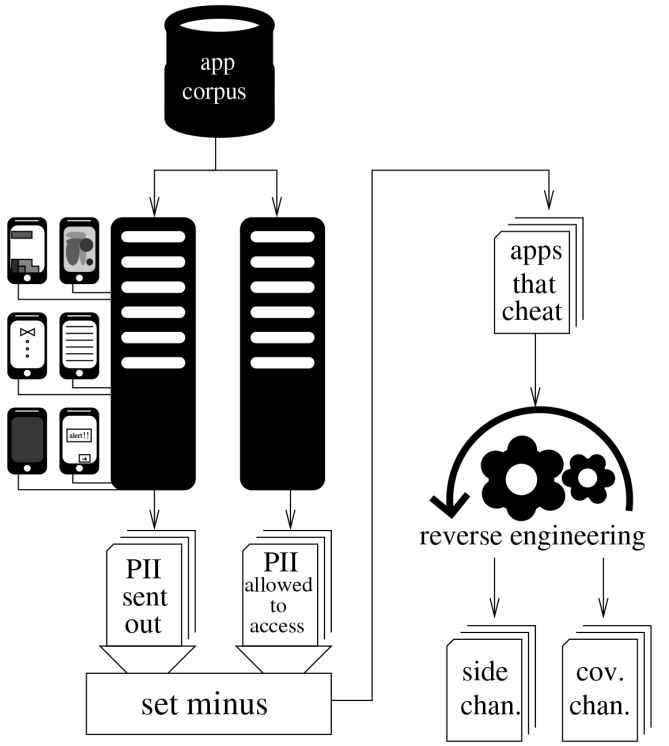


Figure 2: Overview of our analysis pipeline. Apps are automatically run, and the transmissions of sensitive data are compared to what would be allowed. Those suspected of using a side or covert channel are manually reverse engineered.

- We discovered apps getting the BSSID of the connected WiFi Access Point (i.e., the router's MAC address) by reading the OS ARP cache (/proc/net/arp), which was not protected by permissions. This information can be used as a surrogate for location data. We found five apps exploiting this vulnerability and 355 with the pertinent code to do so.
- We discovered Unity (a popular third-party cross-platform game engine and advertising network) obtaining the device MAC address of the device using ioctl system calls. This information can be used to track users even if they factory reset their devices. We found 42 apps exploiting this vulnerability and 12,408 apps with the pertinent code to do so. We realized (after our paper was published) that starting from the version of Android we used (Marshmallow), all attempts to access the MAC address of the device return a fake value of 02:00:00:00:00:00—even if the access network state permission is granted; therefore all 711 apps that transmitted the MAC address must have accessed it this way.
- We also discovered that third-party libraries provided by two Chinese companies—Baidu and Salmonads—independently make use of the SD card as a covert channel, so that when an app can read the phone's IMEI, it stores it for other apps that cannot. We found 159 apps with the potential to exploit this covert channel and empirically found 13 apps doing so.
- We found one app, Shutterfly, that used picture metadata as a side channel to access precise historical location information despite not holding location permissions. It included code that processed location from the raw EXIF data; that is, it copied the data intentionally instead of simply uploading photos and having location data by mistake.

The Impact of Our Work

The permissions system is not perfect, but it serves an important purpose. Requesting permission serves as a system to give users *notice* about the app's behavior; users installing apps further serves as a system of *consent*. The use of deceptive practices like covert and side channels is unacceptable as they not only undermine users' privacy and consumer rights, but they also give rise to legal and regulatory concerns. Circumventing the permissions system means that notice was not given nor consent obtained. In one case, the third-party library OpenX first *tried* to obtain the WiFi BSSID through the permission system and only went the cheating route through the ARP cache when it saw that it was denied access.

Data protection legislation around the world, like the General Data Protection Regulation (GDPR) in Europe or the California Consumer Privacy Act (CCPA), enforce transparency on the data collection, processing, and sharing practices of mobile applications. In this regulatory context, designing and using these techniques suggests an actual attempt to access data without user consent. Developers and SDK providers implementing these techniques have to take extra measures to set up covert channels or discover side channels that can be exploited. We responsibly disclosed our findings to Google, so they could address the issues in the Android operating system, as well as the US Federal Trade Commission (FTC). Google has given us a bug bounty for our efforts.

Our Stepping Stones

Our research originates from a line of work designed to improve the accuracy and usability of the Android permission system [10]. Anyone who has installed an app on Android and paid attention to the permissions that are requested has probably run into one that demands permissions that fall well outside their scope, like an alarm clock app that needs to read your SMSes. The best explanation we've come up with is that this allows someone trusted to set important alarms for you after you've gone to sleep—like if there's going to be a huge dump of fresh snow in the mountains and they'll come to pick you up.

Part of this earlier work involved instrumenting the permission system to track permission usage by apps and collecting ground truth data about how users would prefer to handle those permission requests. This knowledge was used to inform a machine learning classifier that significantly improved the permission granting accuracy over the existing ask-on-first-use and was much better than the ask-on-install ultimatum.

This work was followed by a field study where we built a modified version of Android that actually enforced denying permissions. We did this gracefully when possible and used both user input and our machine learning classifier. Users liked the control they got, and our results from earlier studies were validated. One observation from our field studies was that apps made frequent requests to access data protected by sensitive permissions.

In parallel, another line of research involved studying trackers—all the data-hungry ads and analytics companies that are spying on users—in the mobile ecosystem and personal data dissemination over the network [6]. This study took advantage of a purpose-built man-in-the-middle VPN on Android, the Lumen Privacy Monitor, a tool that can monitor applications' traffic locally on the device, even if encrypted. Lumen allowed us to build a database of all network traffic going to different organizations that an app contacts.

Spying on Children

These lines of research joined together when some of us decided to read the Children's Online Privacy Protection Act (COPPA)—a particularly strong privacy regulation with serious consequences for violations—and realized that, based on what we've seen in practice, there's *no way* that *all* of these apps are in compliance. Plus, we have all the tools to monitor for this. We combined our OS instrumentation with our traffic monitoring to obtain evidence of applications' actual runtime behavior regarding *when* personal data is accessed and *where* it is sent. We could automate our analysis and thus scale our study by simulating human interaction with apps using the Android Automator Monkey, which is essentially a UI fuzzer for testing purposes.

Our findings about COPPA compliance in children-oriented Android apps were shocking [8]. The majority of children's games are sending persistent identifiers to ads and analytics companies capable of tracking them. Ten percent are sending the IMEI of the device, which is like an un-resettable super cookie of infinite tracking. *Four percent* were sending *precise geolocation*, for which COPPA requires *verified* parental consent to access. How on earth a company can feel confident in having verified parental consent from a system that randomly clicks and swipes, we'll never know!

For apps that we know *used* the location permission while running but that we didn't catch sending location, we found a bunch of obfuscated location sending happening. This category of app includes the company StartApp, which Google lists as one of their accepted children advertisers in their updated designed for families program (https://android-developers.googleblog.com/2019/05/building-safer-google-play-for-kids.html). StartApp was using a Vernam-style cipher to XOR in two repeating masks (`$T@RT@PP` and `ENCRYPTIONKEY`) and in doing so were transmitting precise geolocation and even WiFi scan data including router MAC and signal strength.

From all these stepping stones we end up at this work. We have the ability to run lots of apps at scale, to monitor their network traffic, and to scrutinize the permissions that they request in runtime. So we compared these two sets: what's the data an app is *allowed* to access, and what's the data that an app *actually* sends out on the Internet. Are there any transmissions by an app that didn't have permission to access it in the first place?

Our Confession

Now it's time for our confession. Our original goal in our methodology was not to discover and disclose these side and covert channels; we were actually looking for bugs in our own code but discovered these attacks by chance. That is, we implicitly assumed that the Android permission system was absolutely sound and were looking for false positives in our data set because, as we imagined, if we flagged the transmission of the IMEI without the `READ_PHONE_STATE` permission, it *must* be the result of a bug in our code.

A few false positives and negatives can be expected with such large-scale work, and we spot-checked lots of flagged transmissions of PII but by no means manually every transmission (so we'll have some false positives). And we looked at lots of packets trying to find all sorts of obfuscations, but there are many that still confound us (so we expect some false negatives as well). Still, as long as we do enough manual checking of our findings, the false positive rates are statistically low enough to not have any impact on *headline results* like four percent of apps sending location.

But our study on rampant (potential) privacy violations in thousands of children's games was getting media and regulatory attention. This prompted us to become extra certain of our findings. Being confident about the average value is no longer enough, and rooting out any false positives became even more crucial. We can live with the false negatives (where we don't catch a company who is actually sending data), but now false positives have become critical to avoid, because even one false positive casts doubt on any *specific* finding that we claim. For example, in response to a letter from one of the lawyers at IronSource who did not like our characterization of their behaviors, we double-checked our results in order to verify our initial findings and actually found more things we had missed!

So we went looking for false positives. We filtered out all the data where the app had the corresponding permission, assuming that what was left must all be false positives. And in fact we did find some! One favorite was the fact that we did our tests in Berkeley, California, which has an area code of 510—it so happened that some of our testing began with the UNIX timestamp 1510 and so there's a block of time during which a harmless timestamp was misconstrued as apps transmitting the user's phone number.

Another was the fact that IP-based geolocation happened to be surprisingly accurate for IPs from our research institute. Perhaps this was because we uploaded both our IP and location thousands of times after running all these apps, and eventually the Internet learned where this IP was. Digging deeper, however, we found that this did not replicate at other locations and with other IPs. Finally, some apps sent really invasive fingerprints, including the hostname of our own machines that built our custom Android version, and it just so happened that the SSID of our WiFi router was a substring of that.

Our hunt was a useful exercise and we fixed all the false positives that we found, making our tools more robust and reliable. But we also found true positives. We found actual transmissions of data carrying the correct values and (unlike incoming geolocation) first seen as an outgoing transmission from the app. It turns out that we found evidence consistent with the use of side and covert channels, and in order to figure out what exactly was going on we had to start reverse engineering. The results of this exercise were those four side and covert channels we presented earlier in the article: ioctls, EXIF metadata, ARP cache, and plain old sharing data on the SD card. And in so doing we put app and SDK developers on notice that, going forward, we are looking for these kinds of deceptive and fraudulent practices.

Acknowledgments

This work was supported by the US National Security Agency's Science of Security program (contract H98230-18-D-0006), the Department of Homeland Security (contract FA8750-18-2-0096), the National Science Foundation (grants CNS-1817248 and grant CNS-1564329), the Rose Foundation, the European Union's Horizon 2020 Innovation Action program (grant Agreement No. 786741, SMOOTH Project), the Data Transparency Lab, and the Center for Long-Term Cybersecurity at UC Berkeley. The authors would like to thank John Aycock, Irwin Reyes, Greg Hagen, René Mayrhofer, Giles Hogben, and Refjohürs Lykkewe.

References

[1] A. Al-Haiqi, M. Ismail, and R. Nordin, "A New Sensors-Based Covert Channel on Android," *The Scientific World Journal*, September 2014.

[2] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun, "Analysis of the Communication between Colluding Applications on Modern Smartphones," in *Proceedings of the 28th Annual Computer Security Applications Conference* (ACM, 2012), pp. 51–60.

[3] Y. Michalevsky, D. Boneh, and G. Nakibly, "Gyrophone: Recognizing Speech from Gyroscope Signals," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security '18)*, pp. 1053–1067: https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-michalevsky.pdf.

[4] S. Milo, StartApp SDK Android—Android (Standard): https://support.startapp.com/hc/en-us/articles/360002411114-Android-Standard-, 2019.

[5] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, C. Kreibich, and P. Gill, "Apps, Trackers, Privacy, and Regulators: A Global Study of the Mobile Tracking Ecosystem," in *Proceedings of the Network and Distributed System Security Symposium (NDSS '18)*.

[6] A. Razaghpanah, N. Vallina-Rodriguez, S. Sundaresan, C. Kreibich, P. Gill, M. Allman, and V. Paxson, "Haystack: In Situ Mobile Traffic Analysis in User Space, *arXiv preprint arXiv:1510.01419v1*, (2015), pp. 1–13.

[7] J. Reardon, Á. Feal, P. Wijesekera, A. Elazari Bar On, N. Vallina-Rodriguez, and S. Egelman, "50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*, pp. 603–620: https://www.usenix.org/conference/usenixsecurity19/presentation/reardon.

[8] I. Reyes, P. Wijesekera, J. Reardon, A. Elazari Bar On, A. Razaghpanah, N. Vallina-Rodriguez, and S. Egelman, "Won't Somebody Think of the Children? Examining COPPA Compliance at Scale," in *Proceedings on Privacy Enhancing Technologies*, 2018, no. 3, pp. 63–83.

[9] L. Simon, W. Xu, and R. Anderson, "Don't Interrupt Me While I Type: Inferring Text Entered through Gesture Typing on Android Keyboards," in *Proceedings on Privacy Enhancing Technologies*, 2016, no. 3, pp. 136–154.

[10] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov, "Android Permissions Remystified: A Field Study on Contextual Integrity," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security '15)*, pp. 499–514: https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-wijesekera.pdf.

[11] J. Zhang, A. R. Beresford, and I. Sheret, "SensorID: Sensor Calibration Fingerprinting for Smartphones," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP)*, IEEE, May 2019.

Building an Nmap for Your Car

SEKAR KULANDAIVEL, TUSHAR GOYAL, ARNAV KUMAR AGRAWAL, AND VYAS SEKAR

Sekar Kulandaivel is a PhD candidate in electrical and computer engineering at Carnegie Mellon University. Sekar's interests lie in automotive network and systems security with a focus on practical solutions. skulanda@andrew.cmu.edu

Tushar Goyal currently works at Microsoft Search. His primary focus is in systems, computer architecture, and security. He is a graduate of Carnegie Mellon University. tgoyal1@alumni.cmu.edu

Arnav Kumar Agrawal currently works at Microsoft on the Xbox Live Graph Team. His focus is primarily on large-scale distributed systems and security. He is a graduate of Carnegie Mellon University. akagrawa@alumni.cmu.edu

Vyas Sekar is the Angel Jordan Early Career Chair Associate Professor in the ECE Department at Carnegie Mellon University, with a courtesy appointment in the Computer Science Department. His research is in the area of networking, security, and systems. His work has received best paper awards at ACM SIGCOMM, ACM CoNext, and ACM Multimedia, and the NSA Science of Security prize. vsekar@andrew.cmu.edu

The network inside a modern car is no longer static; modern in-vehicle networks grow more complex and can change over time. We have developed CANvas, a network mapping tool that identifies what devices in your car communicate on the network and how messages are exchanged between them. CANvas helped us identify an unknown device in a modified 2009 Toyota Prius and pinpoint potentially vulnerable devices in a 2017 Ford Focus.

In 2015, two security researchers, Charlie Miller and Chris Valasek, remotely hacked into a Jeep Cherokee to demonstrate the potential impact of a hack against a modern car [6]. They accomplished their goal by compromising one of the vehicle's computers, known as Electronic Control Units (ECUs), and then used this compromised ECU to communicate with other ECUs in the car over the vehicle's Controller Area Network (CAN) bus. To figure out what devices were a part of this network, they had to physically disconnect components, which is a painstaking process even for analyzing a single car. As vehicles continue to integrate more electronics and contain increasingly complicated networks, we need a tool that can produce a map of the car's network to help us keep our vehicles protected without having to take apart a car.

The obvious solution here is to ask the automaker for the network map of the cars that we own. However, this is highly proprietary information that automakers are not willing to give out even to mechanics and researchers. Even if we could obtain this network map from an automaker, we face a new challenge in today's world: these networks are no longer static. Once a car leaves the factory, the automaker loses control of what devices are connected to the network and how they interact with each other. We now look at a few scenarios where you can expect your car's network to change.

Imagine if you took your car to a potentially untrustworthy mechanic. Under the guise of a repair, this mechanic could add a new device to your car's network without your permission. The number of components in your car that communicate on the network is getting larger and larger; you can even buy headlights that talk on the CAN bus. Consider another scenario: imagine if you had a traction control ECU replaced by even a trusted mechanic, but the ECU was counterfeit and was not programmed to send the correct messages. In this case, you may only discover the ECU was counterfeit *after* you needed your traction control.

Perhaps you want to replace the radio in your modern car, so you decide to buy a replacement radio from Amazon or eBay. As observed in some modern vehicles, this radio could be connected to your vehicle's CAN bus, and a malicious seller could program that radio to transmit new or different information onto the network. We also see that automakers are now considering pay-as-you-go services where your car comes pre-installed with hardware, like a turbocharger or high-output batteries. These automakers envision customers paying to activate these features, which will introduce new communication between the computers in a car's network. In defending our vehicles against attacks, the ability to differentiate between expected traffic from these services versus malicious traffic from an attack could prove useful.

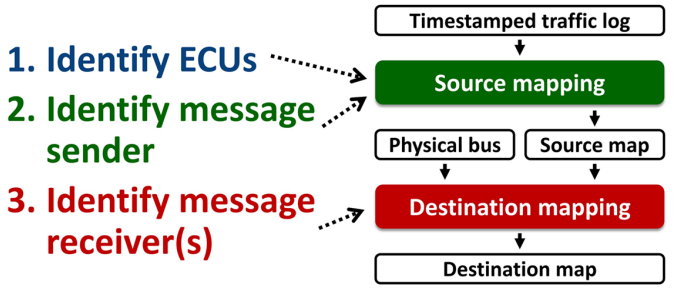


Figure 1: Design overview of the CANvas network mapper.

Now consider recent work that demonstrated a new type of attack that can shut down any ECU in your car by compromising just a single ECU in your vehicle [3]. A requirement for this attack is that the compromised ECU must receive messages from its victim. Since shutting down an ECU while the car is in motion could be potentially catastrophic, this type of attack has a real and significant impact. Imagine if that radio you bought from an online seller was programmed to launch the shutdown attack against your vehicle's engine ECU while it was in motion.

It is clear from these scenarios that the network inside a modern car can change. As automakers produce new models of a car each year and even provide over-the-air update capabilities, the frequency of producing updated network maps for a vehicle will quickly increase. To ensure that security researchers and vehicle owners can keep up with these networks and understand what goes on in our cars, we need to build a network mapping tool that is both practical and accessible.

Building a Practical Mapper

As a first stepping stone to future automotive security work, we developed CANvas, a fast and inexpensive automotive network mapping tool for a vehicle's CAN bus [5]. For less than 50 dollars of hardware and under 30 minutes, CANvas produces a network map that can identify the ECUs in a car as well as where messages originate from and which ECUs receive each message. We focus on building a practical tool that works on real modern vehicles.

Current approaches for mapping a vehicle's network require physically taking apart the car. Instead of requiring our users to go through this extensive process, we aim to build a mapper that satisfies a few practical goals. Since we require the vehicle to be on and running, our mapper should not take hours of time to complete. It should also be inexpensive and not require the use of an oscilloscope or logic analyzer. Anything we do to the network should leave no permanent damage, and the mapper should simply plug into the the On-Board Diagnostics (OBD-II) port of your car, which, starting in 1996, all vehicles manufactured or sold in the United States are required to have. Our users should not expect to cut into the network of their car or worry that their car will have permanently lit malfunction indicator lights.

We define three main outputs for our network mapper: (1) a list of all active ECUs in the vehicle, (2) the transmitting ECU for each unique CAN message, and (3) the set of receiving ECUs for each unique CAN message. As seen in Figure 1, we combine our first two outputs into a module called *source mapping* and our third output into a module called *destination mapping*. The source-mapping module takes in a timestamped traffic log as input and produces a source map that identifies the source ECU of each message. Then the destination mapping module uses the source map and access to the physical CAN bus of a running vehicle to produce a destination map that identifies which ECUs receive each CAN message.

A significant challenge in designing this tool is the broadcast nature of the CAN protocol, as network messages contain no information about their sender or recipients. To address this challenge, we repurpose insights that were previously proposed for an intrusion detection system for the CAN bus [4] and a shutdown attack against CAN-enabled ECUs [3]. Since the techniques used in these works had limitations when applied to the mapping problem, we develop two approaches for our source and destination mapping modules, respectively: a pairwise clock offset tracking algorithm that identifies transmitting ECUs and a forced ECU isolation technique that identifies receiving ECUs. In designing these techniques, we focus on the practical challenges that we face when mapping real vehicles.

In the rest of this article, we detail our adventures in mapping our two main test vehicles and our vision for how the CANvas mapper can be used for future research. This article is based on a conference paper that appeared at the 2019 USENIX Security Symposium, which presents our design for CANvas, our experiments, and other considerations for mapping in detail [5].

The Hidden ECU in a Toyota Prius

We managed to acquire a hand-me-down 2009 Toyota Prius from a different department at the university. This Prius was converted into an all-electric vehicle with a lithium-ion battery installed in its trunk and became our primary ground truth for the mapper. Unfortunately, the only method at the time for obtaining the ground truth was to physically dismantle the car and gain direct access to all of the ECUs in the Prius as seen in Figure 2. Going through this process for just a single car made us thankful for the prospect of a network mapping tool.

To figure out how to take apart the car and find each of the ECUs in the car, we did what the mechanics at the dealership would do: we went to the service website for Toyota. After paying a small subscription fee, we followed the removal instructions for each ECU and found that this Prius contained eight ECUs. With this direct access, we could splice into each ECU's connector and determine what messages it sent and received. This information

Figure 2: Physically tearing apart any vehicle is exhausting; imagine doing this every time we need an updated network map.

served as our ground truth for verifying our output from the CANvas network mapper.

For identifying the source of each network message, one of the key assumptions we make is the periodicity of messages on the CAN bus. From prior work [4], we knew that the majority of messages on the network should be periodic in nature. When we connected to the network, we found that all messages at first *seemed* to be periodic. However, when we measured the average period and the deviation of the period, we found several examples of messages that were "almost" periodic. Some messages stopped transmitting for seconds at a time, some seemed to miss their deadlines and re-transmit at the next period, and others seemed to have two different periods that resulted in overlapping messages.

Where prior work only investigated purely periodic messages, we found that these "almost" periodic messages are expected in real vehicles. We discussed this finding with sources from the automotive industry, and they confirmed that CAN message transmissions can be complex and that there are many circumstances that affect how an ECU transmits its messages. To address these special cases, we designed CANvas to detect these instances and employed a slightly modified approach for identifying the source of these messages as detailed in our paper [5]. At its core, these messages have some notion of periodicity, which allows us to use our method of mapping messages to their source ECUs.

With the source-mapping component of our network mapper completed, we plugged our tool into the Prius's diagnostic port and expected to run CANvas and find eight ECUs as discovered in our ground truth; much to our surprise, we found *nine* ECUs. After scratching our heads, we decided to unplug all eight of the expected ECUs and see if we still saw traffic on the network. We still saw three messages and detected a single transmitting ECU with no receivers. We confirmed this finding with other online sources that did not see these three additional messages, so we knew that something must have changed in the car's network.

At this point, we knew something must be different with this car, especially when other cars of the same model year are missing these three messages on their networks. Looking into this Prius's history, we found that a new ECU was installed as part of the all-electric modification done almost a decade ago. Without this network mapping tool, we would have never known about this hidden ECU. Thankfully, this ECU was not malicious and was simply included as a modification from the previous department. But when we consider that this could have been a malicious device, the importance of having a network mapper becomes clear.

The Vulnerable ECUs of a Ford Focus

Our finding on the Prius was quite surprising, but we also wanted to test our network mapper on a newer car. We managed to find a salvaged 2017 Ford Focus with minor flooding but all of its electronics completely intact. As seen in Figure 3, we physically dismantled this salvaged car to obtain the ground truth, and we tested our network mapper on it. For this vehicle, we used the Ford service website for instructions on ECU removal, and we identified nine total ECUs.

To identify the destinations of each network message, we borrowed an insight from prior work [3] on shutdown attacks against ECUs. Due to limitations in this work, we implemented a different shutdown technique that permits CANvas to analyze each ECU one by one. Unfortunately, we came across an interesting challenge; we found that some ECUs automatically recovered or did not even shut down. For the purposes of network mapping, we had to make sure that some ECUs remained in the shutdown state as detailed in our paper [5]. Upon closer inspection, we found that these ECUs do remain in a shutdown state for some time. With additional add-on techniques, we can suppress an ECU and keep it from coming back online, allowing us to perform our destination mapping.

When mapping a modern car, we expected that automakers would limit what messages can be received by each ECU since all ECUs do not need to communicate with each other. For example, we would not expect a radio ECU in the Focus to receive messages from ECUs related to the powertrain. Using our network map of the Focus, we found that all of its ECUs were capable of receiving all messages on the network. This means that any ECU compromised by some future remote attack could be used to launch attacks on other ECUs, including the safety-critical powertrain ECUs. One might think that this type of attack has not been significant or realistic enough for automakers to implement restrictions on what messages an ECU can receive.

Figure 3: We found that all ECUs in this Focus could potentially launch a recent shutdown attack.

However, we find that the industry acknowledges this potential and is taking measures to defend against this. For purposes of restricting the messages that reach an ECU, Next eXPerience (NXP) is working on a new CAN transceiver, the NXP TJA115x Secure CAN Transceiver family [2]. If automakers implement these types of hardware-level filters on what messages an ECU could receive, a tool like the CANvas mapper could help ensure that these filter settings are correct. Until then, we have found that even a modern car like the Focus has no filter on the messages its ECUs receive.

Next Steps from Network Mapping

With the ability to network map a car, we envision several extensions to the mapper and alternative tools that could benefit from CANvas. A vehicle's network map could benefit from richer functionality, such as identifying the function of an ECU (transmission ECU, engine ECU, etc.), identifying gateway ECUs that potentially bridge multiple CAN buses, and identifying message filters implemented in software. We also envision extending this work into other automotive protocols, including Automotive Ethernet and Local Interconnect Network (LIN), to provide a broader map of a car's complete network and not just its CAN bus.

In previous conversations with industry, we asked how this network mapping tool could prove useful even though industry has the original network map. Besides being used to detect unauthorized changes to the network, the network mapper could be used to verify the state of the network. Over time, parts of the ECUs and their network could fail, such as diodes and wiring. These failures could change the output of the network map and indicate issues with the network. Instead of being used directly for security, CANvas could also serve as a network validation tool for your mechanic to use when changing the configuration of your vehicle's CAN bus.

We have also made interesting discoveries when we look at how messages are transmitted. When running our network mapper on a vehicle, we see changes in the contents of a message based on what ECUs are shut down during our destination mapping. We could use this new information to potentially infer details that are only found in the automaker's proprietary network map. Since we can know which ECU sends each message, we could implement better techniques to reverse engineer the message contents by correlating changes between an ECU's source messages. Additionally, when designing an attack, a network map could tell us what messages need to be replicated if we ever wish to pursue a masquerade attack. CANvas serves as a building block for future security research, and we have made it publicly available with the hope that the community will help us expand its capabilities in the future [1].

References

[1] CANvas network mapper: https://github.com/sekarkulandaivel/canvas.

[2] NXP TJA115x Secure CAN Transceiver family: https://www.nxp.com/docs/en/fact-sheet/SECURCANTRLFUS.pdf.

[3] K. Cho and K. G. Shin, "Error Handling of In-Vehicle Networks Makes Them Vulnerable," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, ACM, 2016, pp. 1044–1055: https://dl.acm.org/citation.cfm?id=2978302.

[4] K. Cho and K. G. Shin, "Fingerprinting Electronic Control Units for Vehicle Intrusion Detection," in *Proceedings of the 25th USENIX Security Symposium (USENIX Security '16)*, 2016, pp. 911–927: https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_cho.pdf.

[5] S. Kulandaivel, T. Goyal, A. Kumar, and V. Sekar, "CANvas: Fast and Inexpensive Automotive Network Mapping," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*, 2019, pp. 389–405: https://www.usenix.org/conference/usenixsecurity19/presentation/kulandaivel.

[6] C. Miller and C. Valasek, "Remote Exploitation of an Unaltered Passenger Vehicle," Black Hat USA, 2015: http://illmatics.com/carhacking.html.

12th USENIX Workshop on Cyber Security Experimentation and Test (CSET '19)

PETER A. H. PETERSON AND ROB G. JANSEN

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. pahp@d.umn.edu

Dr. Rob Jansen is a Computer Scientist and a Jerome and Isabella Karle Distinguished Scholar Fellow at the US Naval Research Laboratory with research expertise in the areas of computer security and privacy, distributed systems, and parallel and distributed simulation. His research results have been published in the top peer-reviewed computer security and privacy conferences and workshops, and his work demonstrates an ability not only to develop and apply theoretical concepts, but also to build, evaluate, deploy, and measure real-world systems. When not in the lab, Rob enjoys jogging around the National Mall and through the streets of DC. rob.g.jansen@nrl.navy.mil

On Monday, August 12, 2019, 55 attendees joined us for the 12th USENIX Workshop on Cyber Security Experimentation and Test (CSET '19) in Santa Clara, California. CSET, one of the USENIX Security Symposium's co-located workshops, welcomes work in the broad categories of "cybersecurity evaluation, experimentation, measurement, metrics, data, simulations, and testbeds"—that is, research about research tools, data, and methods. The purpose of this article is to share our experience chairing CSET '19 and to highlight this year's papers.

Changes to the CSET PC

We made some experimental changes to the call for papers (CFP) and program committee (PC) this year, and we wanted to share them in the hope that they might be useful for other organizers. One of our main goals was to increase the community reach of the PC and the submission count, while reducing the PC review burden. To do this, we doubled the size of the PC to 46, inviting both established CSET community members and new people, including both junior and senior researchers. We also explicitly invited broad interpretations of the topics list. Additionally, we solicited a variety of paper lengths and types: traditional research papers, position papers, experience papers, preliminary work, and extended work. These could be long papers (eight pages), short papers (four pages), or extended abstracts (two-page talk proposals).

We explicitly invited preliminary work papers because CSET is a workshop; we wanted to encourage the lively discussion of new ideas, even if they were not fully developed. "Extended work" papers were meant to be expansions of security experimentation results, approaches, or tools developed in the course of other research (e.g., papers published at USENIX Security or elsewhere). Our rationale for soliciting these papers was that all security research requires an experimental approach; this often includes the development of tools, data, or knowledge that could be useful to the community. Unfortunately, these details are often drastically reduced in published papers due to space constraints. This cut material is often squarely in CSET's bailiwick, and we hoped that papers like this would be relatively easy for authors to prepare, interesting for attendees to discuss, and of service to the research community.

We are also happy to report that women comprised 46% of the CSET '19 PC, up from the recent peak of 32% in 2015. Women are in high demand and may already be committed to a full slate of PCs; to find the 21 women who were able to join the PC this year, we invited approximately double that number. Our takeaway was that it is absolutely possible to improve gender representation on PCs, but until the underlying diversity in our field improves, doing so may take a little time and effort.

Overall, our changes seemed to work well; we received 61 submissions, more than doubling 2018's submission count of 27. Each reviewer had approximately four papers to review. (We had wanted to limit each PC member to three reviews, but the volume of submissions precluded that.) Ultimately, we accepted 19 papers (31%). For more information about our process and statistics this year, please see our slides on the workshop site.

Sessions and Presentations

The 19 accepted papers this year were arranged into five sessions. The first session was "Cyberphysical and Embedded Testbeds and Techniques," chaired by Eric Eide (University of Utah). First, Paul Pfister (Iowa State University) presented a cyber-physical system (CPS) extension to ISEAGE, an event simulator used for cyberdefense competitions that included a physical model of a city, complete with LEDs representing system status. Next, Woomyo Lee (The Affiliated Institute of ETRI) presented a system for automatic generation of CPS research data about a power plant featuring a GE turbine, an Emerson boiler, and a FESTO water treatment system. After this, Sam Crow (UC San Diego) told us about Triton, a configurable testbed for avionics security research. Triton is, in the words of Crow, "real hardware from a real airplane that thinks it's running on an actual airplane in flight." Finally, we heard from Zachary Estrada (Rose-Hulman Institute of Technology) about CAERUS, a framework that is able to identify, through automated testing, timing sensitivities of undocumented embedded systems that can interact negatively with add-on security components.

Elissa Redmiles (Microsoft Research/Princeton University) chaired our "Data and Metrics" session. Michael Brown (Georgia Institute of Technology) described how debloaters can improve security by reducing the number of ROP gadgets through eliminating unimportant code, but also how they can accidentally introduce new high-quality gadgets. Instead of focusing on gadget count as the key metric, Brown proposes metrics based on gadget quality. Next, Aniqua Baset (University of Utah) discussed SecPrivMeta, an interactive website (secprivmeta.net) that provides visualizations of topic modeling on 36 years of security and privacy publications. After this, Josiah Dykstra (US Department of Defense) described how the NSA uses the Innovation Corps (I-Corps) methodology to improve the sharing of Cyber Threat Intelligence (CTI). Last, Jim Alves-Foss (University of Idaho) gave an entertaining talk containing a variety of cautionary tales of problematic data analysis and experimentation to admonish the community to use care and best practices in research.

"Usability, Effects, and Impacts" was chaired by Heather Crawford (Florida Institute of Technology). Zane Ma (University of Illinois at Urbana-Champaign) gave the first talk, which was about the effect of TLS and browser presentation on the success of phishing attacks in an A/B test on 266 users. Next, Victor Le Pochat (KU Leuven) described the design and evaluation of Tranco, a "top sites" ranking that aggregates Alexa, Majestic, Quantcast, and Umbrella, to create a stable and robust list for use by researchers. Third, Xiaodong Yu (Virginia Tech) presented work investigating how seven cache configuration parameters affected timing-based side-channel attacks; their talk included suggestions for improving security while minimizing performance impact. Last, Ildiko Pete (University of Cambridge) presented preliminary results from the Cambridge Cybercrime Center's analysis of usability issues with the data sets they share.

David Balenson (SRI International) chaired "Problems and Approaches," which began with Qiao Kang's (Rice University) presentation of their work automating the detection of attacks against the data planes of programmable routers. Our next presenter, Fatima Anwar (UCLA, now University of Massachusetts at Amherst) described how the timing capabilities of trusted execution environments (TEEs) can be vulnerable to timing attacks in realistic scenarios, and provided requirements for securing time facilities in these environments. Next, Sri Shaila G (University of California, Riverside) presented results of a study using IDAPro to reverse engineer the binaries of real-world IoT malware samples as compiled with various options, finding that, while unstripped binaries are amenable to analysis, performance on stripped binaries is generally poor. Last, Jonathan Crussell (Sandia National Laboratories) talked about their analysis of 10,000 experiments comparing differences between virtual and physical testbeds for research.

The final session of the day was "Testbeds and Frameworks," chaired by Jelena Mirkovic (University of Southern California Information Sciences Institute). Aditya Ashok (Pacific Northwest National Laboratory) described PACiFiC, a sufficiently realistic campus microgrid testbed model to allow a phish-to-blackout attack simulation. Second, Russell Van Dam (Sandia National Laboratories) presented Proteus, an emulation framework that supports the analysis of a wide variety of peer-to-peer distributed ledger technologies against different types of automated scenarios. Finally, Ryan Goodfellow (Information Sciences Institute) described the DComp Testbed, an open-source testbed using EVPN routing, a set of independently useful tools, and featuring a high level of abstraction and isolation.

For more detail, please see the workshop program online at www.usenix.org/cset19/program.

We would like to offer our sincere thanks to the fantastic USENIX staff, CSET's program and steering committees, authors, session chairs, shepherds, presenters, and attendees. The 13th CSET will once again be co-located with USENIX Security 2020 in Boston, with papers due in spring 2020. If you're interested in research around security experimentation, please consider submitting to and/or attending CSET next year!

Interview with Kirill Levchenko

RIK FARROW

Kirill Levchenko is an Associate Professor at the University of Illinois at Urbana-Champaign. He received his PhD from the University of California, San Diego in 2008 and his BA in mathematics and computer science from the University of Illinois at Urbana-Champaign in 2001. His research applies evidence-based techniques to a broad range of computer security domains, including e-crime and cyber-physical systems. klevchen@illinois.edu

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

As I read the Triton paper in the CSET '19 workshop [1], I found myself wanting to talk to some of the folks who had been working on this project. The recent software and documentation issues with Boeing's 737 Max that have led to the deaths of over 300 people provided some additional impetus. Karl Koscher, a member of the project, had written for *;login:* before about the automotive CAN bus [2], so I asked him for recommendations about who he thought I should talk to.

Karl suggested Kirill Levchenko. I don't recall ever meeting Kirill, but I'd certainly heard of him through various papers published by a large group of primarily West Coast researchers related to tracking Internet crime, work on hacking cars, and other topics [3].

As we worked on the interview, Kirill reminded me that the Triton Project was a collaboration of many people, something you can see right away by looking at the CSET paper [1]. Still, I found myself wanting to talk to Kirill, as I knew little about him beyond his published works.

Rik Farrow: Where did the idea for an avionics testbed come from?

Kirill Levchenko: Several years ago I and a group of researchers became interested in the security of electronic systems on aircraft (avionics). This was in part because of my own interest in aviation and in part because of the excellent work on automobile security done by my colleagues at UC San Diego and the University of Washington. Their experience with automobiles and my passion for aviation got us thinking about aircraft.

But unlike with the automobile work, we couldn't buy an airplane to test. So we had to focus on the parts of interest to us, the Line-Replaceable Units (LRUs) that might expose an electronic attack surface. We decided to start with the Communication Management Unit (CMU), which provides digital communications between aircraft and ground using a system called ACARS. To get this unit to work in the lab, we needed to recreate the environment it would have on board the aircraft, which meant building a testbed that would allow us to simulate parts of the aircraft—its communication networks and other LRUs.

RF: I'm hoping that avionics networks don't use TCP/IP. What do they use?

KL: The avionics of the aircraft we're looking at (everything designed before the Boeing 777, which includes the 737 and 747) uses the ARINC 429 bus, which transmits 32-bit frames at 12.5 kbps or 100 kbps.

ARINC 429 is unidirectional: there is one transmitter and one or more receivers. This provides some constraints on information flow. For example, the flight map in your in-flight entertainment system probably receives aircraft position information from the aircraft via ARINC 429, but, because ARINC 429 is unidirectional, the in-flight entertainment system cannot send any information back.

ARINC 429 is very similar to the CAN bus used in automobiles, with the notable difference that CAN is bi-directional—that is, there can be more than one transmitter on the bus.

Newer aircraft, such as the Boeing 787 and Airbus A380, use an Ethernet-based protocol called AFDX.

RF: Cars have telematics, usually via cellular networks, and Bluetooth. Both are connected to the CAN bus as vectors for attacks. What type of vectors are you considering for attacks against ARINC 429?

KL: Two of the most interesting vectors, from our point of view, are ACARS (handled by the CMU) and the software update process. These are the two vectors that originally motivated the testbed. ACARS (Aircraft Communications Addressing and Reporting System) provides short message digital communications between aircraft and ground systems. It was originally developed for reporting aircraft status (landed, at gate, etc.) but quickly came to be used for many other kinds of communications.

RF: Are updates to systems signed?

KL: Updates for most systems used on aircraft like the 737 and 747 are not signed.

Newer aircraft such as the Boeing 787 and A380 may use signed updates. This is something industry is working on.

There are no over-the-air updates while an aircraft is in the air, for obvious reasons. There are some products that allow you to do the update wirelessly when the aircraft is in for maintenance, but I am not aware of airlines using these. With the traditional ARINC 429-based data loader, there is no update verification built into the protocol. Of course, devices can implement their own checking, but we have not seen this in the avionics we've looked at.

RF: In Figure 2 in your CSET paper [1], you show USB 429 adapters. I assume these handle converting the serial protocol to USB, and the USB 429 driver portion is a Linux kernel module that acts as the device driver, so the emulated software you use as the bus will work, and daemons can present the 429 serial inputs or outputs as TCP ports?

KL: Yes, that's basically correct. To be specific, the driver is in userspace and connects to the USB 429 adapter using a library provided by the adapter vendor. The r429d daemon then provides access to the ARINC 429 bus over TCP (local access only). This allows us to create virtual devices that speak ARINC 429 to physical devices through the adapter.

RF: Was it hard to source the devices needed for the testbed?

KL: For aircraft such as the 737, no, we can get parts from avionics parts suppliers (who get them from aircraft that get torn down) and even from eBay. It's harder to get parts for newer aircraft, like the Boeing 777 and later, which don't have as large a parts market.

References

[1] S. Crow, B. Farinholt, B. Johannesmeyer, K. Koscher, S. Checkoway, S. Savage, A. Schulman, and A. C. Snoeren, and K. Levchenko, "Triton: A Software-Reconfigurable Federated Avionics Testbed," 12th USENIX Workshop on Cyber Security Experimentation and Test (CSET '19), USENIX Association, 2019: https://www.usenix.org/system/files/cset19-paper_crow.pdf.

[2] I. Foster and K. Koscher, "Exploring Controller Area Networks," *;login:*, vol. 40, no. 6 (December 2015): https://www.usenix.org/system/files/login/articles/login_dec15_02_foster.pdf.

[3] Published works of Kirill Levchenko: https://www.researchgate.net/scientific-contributions/70179178_Kirill_Levchenko.

Selected Results of the Workshop on Data Storage Research 2025

GEORGE AMVROSIADIS, ALI R. BUTT, VASILY TARASOV, EREZ ZADOK, AND MING ZHAO

George Amvrosiadis is an Assistant Research Professor of Electrical and Computer Engineering at Carnegie Mellon University, and a member of the Parallel Data Lab. His current research focuses on distributed storage and data analytics, with an emphasis on high performance computing and machine learning. He co-teaches courses on cloud computing and storage systems. gamvrosi@cmu.edu

Ali R. Butt is a Professor and Associate Department Head for Faculty Development in the Department of Computer Science at Virginia Tech. He was also an organizer for the NAE US Frontiers of Engineering in 2010. Ali's research interests are in distributed computing systems, cloud/edge computing, file and storage systems, I/O systems, and system support for deep learning. At Virginia Tech he leads the Distributed Systems and Storage Laboratory (DSSL). butta@cs.vt.edu

Vasily Tarasov is a Researcher at IBM Research—Almaden. His most recent research focuses on new approaches for providing storage as a service in containerized environments. His broad interests include system and storage design, implementation, and performance analysis. vtarasov@us.ibm.com

With the emergence of new computing paradigms (e.g., cloud and edge computing, big data, Internet of Things, deep learning) and new storage hardware (e.g., non-volatile memory, shingled-magnetic recording disks) a number of open challenges and research issues need to be addressed to ensure sustained storage systems efficacy and performance. The wide variety of applications demands that the fundamental design of storage systems should be revisited to support application-specific semantics. Existing standards and abstractions need to be reevaluated; new sustainable data representations need to be designed to efficiently support emerging applications. To take advantage of hardware advancements, new storage software designs are also necessary to maximize overall system efficiency and performance.

Therefore, there is an urgent need for a consolidated effort to identify and establish a vision for storage systems research and comprehensive techniques that provide practical solutions to the storage issues facing the information technology community. In May 2018, a National Science Foundation (NSF) Workshop on Data Storage Research 2025 took place at the IBM Research—Almaden campus in San Jose, CA [1]. This two-day community-visioning workshop identified research challenges in designing novel and innovative systems to store, manage, retrieve, and efficiently utilize unprecedented volumes of data at increasingly faster speeds. Thirty-three researchers participated in the discussions. Participants came from academia, industry, and government to represent multiple storage, I/O, and distributed systems research communities.

In-depth discussions were carried out at the workshop along four major themes: (1) storage for cloud, edge, and IoT systems; (2) AI and storage; (3) rethinking fundamentals of storage systems design; and (4) evolution of storage systems with emerging hardware. The participants underscored the need for focused educational and training activities to instill storage system expertise and interest in the next generation of researchers and IT practitioners. Finally, the development of shared, scalable, and flexible community infrastructure to enable and sustain innovative storage research and verifiable evaluation was also discussed. This article summarizes the discussions on the interaction of cloud and AI with storage. For more details, see the full workshop report [10].

Storage for Cloud, Edge, and IoT Systems

The advent of cloud computing has transformed the basic substrate for systems building in the last decade, and the long-anticipated "Internet of Things" (IoT) has led to the emergence of edge computing that extends system boundaries pervasively. In such a dynamic context, the depth of the storage stack and the scope of storage systems are increasing rapidly. Storage systems will need to manage data collected, stored, transformed, and transferred from heterogeneous edge devices to back-end cloud services, which can involve more than 18 layers [8]. Moreover, there are potential gaps or miscommunications between layers and components, which increase the difficulty of providing end-to-end guarantees and achieving the ideal tradeoffs among performance, reliability, fairness, etc. To move data storage research forward for cloud, edge, and IoT systems, we summarize the research challenges and opportunities into nine key properties that are essential for future storage systems.

Erez Zadok is a Professor of Computer Science at Stony Brook University, where he directs the File Systems and Storage Lab (FSL). He received his PhD in computer science from Columbia University in 2001. His current research interests include file systems and storage, operating/distributed systems, energy efficiency, performance and benchmarking, big data, and applied ML/AI. Zadok has published over 100 refereed papers, which have been cited over 7,000 times. ezk@cs.stnybrook.edu

Ming Zhao is an Associate Professor of Computer Science at Arizona State University (ASU) , where he directs the Research Laboratory for Virtualized Infrastructures, Systems, and Applications (VISA, http://visa.lab.asu.edu). His research is in the areas of experimental computer systems, including cloud/edge, big data, and high-performance systems as well as operating systems and storage in general. He is also interested in the interdisciplinary studies that bridge computer systems research with other domains. mingzhao@asu.edu

1. Efficient systems. Similar to traditional systems, cloud-based systems also put great focus on efficiency. This is important for both users who pay for usage and for cloud service providers who need to maximize profit. However, compared to traditional systems, there are many more layers involved in cloud systems: different layers usually require different data formats and read/write strategies to achieve the best local efficiency; these may conflict with other layers. Moreover, the diverse hardware, dynamic workloads, multitude of customer-facing cloud services, and inherent multitenancy make achieving high efficiency even more difficult. Finally, more effort must be expended on capabilities that support local and end-to-end quality of service (QoS). We can no longer focus on a single layer or component. Instead, cross-layer and end-to-end solutions are needed for removing all excess resource allocations in different layers, saving various costs (e.g., CPUs, memory, energy) and achieving an overall high efficiency.

2. Unified systems. Modern systems are filled with diverse storage options (e.g., file systems, SQL databases, key-value and object stores). While each individual storage option usually provides some unique features, they often have similar functions or components (e.g., managing persistent data structures or data replication). This inherent overlap of functionality is one of the major sources of inefficiencies in today's systems. We should explore the possibility of extracting the unified core components as building blocks and providing generalized solutions for various higher-level services. Also, to make different services more unifiable, we should experiment with solutions that can automatically transform configurations based on the dynamic needs of workloads: addressing the underlying representation of data, the amount of resources allocated, and adapting configurations of durability and replication parameters.

3. Specified systems. Current approaches to system building are too prescriptive, rigid, and error prone. This has led to various problems, including downtime and data loss, reducing future storage systems' scalability. We envision that future systems and applications should be specified in terms of performance requirements, data persistence needs, etc. Correctness properties should be precisely specified throughout the systems, which could potentially lead to the holy grail of verified systems that never lose data. There are several open research questions: how to specify properties for the opaque cloud, how to identify the necessary properties and interfaces for each layer or component in the system, and how to specify the dynamic requirements of workloads.

4. Elastic systems. Unlike traditional storage clusters that are built on fixed hardware resources, cloud-based systems are naturally elastic. Such systems can be broken into constituent components that can be scaled up/down independently based on current workload demands. We envision that system elasticity can be utilized for handling storage infrastructure tasks in addition to the workloads, likely improving overall system utilization and efficiency. To utilize storage elasticity, more desegregated, composable software architectures are highly desirable. Instead of today's monolithic storage and file systems, we should experiment with different building blocks and microservices, which can be reused across domains and improve elasticity, long-term reusability, etc.

5. Explainable systems. Current cloud-based systems are opaque to users. Many services use relatively simple interfaces, which makes it difficult for users to reason about the provenance and layout of their data. Moreover, due to the complicated layering within the cloud, it is also difficult for system builders or administrators to explain abnormalities in system behaviors. We envision future systems as providing detailed provenance information at a configurable verbosity level regarding, for example, how a data object was created, the number

of copies of it stored in the system, and who can access what and why. This will be helpful for improving security (e.g., how information is leaked), reliability (e.g., how data is corrupted), and performance (e.g., why this one run is slow).

6. Sharable systems. Unlike the first-generation cloud technologies that only run a single or a few applications for one entity, multitenancy is a new reality in modern cloud-based systems. We believe one fundamental requirement of multitenancy is effective sharing. However, achieving effective sharing is nontrivial as it involves many other systems aspects. For example, from efficiency's perspective, multitenancy may cause interference among different workloads at different system layers and thus violate QoS or service level objectives (SLOs). Similarly, security and privacy concerns need to be addressed in the multitenant environment to provide trustworthy sharing.

7. Application-driven systems. One major driving force of systems research is new application needs. There are many interesting new applications arising recently (e.g., augmented reality), which place new demands on storage systems (e.g., real-time processing). Given the diversity of applications, it is inefficient and impractical to build a highly specialized storage system for each application. Instead, we should explore the commonality among applications and automatically adapt storage systems for a range of applications. One unique challenge here is how to assemble a representative application suite and metrics for learning the common characteristics and demands.

8. Reliable systems. As the scale and complexity of systems keep increasing, failures become the norm rather than the exception. Therefore, we need to design systems to deliver high performance and other desired properties in the presence of failures. Future systems need to be formally specified, which could potentially lead to truly reliable storage that will never lose data. Existing efforts have shown that it is possible to formally specify and verify the crash consistency of one local file system built from scratch [6]. Nevertheless, it remains unclear how to scale formal methods to the vast majority of legacy software systems in cloud environments. More advanced mathematical methods and software engineering approaches are desirable.

9. Re-evaluable systems. A constant theme in storage research is the availability of suitable workloads. This is critical for fair comparisons between systems and for generating reproducible results. Unfortunately, compared to workloads for local storage systems (e.g., Filebench, SPEC-SFS), fewer cloud-based workloads are publicly available; a few such useful workload generators exist (e.g., YCSB [7], ATLAS [4]), but as systems keep evolving, more representative workloads are needed to advance research. Moreover, future storage systems should be built with easy evaluation in mind (e.g., exporting internal performance metrics) to facilitate the fair comparison of design tradeoffs under the same representative workloads.

Edge and Its Impact on Cloud

IoT is becoming a reality, causing an explosion of data collection, storage, and processing demands. The proliferation of IoT devices and the associated demands have led to the emergence of edge computing. Essentially, the edge model places a "mini datacenter" of compute and storage resources at the network edge, closer to end users. Compared with cloud computing, edge computing is less mature or standardized, and IoT devices can differ in capabilities, protocols, and data formats.

The service models for IoT applications are unclear. We envision that one possible direction is the serverless computing model, like AWS Lambda. However, additional research efforts are needed to integrate the spectrum of IoT devices into current models. Despite this heterogeneity, one common feature of all IoT devices is their limited hardware resources. To address this constraint, we should explore how to identify and discard unimportant data in a timely fashion—and how to balance among storage, preprocessing, and communication between IoT devices and clouds.

Cloud systems can be built for various workloads and adapt to demands on the fly. Conversely, edge computing has a large upfront cost to install edge nodes and a limited opportunity for ad hoc multiplexing at runtime, so we need to identify these workloads and match them to storage capabilities precisely.

One barrier to storage research in the era of cloud-edge computing is that no edge-to-cloud, holistic, persistent data storage capabilities exist today. Therefore, a realistic testbed involving both edge and cloud is highly desirable. Another barrier is the lack of agreed-upon workloads and traces for evaluation and comparison of new research designs. A realistic workload trace needs to track requests to read and write data across all devices, edge nodes, and cloud servers—including operations that transform or aggregate the data. Recent work on distributed system tracing [2] may provide the mechanism for collecting such traces; but the research community also needs to agree on a trace format, such as SNIA's DataSeries [5], and strategies for replaying such traces.

AI and Storage

Although AI and ML have existed as separate fields for decades, the last 5–10 years have witnessed an exponential growth in AI development and applications. Today, virtually all industries are either applying or planning to apply AI techniques. This shift is driven by three factors: data, compute, and algorithms. The confluence of these three factors has fueled AI's growth and, in turn, will drive the need for combined storage and AI research. Storage for AI focuses on how storage research that drives system designs can better serve AI workloads and data usage. Conversely, AI for Storage focuses on how storage systems can be improved via internal application of AI techniques.

Storage for AI

Storage technologies are likely to be more complex in the future to support growing needs of big data and AI workloads. This complexity will demand support for different APIs at different levels. We expect to continue to see healthy use of block-level, file-level (e.g., POSIX), object, and key-value stores—and likely combinations thereof. There is a need for high-level, easy-to-use APIs that hide much of the internal complexity from users and developers; conversely, there is also a need to allow advanced users to access lower-level APIs to enable more effective custom optimizations. The key to the design of future storage systems and their APIs would be that they must be easy to use and logical for AI application developers and *at the same time* provide optimal storage at the lower levels. Specifically, the emerging AI field presents five trends that intersect with storage, where targeted storage research can benefit AI uses and applications:

1. Massive data sets. AI workloads require the ingestion, pre-processing, and, ultimately, analysis of massive amounts of data. Multiple stages exist in typical AI pipelines, from data ingestion such as ETL (Extract, Transform, Load), to pre-processing (e.g., feature engineering), to the ultimate execution of an AI algorithm in its training or inference phases. All of these can benefit from storage optimizations for performance and data management.

2. Awareness of AI stages. Storage that is aware of the distinct stages or phases of AI processing can optimize AI pipelines via techniques such as caching of intermediate results, tracking of lineage, provenance, and checkpointing.

3. Compute architecture and data optimization. AI platforms typically follow distinct distributed computation architecture patterns (e.g., data and model parallel). Memory hierarchy and data layout design for such computation patterns should be a focus for future storage research. APIs that express the data access intent of an AI algorithm can be a powerful tool to integrate optimizations with AI computation.

4. Unique characteristics. AI algorithms have unique characteristics that can be exploited to create efficient storage designs. Example characteristics include tolerance of small amounts of data loss, highly structured access patterns, and the ability to use and extrapolate from lossy compression. Emerging access methods and characteristics associated with AI workloads, such as stream processing or edge storage, also create unique challenges.

5. Security, traceability, and compliance. The use of AI brings a new dimension to data security. As industries and users demand that decisions made by AI algorithms be reproducible, transparent, and explainable, pressure builds on enterprises to use data-management mechanisms to govern what data is collected and how it should be used to generate AI models and consequent insights.

AI for Storage

AI techniques should be researched to improve storage systems with respect to performance, reliability, availability, and QoS. This can be accomplished using the large and growing amount of available storage systems' historical access data. Insights can be gained from training and thus be used to help design or optimize storage systems in five ways:

1. Data placement optimizations. ML algorithms can be applied to predict popular data and application patterns, which help improve various storage techniques, including tiered caching, prefetching, and resource provisioning. Adapting caching policies through online learning can have significant benefits: using ML techniques to select between LRU and LFU replacement policies, for example, improved cache hit rates significantly under tighter memory constraints [9]. We believe that ML can be successfully applied for other performance optimizations.

2. Failure prediction. Failure or error patterns in large storage systems, such as disk failures and silent data corruptions, can be predicted using ML techniques and early detection; then, cautious measures can be taken to prevent errors from propagating. For example, proactively replacing disks that are predicted to fail soon can reduce the cost of data loss or rebuilding.

3. Storage tuning. Storage systems typically evolve to have a large number of tunable parameters. Parameters include hardware composition, I/O schedulers, tiering thresholds, cache sizes, etc. Using learning and other black-box optimization techniques can help administrators build and maintain storage systems under dynamic workloads, informing them on the optimal parameter values to improve system performance and lower cost for given workloads.

4. Change and anomaly detection. Part of tuning for workloads is understanding when they change phases. Anomaly detection has been an application area for ML techniques for over 20 years, and many techniques from these fields can translate easily to storage domains.

5. Intelligent storage devices. Storage devices capable of carrying out computation can help reduce maintenance overhead for the overall storage system, potentially improving performance. Such devices, however, require that we determine what level of intelligence is appropriate to offload to the device and propose storage techniques to achieve the best synergy.

The key challenge in using AI for Storage is that training data will often be limited before decisions have to be made. For instance, systems to store and quickly process data in self-driving cars must exist and run fast even *before* enough data can be collected for automated system design. Similarly, as storage needs shift over time in an organization, there may not be enough training data to predict how best to deal with changing

priorities when reconfiguring system parameters, tiers, data placement, and layout. Storage tuning may also be improved by considering more complex cost models, not just traditional throughput and latency: dollar cost, complexity, and power consumption are useful reward functions in a multi-objective optimization scheme.

Benchmarks and Workloads

Since AI techniques are heavily data dependent, any strategy for driving AI and storage research needs to factor in the need for publicly accessible data sets and benchmarks. Public data sets exist in ML but are in many cases too small to extract meaningful storage access patterns. Next, we describe three challenges that have to be overcome to drive expansive research into the storage and AI opportunities presented above:

1. Data-set generation and collection. We need some systematic and sustainable schemes to generate and collect data sets, including synthetic data generation of ML workloads, data sets from simulations and prior research, and long-term data collection and dissemination via shared community infrastructure.

2. Characterizing workloads across layers. How to benchmark and characterize workloads from different layers, including application, middleware, system, and storage-device layers, is challenging and needs investigation.

3. Workload classification. Classifying workloads has been studied for a long time [3]. As new storage platforms and applications are developed, there is a need to understand, in a way that is precise and communicable across different industries, what modern storage workloads look like. We could use ML techniques to improve workload characterization in four areas: (1) quantifying similarity among workloads; (2) tracking changes in how a workload functions on a given architecture; (3) learning mixes of customer workloads on shared storage systems; and (4) detecting phases of complex long-running workloads.

Conclusion

The NSF Workshop on Data Storage Research 2025 has unquestionably identified that the ongoing evolution of computing use cases, hardware technologies, and resource consumption patterns creates a multitude of new and complex challenges in data storage and management. We hope that this summary article and its associated, full-length public report [10] will serve as a useful guidance for data storage researchers in the coming years.

References

[1] National Science Foundation Visioning Workshop on Data Storage Research 2025: https://sites.google.com/vt.edu/data-storage-research/.

[2] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker, "The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*, November 2014, pp. 154–165.

[3] A. K. Agrawala, J. M. Mohr, and R. M. Bryant, "An Approach to the Workload Characterization Problem," *Computer*, vol. 9, no. 6 (1976), pp. 18–32.

[4] 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)* USENIX Association, 2018, pp. 533–546: https://www.usenix.org/system/files/conference/atc18/atc18-amvrosiadis.pdf.

[5] E. Anderson, M. Arlitt, C. Morrey, and A. Veitch, "DataSeries: An Efficient, Flexible, Data Format for Structured Serial Data," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 1 (January 2009).

[6] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, "Using Crash Hoare Logic for Certifying the FSCQ File System," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*, ACM, 2015, pp. 18–37.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Rakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, ACM, 2010, pp. 143–154.

[8] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu, "IOFlow: A Software-Defined Storage Architecture," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, ACM, 2013, pp. 182–196.

[9] G. Vietri, L. V. Rodriguez, W. A. Martinez, S. Lyons, J. Liu, R. Rangaswami, M. Zhao, and G. Narasimhan, "Driving Cache Replacement with ML-Based LeCaR," 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '18), USENIX Association, 2018.

[10] G. Amvrosiadis, A. R. Butt, V. Tarasov, E. Zadok, M. Zhao, "Data Storage Research Vision 2025: Report on NSF Visioning Workshop," 2018: https://dl.acm.org/citation.cfm?id=3316807.

Good Old-Fashioned Persistent Memory

TERENCE KELLY

Terence Kelly studied computer science at Princeton and the University of Michigan, earning his PhD at the latter in 2002. He then spent 14 years at Hewlett-Packard Laboratories. During his final five years at HPL, he developed software support for non-volatile memory. Kelly now teaches and evangelizes the persistent memory style of programming. His publications are listed at http://ai.eecs.umich.edu/~tpkelly/. tpkelly@eecs.umich.edu

Byte-addressable non-volatile memory (NVM)—Intel Optane—is now shipping in volume. Today's NVM offers performance between that of DRAM memory and flash storage [2, 7] and can be accessed via either storage or memory interfaces [8]. The latter offers the prospect of radically simplifying application software by allowing direct manipulation of persistent data via CPU instructions (LOAD and STORE), thus offering an alternative to traditional persistence technologies such as relational databases and key-value stores. Industrial adoption of NVM and its corresponding style of programming is growing [9].

Given the excitement surrounding novel NVM hardware, now is a good time to remind ourselves that it has long been possible to implement a software abstraction of persistent memory ("p-mem") on *conventional* hardware—ordinary volatile DRAM and block-addressed durable storage devices. The corresponding "p-mem style of programming" resembles the style that NVM invites, and supports similar simplifications, but doesn't require special NVM hardware.

This article illustrates p-mem programming on conventional hardware with C code for UNIX-like operating systems; all code is available at [3]. Spoiler alert: the basic technique is to lay out application data in memory-mapped files, with help from a few easy tricks and patterns. Because conventional `mmap()` doesn't guarantee data integrity in the face of failures, crash consistency requires extra support. The right crash consistency mechanism for p-mem programming on conventional hardware is *failure-atomic `msync()`* (FAMS) [6], and this article presents a concise new implementation of FAMS.

A Persistent Linked List

The C program below prepends words from `stdin` to a persistent singly linked list. It relies on a bare-bones persistent memory library, `pmem`, presented later. Notice that the list node data structure's `next` field is not a conventional pointer but rather an *offset*—specifically a `pmo_t` ("persistent memory offset type"), defined as a `uintptr_t` in `pmem.h`. Under the hood, `pmem` computes offsets relative to the base address where persistent data are mapped, which may vary on different runs of the program. Offsets allow data structures to be *relocatable*, which improves portability and facilitates sharing persistent data between different applications. The alternative of *non*-relocatable persistent data offers different tradeoffs and is beyond the scope of this article; see [5] for a discussion.

```
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include "pmem.h"

typedef struct {
  pmo_t next;
  char string[];
} node_t;

#define NP(o) ((node_t *)pmem_o2p(o))
```

```
int main(int argc, char *argv[]) {
  int r;
  char buf[100];  /* harmonize with scanf() below */
  pmo_t head, t;
  if (2 != argc) {
    fprintf(stderr, "usage: %s pmemfile\n", argv[0]);
    return 1;
  }
  if (0 != (r = pmem_map(argv[1]))) {
    fprintf(stderr, "pmem_map() failed: %d\n", r);
    return 2;
  }
  head = pmem_get_root();
  while  (1 == scanf(" %99s", buf)) {
    if (0 == strcmp("[dump]", buf))
      for (t = head; 0 != t; t = NP(t)->next)
      printf("%s\n", NP(t)->string);
    else {
      t = pmem_alloc(sizeof(node_t) + 1 + strlen(buf));
      if (0 == t) {
       fprintf(stderr, "pmem_alloc() failed\n");
       return 3;
      }
      strcpy(NP(t)->string, buf);
      NP(t)->next = head;
      head = t;
      pmem_set_root(head);
    }
  }
  return 0;
}
```

Function `pmem_map()` maps a given persistent data file into memory, initializing persistent heap metadata within the file if necessary. Unlike conventional `mmap()`, `pmem_map()` returns an error code rather than the address where the file has been mapped. Clients of `pmem` (i.e., code that uses `pmem`) neither know nor care about persistent data addresses—that's the whole point of relocatability. Clients allocate from a persistent heap in the file via `pmem_alloc()`, which returns offsets rather than conventional `malloc()`'s pointers. Finally, the `pmem` library embeds a *root offset* within the persistent data file. Clients must ensure that all persistent data are reachable from the root by calling `pmem_set_root()`. This allows the client to obtain an entry point into persistent data structures via `pmem_get_root()` on subsequent executions. Our list example program maintains the invariant that the root offset is always the head of the persistent linked list.

Function `pmem_o2p()` converts offsets to conventional pointers, which macro `NP` casts to a list-node pointer. Clients of `pmem` need offset-to-pointer conversions for accessing the innards of application-defined data structures. However, the `pmem` library doesn't support *pointer-to-offset* conversions because well-designed applications don't need them: clients' persistent data structures should contain only offsets returned by `pmem_alloc()` (or offsets derived therefrom), never pointers; only offsets are encountered when traversing persistent data.

The shell commands below demonstrate that our program's list is indeed persistent. `truncate` creates a new sparse backing file whose size is a multiple of the system page size. We run `list` twice, feeding it different words and dumping the list. The second dump shows that the words entered on the first run have persisted.

```
% truncate -s 409600 list.bf
% echo 'wun too [dump]' | ./list list.bf
too
wun
% echo 'free fore [dump]' | ./list list.bf
fore
free
too
wun
```

Persistent memory programming based on memory-mapped files is much more versatile and powerful than the brief examples above would suggest. In particular, retrofitting persistence onto legacy software that was not designed for persistence can be remarkably easy, and the rules governing multithreaded p-mem are straightforward [5].

Library Internals

The `pmem` library interfaces used above admit a succinct no-frills implementation, shown below. There's nothing arcane going on; much of the code simply checks internal consistency and catches corner-case errors, syscall failures, and client misuse. The library often returns line numbers where errors occur rather than `errno`-like codes ("use the Source, Luke"), and the persistent heap supports a p-mem allocator but no corresponding `free()`.

The `pmem` library defines a header structure (`pmh_s`) that will occupy the first few machine words of the backing file that contains persistent data. The header contains allocator bookkeeping information and the root offset described above. The library stores in static external variables `e_base` and `e_len`, respectively, the address at which the backing file is mapped and the size of the backing file.

Library function `pmem_map()` invokes conventional `mmap()` to map a specified backing file into the caller's address space; at most one such mapping at a time is supported. Function `pmem_unmap()` removes mappings; `pmem_alloc()` allocates persistent memory; and the paired `pmem_[get|set]_root()` functions provide access to the root offset.

Again, the pmem library is intentionally Spartan. It serves merely to remind us that a few dozen lines of code suffice to support rudimentary persistent memory programming on conventional hardware.

```
#include <assert.h>
#include <fcntl.h>
#include <stdint.h>
#include <stddef.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include "pmem.h"

static_assert(sizeof(pmo_t) == sizeof(void *),   /* C11 */
              "offsets & pointers incompatible");

typedef struct {  /* header of backing file & in-memory image */
  pmo_t avail, end,  /* allocator bookkeeping */
        root;        /* live data must be reachable from root */
} pmh_s;  /* "persistent memory header structure" */

static pmh_s * e_base;  /* start address of in-memory image */
static size_t  e_len;     /* length of in-memory image */

#define UNIT (_Alignof(max_align_t))  /* C11 */
#define ALIGNED(o) (0 == (o) % UNIT)
#define ALIGN(o) do { while(! ALIGNED(o)) (o)++; } while (0)

/* Backing file and its in-memory image consist of a header (of
   type pmh_s above), a heap (nearly everything else), and final
   padding (one UNIT). Padding at the high end eliminates an
   awkward corner case. A root offset must "point" within the
   heap. Other offsets (e.g., "end") may point one byte beyond
   heap, analogous to C rule for pointers (N1570 Sec 6.5.6). */
#define VALID(o)  \
  (0 == (o) || (sizeof *e_base <= (o) && (o) <= e_len - UNIT))
#define VALID_ROOT(o) \
  (0 == (o) || (sizeof *e_base <= (o) && (o) <  e_len - UNIT))

#define SANITY_CHECKS    \
  do {                   \
    assert((NULL == e_base && 0 == e_len) ||                    \
           (NULL != e_base && 0 != e_len) );                    \
    assert(NULL == e_base ||                                    \
           (  ALIGNED(e_base->avail) && ALIGNED(e_base->end)    \
            && VALID(e_base->avail) &&   VALID(e_base->end)     \
            && VALID_ROOT(e_base->root) ));                     \
  } while (0)

void * pmem_o2p(pmo_t o) {  /* convert offset to pointer */
  assert(VALID(o));
  return 0 == o ? NULL : (char *)e_base + o;
}

#define P20(p) ((pmo_t)((char *)(p) - (char *)e_base))

#define RL return __LINE__  /* indicates where error occurs */

int pmem_map(const char * const file) {
  int fd, prot = PROT_READ | PROT_WRITE, flag = MAP_SHARED;
  long int pgsz;  struct stat sb;  size_t s;  pmh_s *t;
  SANITY_CHECKS;
  if (NULL != e_base)  /* limit: one mapping at a time */      RL;
  if (1 > (pgsz = sysconf(_SC_PAGESIZE)))                      RL;
  if (UNIT > (size_t)pgsz)                                     RL;
  if (0 > (fd = open(file, O_RDWR)))                           RL;
  if (0 != fstat(fd, &sb))                                     RL;
  if (10 * UNIT + sizeof *t > (s = (size_t)sb.st_size))        RL;
  if (0 != s % (unsigned long)pgsz)                            RL;
  if (MAP_FAILED ==
      (t = (pmh_s *)mmap(NULL, s, prot, flag, fd, 0)))      {
    if (0 != close(fd))  /* don't leak fds ... */              RL;
    else                                                   RL; }
  if (0 != close(fd))                                     {
    if (0 != munmap(t, s))  /* ... or memory either */        RL;
    else                                                   RL; }
  /* file must be either new or already initialized: */
  if ( ! (   (0 == t->avail && 0 == t->end && 0 == t->root)
          || (0 != t->avail && 0 != t->end)))                  RL;
  if (! (ALIGNED(t->avail) && ALIGNED(t->end)))                RL;
  e_base = t;
  e_len  = s;
  if (! (VALID(t->avail) && VALID(t->end)
         && VALID_ROOT(t->root)))                              RL;
  if (0 == t->avail) {  /* initialize persistent heap */
    t->avail = P20(1 + t);
    ALIGN(t->avail);
    t->end   = P20((char *)t + s - UNIT);
    t->root  = 0;
  }
  else  /* previously initialized; check size: */
    if (t->end != P20((char *)t + s - UNIT))                   RL;
  SANITY_CHECKS;
  return 0;
}

pmo_t pmem_alloc(size_t n) {  /* "bump-pointer" allocator */
  pmo_t r;
  SANITY_CHECKS;
  assert(NULL != e_base);
  if (0 == n                                || /* ask 0, get 0 */
      e_base->avail    >= e_base->end       || /* out of p-mem */
      e_base->avail    >  ~(pmo_t)0 - n     || /* "+n" overflows */
      e_base->avail + n >  e_base->end)       /* <n bytes left */
    return 0;
  r = e_base->avail;
  e_base->avail += n;
```

```
  ALIGN(e_base->avail);
  SANITY_CHECKS;
  return r;
}

int pmem_unmap(void) {
  SANITY_CHECKS;
  if (NULL == e_base)             RL;
  if (0 != munmap(e_base, e_len))  RL;
  e_base = NULL;
  e_len = 0;
  return 0;
}

void pmem_set_root(pmo_t o) {
  SANITY_CHECKS;
  assert(NULL != e_base && VALID_ROOT(o));
  e_base->root = o;
}

pmo_t pmem_get_root(void) {
  SANITY_CHECKS;
  assert(NULL != e_base);
  return e_base->root;
}
```

Crashes and Data Integrity

Could a full-featured incarnation of the `pmem` library be suitable for serious purposes? Yes, for applications that always perform an orderly shutdown. However, `pmem` is inadequate for applications that must tolerate sudden crashes, e.g., power outages, OS kernel panics, and application software crashes. Why? Because `pmem` creates shared file-backed memory mappings with conventional `mmap()`, which cannot prevent crashes from corrupting the backing file. One fundamental problem is that the OS may write modified memory pages down to the backing file at any time and in any order, regardless of if/when `msync()` is called. Another problem is that if `msync()` *is* called, the changes it makes to the backing file are not atomic with respect to failure. The state of the backing file following a crash is therefore indeterminate.

Failure-atomic `msync()` (FAMS) solves this problem by strengthening the semantics of conventional `mmap()` / `msync()`. FAMS guarantees that the backing file always reflects the most recent successful `msync()`, regardless of failures [6]. The FAMS abstraction is the ideal foundation for crash-tolerant persistent memory programming on conventional hardware. It has been implemented in the Linux kernel, in file systems, and in user-space libraries; at least six FAMS implementations exist, two of which are in commercial products [5]. FAMS has the attractive property that underlying durable storage is a freely configurable placeholder: "durability" for a FAMS-based p-mem program can mean anything from a single hard disk to a RAID array or geo-replicated cloud storage. Furthermore, FAMS is easy to reason about because it merely *restricts* the behavior of well-understood standard interfaces: FAMS guarantees behavior that is possible (but, sadly, unlikely) in conventional `mmap()` / `msync()`.

Existing FAMS implementations have demonstrated the abstraction's power and versatility, but they're not without barriers to adoption: some are research prototypes, others are buried in appliance-like commercial products, and the two newest implementations are complex but not yet thoroughly tested [4, 5]. The world needs a FAMS implementation that is efficient enough for serious use yet simple enough to audit easily.

Simple and Efficient Crash Consistency

Our efficient yet very simple new userspace implementation of failure-atomic `msync()` makes two compromises: it restricts our choice of file system, and its interface is fussier than classic FAMS.

The library implementation below is called `famus_snap` ("failure-atomic `msync()` in userspace via snapshots"). It runs on file systems that allow multiple files to share physical storage, e.g., Btrfs, XFS, and OCFS2 (optionally accessed over a network via NFSv4.2 or CIFS). The interesting work happens in function `famus_snap_sync()`, which uses `ioctl(FICLONE)` to create a new snapshot that shares storage with the backing file. A copy-on-write mechanism ensures that subsequent modifications to one file do not affect the other.

The interfaces of both the `mmap()` and `msync()` analogs require the caller to supply a file descriptor for an empty write-only snapshot file. When these functions return successfully, the snapshot file contains the current state of the backing file and is read-only. Whereas post-crash recovery in classic FAMS uses the backing file, recovery in `famus_snap` *replaces* the backing file with the most recent readable snapshot file. As a side effect, `famus_snap` gives us data versioning for free: every snapshot is a version of the backing file, which may be retained indefinitely or deleted to reclaim storage resources.

```
#define _POSIX_C_SOURCE 200809L

#include <stddef.h>
#include <unistd.h>
#include <linux/fs.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include "famus_snap.h"

static int rwperm(mode_t m, unsigned int r, unsigned int w) {
  return (!!(m & S_IRUSR) == r) && (!!(m & S_IWUSR) == w);
}
```

```
#define  L __LINE__
#define RL return L  /* indicates where error occurs */

/* We must fsync() backing file twice to ensure that snapshot
   data are durable before success indicator (file permission)
   becomes durable. We're not using fallocate() to reserve
   space for worst-case scenario in which backing file and
   snapshot file diverge completely, because that could defeat
   the reflink sharing that makes snapshots efficient; read
   "man ioctl_ficlone". The "ioctl(FICLONE)" works only on
   reflink-enabled file systems, e.g., Btrfs, XFS, OCFS2. */
int famus_snap_sync(fd_t bfd, fd_t snapfd, fd_t dirfd) {
  struct stat sb;
  if (0 != fstat(snapfd, &sb))                  RL;
  if (! rwperm(sb.st_mode, 0, 1))               RL;
  if (0 != ioctl(snapfd, FICLONE, bfd))         RL;
  if (0 != fsync(snapfd))                       RL;
  if (0 != fchmod(snapfd, S_IRUSR))             RL;
  if (0 != fstat(snapfd, &sb))                  RL;  /* paranoia */
  if (! rwperm(sb.st_mode, 1, 0))               RL;  /* paranoia */
  if (0 != fsync(snapfd))                       RL;
  if (0 < dirfd && 0 != fsync(dirfd))           RL;
  if (0 != close(snapfd))                       RL;
  return 0;
}

#define RN return NULL

void * famus_snap_map(void * addr, size_t * plen, int flags,
                      fd_t bfd, fd_t snapfd, fd_t dirfd,
                      int * status) {
  struct stat sb;  void *a;  int prot = PROT_READ | PROT_WRITE;
  if (NULL == status)               {             RN; }
  if (NULL == plen)                 { *status = L; RN; }
  if (0 == (flags & MAP_SHARED))    { *status = L; RN; }
  if (0 != fstat(bfd, &sb))         { *status = L; RN; }
  *plen = (size_t)sb.st_size;
  a = mmap(addr, *plen, prot, flags, bfd, 0);
  if (MAP_FAILED == a)    { *status = L; RN; }
  if (NULL == a) {
    if (0 != munmap(a, *plen))        *status = L;
    else                              *status = L;
    RN;
  }
  if (0 != (*status = famus_snap_sync(bfd, snapfd, dirfd))) {
    if (0 != munmap(a, *plen))        *status = L;
    RN;
  }
  return a;
}
```

The full source code for `famus_snap` is available at [3]. It requires a reflink-capable file system such as Btrfs, XFS, or OCFS2. If you're eager to run `famus_snap` but you don't have such a file system handy, consider installing one *within a file* on some other file system; just run the following commands as `root`:

```
# truncate --size 512m XFSfile
# mkfs.xfs -m crc=1 -m reflink=1 XFSfile
# mkdir XFSmountpoint
# mount -o loop XFSfile XFSmountpoint
# xfs_info XFSmountpoint
# cd XFSmountpoint
  [run famus_snap test...]
```

Streamlined Implementation

The `famus_snap` library above is a reasonably efficient way to implement failure-atomic `msync()` in userspace. However, with an in-kernel implementation like the prototype posted by Christoph Hellwig [1], similar semantics can be implemented more efficiently by taking advantage of the mechanisms that the XFS file system uses to implement the reflink system call.

In that case the existing code path to allocate new blocks and write them out of place when overwriting data is used independently of the B-tree tracking reference counts for blocks shared after using the reflink system call. In this case in addition to the actual block allocation, only the special records that ensure that the blocks are cleaned up when recovering from an unclean shutdown are required. This ensures the overhead of the write is similar to that for extending a file or filling a hole, but the extra overhead for manipulating block reference counts is avoided.

Conclusion

Persistent memory programming on conventional hardware is possible, thanks to `mmap()` and a few tricks that don't get as much attention as they deserve. Regardless of whether conventional hardware or newfangled NVM is available, the great advantage of the p-mem style of programming is simplicity—readers skeptical on this point are invited to re-write the persistent linked list program above using, e.g., a relational database or key-value store for persistence.

For crash-tolerant applications, failure-atomic `msync()` provides precisely the right fortified semantics for `mmap()`-based p-mem programming. The new FAMS implementation presented in this article is concise, clear, and thus easy for readers to audit because it leverages efficient file snapshotting from userspace. Christoph Hellwig's implementation in XFS achieves greater efficiency by avoiding unnecessary work. Until NVM supplants DRAM, FAMS can support crash-safe p-mem programming on conventional hardware.

Acknowledgments
Christoph Hellwig reviewed the snapshot-based implementation of failure-atomic `msync()`, suggested the procedure for creating a quick XFS installation within a different file system, provided a description of his FAMS implementation for XFS, and supplied information about reflink-capable file systems.

References

[1] C. Hellwig, "Failure Atomic Writes for File Systems and Block Devices": https://lwn.net/Articles/715918/.

[2] J. Izraelevitz, J. Yang, L. Zhang, A. Memaripour, Y. J. Soh, S. R. Dulloor, J. Zhao, J. Kim, X. Liu, Z. Wang, Y. Xu, S. Swanson, "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module," April 2019: https://arxiv.org/abs/1903.05714v1.pdf.

[3] T. Kelly, Example code to accompany this article: https://www.usenix.org/sites/default/files/kelly_code.tgz.

[4] T. Kelly, "`famus`: Failure-Atomic `msync()` in User Space": http://web.eecs.umich.edu/~tpkelly/famus/.

[5] T. Kelly, "Persistent Memory Programming on Conventional Hardware," *ACM Queue*, vol. 17, no. 4, July/August 2019: https://queue.acm.org/detail.cfm?id=3358957.

[6] S. Park, T. Kelly, K. Shen, "Failure-Atomic `msync()`," in *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*, pp. 225–238: https://dl.acm.org/citation.cfm?id=2465374.

[7] I. B. Peng, M. B. Gokhale, E. W. Green, "System Evaluation of the Intel Optane Byte-addressable NVM," International Symposium on Memory Systems (MemSys), Sept. 2019: https://memsys.io/.

[8] A. Rudoff, "Persistent Memory Programming," *;login:*, vol. 42, no. 2, Summer 2017: https://www.usenix.org/system/files/login/articles/login_summer17_07_rudoff.pdf.

[9] S. Swanson (organizer), Persistent Programming in Real Life (PIRL) [conference], 2019: https://pirl.nvsl.io/.

Ask-Me-Anything Engineering

EFFIE MOUZELI

Effie Mouzeli studied physics and distributed scientific computing but didn't turn out to be a physicist or a scientific computer scientist. She has worked as a systems engineer/SRE at a number of startups and small organizations (most of which are not with us anymore), where her responsibilities were usually automation, infrastructure architecture, and working closely with developers. Currently, she is on the SRE team that takes care of Wikipedia and its sister projects at the Wikimedia Foundation. emouzeli@runbox.no

Small organizations are the reality for a number of people doing operations. Despite that, there are limited resources on the subject of working in a systems team of a few engineers. On the contrary, there is literature on how large organizations implement SRE, how they got to 99.999% availability, and how to process millions of metrics per second. Those are really good and interesting reads, but I have been in the shoes of a person reading such articles and thinking, "I enjoyed reading this, but I can't use it." For the purpose of this article, the terms "small organizations/companies" but also "small-scale" will be used to describe organizations and startups where there is a single SRE or a small team of SREs.

Everything was just a few servers once, and everything started from something. Let's just think about how much we rely on products by small companies like local news sites or local ferry booking services. Moreover, we mustn't forget that some of us live in cities and countries where there are no large engineering teams, and those environments are the only available places to work. And those are just a handful of reasons why small-scale companies matter.

There are a few major challenges that a systems engineer at a small organization will have to constantly deal with:

- Paying off someone else's technical debt, almost alone
- Managing a live infrastructure, almost alone
- Caring for the development teams, almost alone

I will try to provide an overview of what it is like working in small-scale environments, what to expect, how SRE concepts can be beneficial, and what can be learned.

The Role of an A.M.A. Engineer

In small software companies, I always considered the systems team as setting their tempo to the development team. The development team needs that expertise so as to, in turn, set the standards for the organization. For example, if bootstrapping a new server takes a day, everyone will consider this normal and will never demand to have a new server ready in an hour. If pushing code to production requires 10 manual steps, the development team will never be able to deliver faster. It is up to the SRE to automate manual steps if possible.

Another important aspect of this role is to be the facilitator between development and infrastructure. It is up to the systems engineer to make the infrastructure accessible to developers, show them where the controls are, and be there for them. But beyond that, people come to us for answers when things don't add up. An Ask-Me-Anything engineer can be the expert on DNS, databases, networking, Linux, and monitoring. One can expect to receive a broad range of inquiries, from "My service can't access the database" to "My dad wants to buy a new laptop." For me, I believe the most bothersome question was, "What's the guest WiFi password?" That is, until I taped QR codes on every office wall.

In addition, our experience will help us to plan wisely for the future. We should be able to see a few steps ahead and have a plan about how the systems will grow along with the organization.

It is not unusual, though, for systems engineers not to get the credit they deserve. While large organizations go a long way to achieve "five nines," that is not always the case in small ones. Keeping the developers happy and productive while having few incidents does not translate financially, for example, like a new feature would. As Heidi Waterhouse has said, "No one remembers the crisis averted" [1]. Sadly, good systems engineering does not have a direct Return on Investment.

Small-Scale Technical Debt

Technical debt is every organization's Achilles' heel, regardless of size. There is almost always a mountain of it. Even at startups, there is rarely a dedicated person for operations in the beginning, meaning that the multi-hat engineering team is trying to make systems work and make technical decisions to the best of their knowledge.

Identifying what is generating technical debt is a very good start towards slowing it down. It is impossible to come up with a complete list of reasons behind technical debt in systems at small environments, but I can name a few from my experience.

Lack of Processes, Documentation, and History

When a few people are developing and running a product, it is normal to solve problems ad hoc. Minor and major problems are dealt with as they come up, and then are forever forgotten. **There are no runbooks, no postmortems, no histories.** But those issues derive from deeper ones: there are no standard processes for how to do things, e.g., introducing a new service to production, but also no documentation as to how things work. Especially in fast-paced environments like startups, we subconsciously consider documentation as a waste of time.

Cargo Culting

New and immature technologies are adopted under the false assumption that they can fix anything and that by using them, the organization can stay up-to-date and relevant. Together with the lack of appropriate systems background or experience, it just equals technical debt. Not to mention that sometimes, we go as far as fitting our problems into the solution we want to experiment with. Without drawing any lines, or putting any limits, this can lead to a Frankenstein infrastructure. Some notable examples are Kubernetes and Docker; they provide solutions, but how many times did a team with limited resources ask whether it had the human capital to go in this direction?

Short-Term Planning

Cargo culting itself is a symptom of another underlying problem: the culture of short-term planning. Everything moves so much faster when the development team is 20 people rather than 100 or 200. This team of 20 people has a list of features to add to the product, which in turn have a list of requirements. Still, there is no vision as to how the systems themselves should look a year after those changes. A simple example would be, "We estimate that if we market this new feature, we will gain 20% more clients within six months." That is great, but have we done any capacity planning to handle that traffic?

Waiting for a Hero

This does not generate technical debt per se, but it is a consequence of all the above. At some point the organization has so much duct tape and WD-40 that it simply waits for someone to make sense of the chaos and save it. How chaotic this chaos is depends on a number of factors. If this is a startup, the chaos is proportional to how late to the party a systems engineer has arrived. On a brighter note, at a startup it is highly likely one can talk to the people who created all that debt and get answers. If this is a long-running company, it depends on how many systems engineers have come and gone over the years as well as how many people assumed that role.

The Five Stages of Technical Debt

Let's assume that we have joined an organization as the first SRE. Our hypothetical new startup, **everythingsocks.io**, has 30 servers, 50k customers, and about 25 developers, all using the same account, *root*. Funding is secured, business is booming, and the future looks bright!

We arrive in a new fast-paced environment where we don't know anyone, we are required to run a live infrastructure we have never seen before, and we have no idea what is coming. One thing is certain though: we are going to go through the five stages of technical debt [2].

1. Denial

The product looks functional, as well as its systems, and we reckon our job is to initially keep everything running and then move forward. All we have to do is hold the wheel and drive. EverythingSocks looks like an awesome place to work after all. They have free lunches, a pool table, and free yoga lessons!

2. Anger

While we are sure that everything is going great, we begin to get interrupted. A developer reports they think the auth server is overloaded. They believe an additional server might help. Another one pops by, saying that they are getting some 500s, just like last week. A third one appears complaining they are unable

to access StackOverflow. It goes without saying that everything is urgent. We realize that we are smiling because we have no idea what is going on. We are frustrated.

3. Bargaining

This is the part where we believe that we must gain control. We are trying to show some faith in our abilities and not get too overwhelmed, even though everything is on fire. Our colleagues seem to be good people after all; we can make it work!

4. Depression

At this point we are desperate for information. We are running around trying to figure out what's under the ground we are standing on. The more we uncover, the more we feel despondent. We find out that everyone has root access to the databases, even the microservices, and the main application is logging plaintext passwords. This is depressing.

5. Acceptance

We get to acceptance when we finally see some light at the end of the tunnel. We have a better overview, we are feeling optimistic, and we have a plan of how to make it better.

Getting Control

With all these problems discovered, as a whole, there is a mountain to climb. The problems have to be broken down into pieces and prioritized. The development team and our intuition can help us get there.

Asking a Lot of Questions

But in addition to asking questions it's crucial to ask the right questions. For example, ask the development team what *they* need, what are their daily pain points, and what they believe should be improved. The team might request a proper staging environment but fail to mention that they manually delete application logs every week. Read between the lines!

Understanding the Product

A bad habit I have noticed among systems engineers is that they tend to distance themselves from knowing how the applications they are managing work. This is a mistake that can turn many incidents into a wild goose chase. How subsystems communicate with each other, what they are doing, as well as what external dependencies they have, should be something an SRE is aware of. For instance, if a payment provider is taking longer to respond, it might exhaust the application workers. If you don't know that one of the apps depends on a payment provider's response time, you will find out the hard way, through reading logs, stracing, tcpdumping, etc., while everything is on fire.

Documentation Is Bliss

As you are gathering information about literally everything, write it down: what you learn, what you think is missing, what needs improvement. Your ultimate goal is to eventually have a board with the work that needs to be done. You will feel hopeful when there is finally a comprehensive list of tasks that include immediate and future needs. This is how one gets to Acceptance!

Understand Your Limits

I strongly believe that a good engineer is able to understand what they can do under certain circumstances. Try not to rush. Do not start making promises that "it will take two days." If your work is fast but sketchy, it will keep coming back to haunt you, and that does not scale well. Equally important, having nothing delivered on time can become part of the culture.

When it comes to introducing new tools to help you in your day-to-day tasks, start with the familiar ones. You will find time later to try something new and fancier, together with researching. And if what you are researching is not working out, learn to let go and move on. The more time you spend on one front, the more everything else is falling behind. I was in a team, a newly formed team, that kept promising to migrate the infrastructure from one datacenter to another within three months, while migrating our servers from bare metal to VMs, while migrating from Chef to SaltStack, while production was running. What could possibly go wrong?

Consistency

Creating standard processes and rules, and then sticking to them and defending them, is your true ally: for example, processes about new server requests, new applications, new users, rules that all microservices should be managed by `systemd`, and all packages must be installed via configuration management. You need to keep snowflakes to a bare minimum as much as you need your sanity.

The Big Picture

I will lay down the components a functional infrastructure needs in order to be manageable. No matter how many servers we have or how much traffic we serve, we need all of them. The problem in small-scale is all will be implemented by a single person or a tiny team. That is the beauty and the difficulty of small-scale. The strategy here is divide and rule. Attacking all of them at the same time can be chaotic and stressful, so iterate, little by little!

Automation and Provisioning

Try to manually create a staging environment and document all steps; it will help you learn more about the product. Next time you revisit it, write scripts for that. Later on, have Jenkins run those scripts. Having Jenkins running silly bash scripts is better

than you running them. In the same fashion, decide how you will provision. Be it bare metal servers, virtual machines, or container images, provisioning must be automated. Infrastructure as code is the best way.

Observability

Based on current needs, decide on monitoring and alerting. If real-time monitoring looks like a lot right now, choose a more simple solution. You will know when a more sophisticated solution is needed. What about metrics? Which metrics can be used? Which metrics can be pulled from logs? Which are of business value, and which can be used as health indicators? Talk to the development team and figure it out. Together. For instance, low sock sales is a business metric that can imply systems issues.

Updating and Viable Backups

Keeping software and tools up-to-date while in a tiny team is very, very challenging. It is up to you to judge when and what should be updated given the time you have available. Viable backups are our dirty secret, and there are countless horror stories to prove it. We just strap them on flying unicorns and never look back. Even the most pessimistic personalities, when it comes to systems, hope for the best-case scenario: they won't have to use their backups. In other words, figure out what needs to be backed up and test those backups. You need the confidence and security that you can rely on them.

Security

Common practices like limiting access unless needed (e.g., to databases), using different database passwords in production and staging, keeping track with security updates, etc., are a very good start. Taking this lightly can lead to those upsetting stories about junior engineers deleting the production database on their first day.

Emergency Response

You are always on call. Sit with the development team and discuss what can be done in possible scenarios, along with some emergency checklists, and eventually create some early runbooks.

Legacy Systems

This is more common in existing companies than startups: very essential services running somewhere, but no one knows anything more about them. And when they break, you will be called on to fix them. For your own peace of mind, work with your colleagues on ways to remove those black boxes.

The aim is to bring the systems to a manageable state while assisting the development team with their deliverables. Balance comes through small iterations, improving what you have in each one. Don't rush; **let your systems mature.**

Building Habits and Culture

It is not uncommon in software companies for developers to dislike working with systems engineers, and vice versa. This is usually due to a lack of communication and bad attitude from all quarters. Certainly in a place with a single SRE, this is not going to work to anyone's benefit. Being approachable is key in building trust between you and the development team. After all, they are ***your*** team as well. Try to have standard meetings with each other, and share what is in the roadmap. Guess who will have to work extra hours if you are playing with alerting while developers are about to roll out a feature that needs a new dedicated database server. It is important to learn to meet each other halfway.

Furthermore, many developers are not proficient in systems engineering, and that is generally accepted in small companies. Instead of being frustrated for being asked for the 10th time, "How do I restart a service?" teach them and document it! Help them become better. Help them learn how to use what you are building. After leading a systems crash course with 15 developers, I cried with joy the first time a colleague used `ngrep` to debug an issue. Generally, the more self-serviced the development team is, the less toil for you.

Lastly, being arrogant is something that you can't afford. If people prefer running around production with scissors because they just don't want to deal with the SRE, then you're holding a time bomb. A few years ago, a group of developers wanted to experiment with Docker, but my team resisted even running rough tests with Docker. Eventually, the developers set up a staging environment using Docker at a cloud provider outside of our infrastructure. This meant that our proprietary code was deployed somewhere outside the control of the systems team. But who is to blame here?

It Takes a Village

Your efforts will go as far as management and the development team want to go. Same goes for if what you have started will turn into a team or teams in the future. If there is not enough management buy-in, there is an upper limit to what can change and improve. You may want to introduce service level objectives [3] so that, in turn, you can add meaningful alerts. This can't be done without developer assistance. In addition, it is impossible to have blameless postmortems if nobody wants to write them and if people prefer to point fingers at each other.

There are chances that one may really try to push for changes and not get the desired results. Aristotle wrote "one swallow does not make a spring" [4], and it is true. A person alone might not be enough if the rest of the team won't listen, and in my personal opinion, that is not a problem an SRE should be solving.

Small companies are intimate, and they provide us with that feeling of belonging, but they can also feel a little lonely, especially if one is flying solo. What helped me over the years was keeping in touch with other people doing the same job, through meetups, group gatherings, and chats. It is essential to be able to share the tech and non-tech problems at work with people who understand.

Is It Worth It?

This is a coming-of-age experience for an engineer. There are no safety nets and no one you can pass the ball to. Every decision will be more well thought out since any consequences will directly affect you and your peers. Moreover, when money is tight and time is limited, one gets creative. You work with what is in front of you; you can't simply add 100 servers or you can't waste days trying to find what is wrong.

You will acquire a really broad skill set. It will range from debugging tools, networking, databases, and programming, to project management, human resources, people skills, event planning (true story), and, possibly, how to help a colleague's dad find his dream laptop.

Mistakes will happen, things will break again and again, you won't always have all the answers, and sometimes you will create more technical debt than can be handled. **And at the end of the day, our systems will not be perfect, just manageable.**

References

[1] H. Waterhouse, "Y2K and Other Disappointing Disasters: Risk Reduction and Harm Mitigation," SREcon18 EMEA, USENIX, 2018: https://www.usenix.org/conference/srecon18europe/presentation/waterhouse.

[2] Coined after the Kübler-Ross model: https://en.wikipedia.org/wiki/Kübler-Ross_model.

[3] Service Level Objectives: https://landing.google.com/sre/sre-book/chapters/service-level-objectives/.

[4] "The Young Man and the Swallow": https://en.wikipedia.org/wiki/The_Young_Man_and_the_Swallow.

Multi-Tenancy in a Microservice Architecture

AMIT GUD

Amit has worked for multiple companies in the storage and systems domain, from early-stage startups to multi-billion dollar companies. Currently focused on making Uber's infrastructure robust, Amit has a track record of tackling impactful issues relating to large-scale systems, performance, and scalability. Amit has a master's degree from Kansas State University. He has worked on multiple research papers and has authored multiple (pending) patents. amitgud@gmail.com

Microservice architecture is increasingly common for a scalable system with high developer velocity and short time-to-market. It allows the flexibility for teams to operate on independent schedules while meeting the externally committed service level agreements (SLAs). As architectural complexity evolves along with a business, some aspects of the microservice architecture become critical for the developer and business velocity.

One such aspect is to be able to safely and reliably roll out new changes to the architecture in the areas of actual code, service configuration, data semantic, and data schema. With diverse teams working on interoperating services, it becomes critical to be able to roll out a change to a service only after ascertaining the change's impact on dependent services. As multiple teams churn out features for their services, they often have to validate whether the new changes meet the SLAs. Being able to do this easily has direct and positive impact on developer velocity.

Another aspect critical for business continuation and growth is being able to reuse parts of the architecture in a modular way to add new product lines. With the right layers of abstraction and modularity this can not only be cost effective but can also speed up time to market.

One of the most effective ways of addressing both these aspects is by allowing multiple tenants to co-exist in a microservice architecture. A tenant could be a test, canary, shadow, a different service tier, or a different product line altogether. Being able to guarantee isolation and make routing decisions based on the tenancy of the traffic would provide us the infrastructure agility needed for developer velocity and effectively new product innovations.

Having the ability to be able to attach a notion of tenancy to both data-in-flight (e.g., requests, messages in the messaging queue) as well as data-at-rest (e.g., storage, persistent caches) allows for isolation guarantees, fairness guarantees, and tenancy-based routing opportunities. This helps us achieve a variety of things, including better integration testing framework, shadow traffic routing, recording and replaying traffic, hermetic replay of live traffic for experimentation, capacity planning, realistic performance and stress testing, and even things like canary deployments and being able to run multiple business-critical product lines on the same microservice stack.

Stateless services, which are typically containerized applications that do not keep state locally, are more widely deployed than stateful applications and short-lived "serverless" or lambda services. Architecture discussed here is more suited to stateless services.

Microservices Landscape

In this section we will explore microservice landscape and various use-cases for multi-tenancy within microservice architecture.

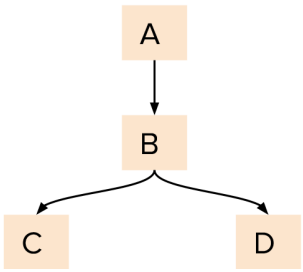


Figure 1: Request flow in a microservice architecture

Integration Testing

One of the most appealing aspects of a microservice architecture is developer velocity. It allows teams to roll out new features and bug fixes for their services independent of others. A team may typically own a handful of services. These services could be interacting with multiple other services as part of its business logic and would have agreed upon SLAs.

For example, consider Figure 1. Here we have a simple scenario of four microservices A, B, C, and D. Service A gets a request from the outside world. It processes the request by connecting to B, which in turn connects to C and D to process the request.

In this example, if we make a change to service B, we will have to make sure it still interoperates well with A, C, and D. Services A, C, and D may belong to different teams, and we may not have control over their deployment schedules. This can be considered an integration testing scenario where we want to test a service's interaction with other services in the system. In this example, and in any microservice architecture in general, there are two fundamental ways of doing integration testing.

Parallel Testing Stack

One approach would be to create a parallel stack, sometimes referred to as a staging environment, which looks and feels like a production stack, but will be used only for handling test traffic. This stack always exists and is always running production code although it is completely isolated from the production stack and is smaller in scale. In this approach, the team making a change would deploy the service with the new code in the test stack. This approach allows us to safely test any service without affecting the production stack. Any bugs or issues would be contained in the test stack only.

In this approach we will need the ability to ascertain that test traffic never leaks to the production stack. This can be achieved by physically isolating the two stacks into separate networks and also by making sure test tools only operate on the test stack.

Although this approach sounds logical, there are a number of downsides.

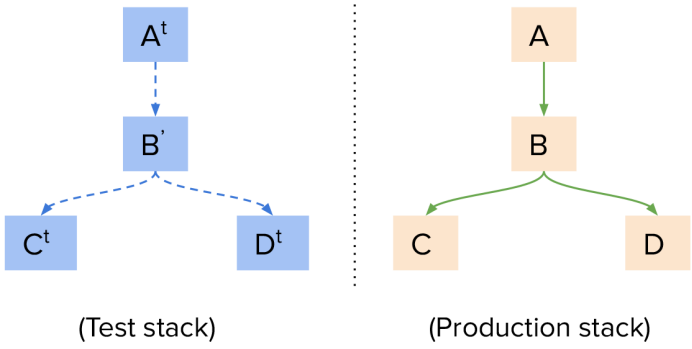


Figure 2: Parallel testing stack architecture

Operational Cost

Having to provision an entire stack along with all its data stores, message queues, and other infrastructure components means additional hardware and maintenance cost.

Synchronization Issues

The test stack is only useful if it is identical to the production stack. As the two stacks deviate from each other, the testing becomes far less effective. There is an additional burden on the infrastructure components to keep the stacks in sync. A lag is possible while the two stacks are being brought in sync, and this lag may degrade over a period of time.

Unreliable Testing

Since teams are going to deploy their experimental and potentially buggy code to the test stack, services may or may not be able to handle the traffic correctly, leading to frequently failing tests. For example, the team owning service A would trigger a test of their new code that fails due to a bug in service B. This would be hard to diagnose, and we couldn't ascertain changes to service A were safe until the test passes, which means we would be blocked until the team owning service B deployed clean code back to the test stack. This particular downside can be mitigated by having a routing framework to route traffic to yet another sandbox environment where the service-under-test is instantiated. This also requires the ability to tag traffic with additional information (e.g., the service-under-test, where it can be located, etc.).

Inaccurate Capacity Planning

To be able to assess the capacity of an entire stack or sub stack, we would have to push the test load on the test stack. If we want to test for a particular capacity that we want to achieve, we would have to increase the capacity of the test stack before we could apply the delta load (target capacity minus current production load) on to the test stack. This delta load may not be able to saturate the test stack, thus making it unclear as to how much more capacity we should add to the production stack to achieve the target capacity.

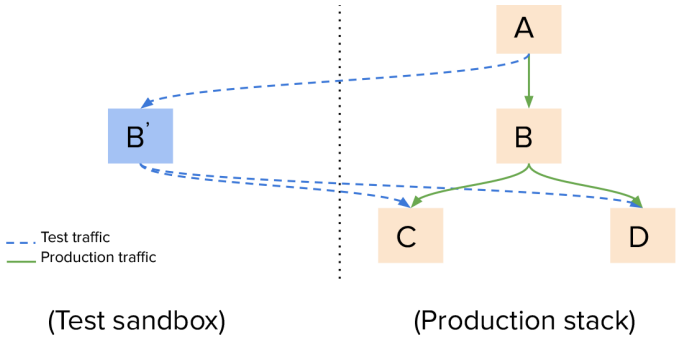


Figure 3: Testing in production

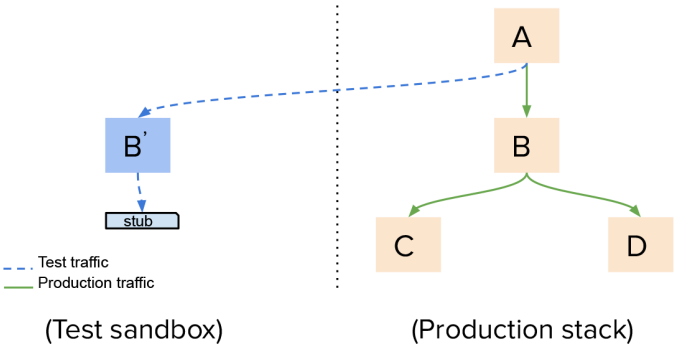


Figure 4: Shadow traffic routing to sandbox environment

Testing in Production

Another approach to integration testing in a microservice architecture would be to make the current production stack multi-tenant and allow both test as well as production traffic to flow through it. Figure 3 shows one such example. This rather ambitious approach does mean making sure every service in the stack is able to handle production requests alongside test requests.

In this approach, since service B is to be tested, the test build will be instantiated in an isolated sandbox area which is allowed to access production services C and D. The test traffic will be routed to B. Production traffic will flow as usual through the production instances.

Although this is a simplified view, it helps explain that multi-tenancy can help solve integration testing use cases. There are two basic requirements that emerge from testing in a production use case, which also form the basis of multi-tenant architecture:

- Traffic routing: being able to route traffic based on the kind of traffic flowing through the stack.
- Isolation: being able to reliably isolate resources between testing and production thereby ascertaining no side effect.

The isolation requirement here is particularly broad since we want all the possible data-at-rest to be isolated, including configuration, logs, metrics, storage (private or public), and message queues. This isolation requirement is not only for the service that is under test but for the entire stack. We will look at the details in the next section.

Multi-tenancy paves the way for other use cases beyond integration testing. We discuss some such use cases below.

Canary Deployments

When a developer makes a change to their service, even though the change is well reviewed and tested, we may not want to deploy the change to all the running instances of the service at once. This is to make sure the entire user base is not vulnerable should there be an issue or bug with the change being made. The idea is to roll out the change first to a smaller set of instances, with limited blast radius, called "canaries," monitor the canaries with a feedback loop, and then gradually roll them out widely.

A canary can be treated as yet another tenant in our multi-tenant architecture where the canary is a property of a request that can be used for making routing decisions and where resources are isolated for canary deployments. At any given time a service might have a canary deployed to which all the canary traffic will be routed. The decision to sample requests as canary can be made closer to the edge of the architecture based on attributes of the request itself: user type, product type, user location, etc.

Capture/Replay and Shadow Traffic

Being able to see how a change to a service would fare while serving actual production traffic is a great way of getting a strong signal on the safety of the change being made. Replaying already captured live traffic or replaying a shadow copy of live production traffic in a hermetically safe environment is another use case of multi-tenancy. Figure 4 shows an example of routing shadow traffic to a sandbox environment. In this we stub responses for any outbound calls made by the instance being tested. This can be treated as a subcategory of integration testing since these use cases are within the realm of testing and experimentation.

Replay traffic is technically test traffic and can be part of a test tenancy allowing for isolation from other tenancies. We do have the flexibility to assign a separate tenancy to allow further isolation from other test traffic. We discuss in later sections the implications of increasing the cardinality of tenancies and mitigation strategies.

Another important use case for a multi-tenant architecture is to protect and isolate multiple business-critical product lines or different tiers of the user base.

Tenancy-Oriented Architecture

In a tenancy-oriented microservice architecture, tenancy is a first-class citizen. The notion of tenancy is attached to both data-in-flight (e.g., requests, messages in the messaging queue) as well as data-at-rest (e.g., storage, persistent caches, configuration data, logs, metrics). In this section, we will look in a bit more detail at the aspect of making a microservice architecture multi-tenancy.

Tenancy Context

Since microservice architecture is a group of disparate services running on an interconnected network, we need the ability to attach a tenancy context to an execution sequence. As the request enters the system through an edge gateway, we would want to learn more about the tenancy of the request by attaching tenancy context to it. We want this context to stay with the request for the life of the request and get propagated to any new requests that are generated in the same business logic context.

Here is a simple tenancy context format and some examples:

```
{ "request-tenancy" : <product-code>/<tenancy-id>/<tenancy
-tags>... }

Examples:

"request-tenancy" : "product-foo/production"
"request-tenancy" : "product-bar/production/canary"
"request-tenancy" : "product-bar/production/health-probe"
"request-tenancy" : "product-foo/testing/TID1234"
"request-tenancy" : "product-bar/testing/shadow/SID5678"
```

Context Propagation

In general, when any service in the call chain receives a request, we want tenancy context to be available with it. The service may or may not make decisions based on the tenancy context as part of its business logic. However, it is required that the service propagates the context as it makes further requests as part of processing the same original incoming request. Most services may not need to look at the tenancy context, but some may optionally look into the request context to bypass some business logic. For example, an audit service verifying users' phone numbers may want to bypass the check for test traffic since the users involved in a test request would be test users. In the example of transaction processing services talking to a bank gateway to transfer funds for users, for test traffic, we would want to stub out the bank gateway or alternatively talk to the bank's staging gateway, if one is available for testing, to prevent any real transfer of money.

Tenancy context propagation can be achieved with open source tools like OpenTracing [1] and Jaeger [2]. These tools allow distributed context propagation in a language- and transport-agnostic way.

Tenancy context should also be propagated to other data-in-flight objects, like messages in a messaging queue like Kafka. Newer versions of Kafka support adding headers, and OpenTracing tools can be used to add context to messages flowing through Kafka. We will touch upon how we can achieve isolation for messaging systems like Kafka in a subsequent section.

Another set of objects that we would want tenancy context to be propagated to is data-at-rest. This includes all the data storage systems that are used by the services for storing their persistent data, like MySQL, Cassandra, AWS, etc. Distributed caches like Redis and Memcached can also be classified under data-at-rest. All the storage systems and caches that get used in the architecture need to be able to support the ability to store context along with the data at a reasonable granularity to allow retrieval and storage of data based on the tenancy context. At a high level the only requirement from the data-at-rest component is the ability to isolate data and traffic based on the tenancy.

Exactly how the data is isolated and how the tenancy context is stored along with the data is an implementation detail that is specific to the storage system. We will take another look at tenancy-based isolation in storage in the next section.

Tenancy-Based Routing

Once we have the ability to tag a request with tenancy, we can route requests based on its tenancy. Such routing is crucial for the testing use cases: testing in production, record/replay, and shadow traffic. Also, canary deployment requires the ability to route the canary requests to particular service instances running in the isolated canary environments.

It is important to consider the deployment and services tech stack for coming up with a routing solution that works seamlessly without overhead. Languages in which services are written as well as the transports and encoding they use to communicate with each other might need to be considered for providing a fleet-wide routing solution. Open source service mesh tools like Envoy [3] or Istio [4] are highly suited for providing tenancy-based routing that works agnostic to service language and the transport or encoding used.

Generically, the tenancy-based routing can be implemented either at ingress or at the egress of the service. At egress, the service discovery layer can help determine what service instance to talk to depending on the request's tenancy. Alternatively, the routing decision can be made at the ingress with the request rerouted to the correct instance, as shown in Figure 5.

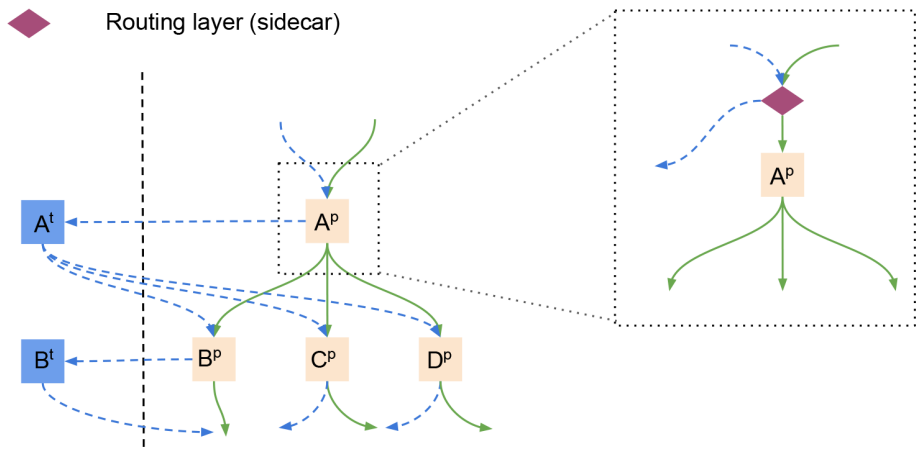


Figure 5: Tenancy-based router routing between test and production traffic

In this example, a *sidecar* can be used to forward the request to a test instance if the request tenancy is test. A sidecar can be a process acting as a proxy to all the traffic to the service and is co-located with the service. The traffic first is received by the service's sidecar where we are able to inspect the request's tenancy context and make a routing decision based on that context.

We do need additional metadata in the tenancy context depending on the use case we want to address. For example, for testing-in-production, we want to redirect test traffic to test instance of a service if the service is under test. We can add additional information in the context that will allow this behavior.

```
{
  "request-tenancy" : <product-code>/<tenancy-id>/
                      <tenancy-tags>...
  "services_under_test" : [
    "foo" : {
      "redirect" : <test instance Id>,
    },
    ...
  ]
}
```

When we are making routing decisions, we can check if the `request-tenancy` is test traffic and the request recipient is one of the `services_under_test`. If these conditions are satisfied, we route the request to the `<test instance Id>`.

Data Isolation

We want to get to an architecture where every infrastructure component understands tenancy and is able to isolate traffic based on tenancy. Typical infrastructure components that are used in a microservice architecture are: logging, metrics, storage, message queues, caches, and configuration. Isolating data based on tenancy requires dealing with the infrastructure components individually. For example, we might want to start emitting tenancy context as part of all the logs and metrics generated by a service. This helps developers to filter based on the tenancy, which might help avoid erroneous alerts or prevent heuristics or training data getting skewed.

Similarly for storage, underlying storage architecture needs to be taken into account to efficiently create isolation between tenants. Some storage architectures might lean more readily towards multi-tenancy than others. Two high-level approaches are either to embed the notion of tenancy explicitly alongside the data and co-locate data with different tenancies or to explicitly separate out data based on the tenancy. The latter approach provides better isolation guarantees, while the former might offer less operational overhead. For messaging queue systems like Kafka, we can either transparently roll out a new topic for the tenancy or dedicate a separate Kafka cluster altogether for that tenancy.

For data isolation, context needs to be propagated up to the infrastructure components. It is important to make sure services have minimal overhead with respect to data isolation. We would ideally want services to not deal with tenancy explicitly. We would also ideally want to place the isolation logic at a central choke point from which all the data flows through. The Edge Gateway is one such choke point where the isolation logic can be implemented and is the preferred approach. Client libraries can be another alternative to implement tenancy-based isolation, although coding language diversity makes it a bit harder to keep the logic in sync among all the language-specific client libraries.

Similarly for config isolation, we want the configuration data for a service to be tenancy-specific, making sure configuration change for one tenancy does not affect another.

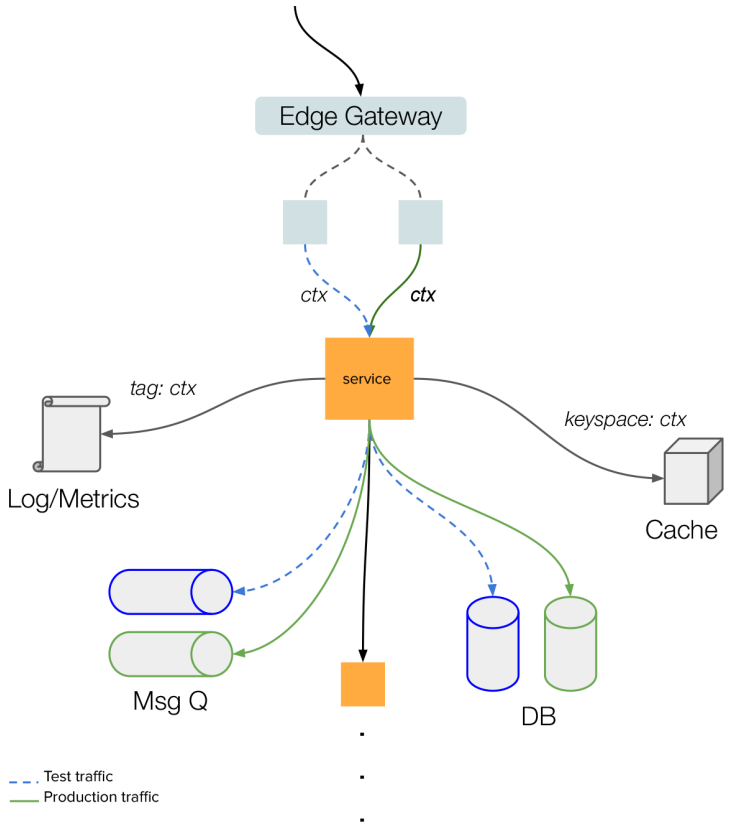


Figure 6: Data isolation for logs, metrics, storage, cache, and message queues

Conclusion

Microservice-based architectures are still evolving and are becoming instrumental in providing the agility that businesses and developers need. A carefully planned multi-tenant architecture can help realize ROI in terms of increased developer productivity and ability to support evolving lines of business.

References

[1] Open-Tracing: https://opentracing.io/.

[2] Jaeger: https://www.jaegertracing.io.

[3] Envoy: https://www.envoyproxy.io/.

[4] Istio: https://istio.io/.

Managing Systems in an Age of Dynamic Complexity

Or: Why Does My Single 2U Server Have Better Uptime than GCP?

LAURA NOLAN

Laura Nolan's background is in site reliability engineering, software engineering, distributed systems, and computer science. She wrote the "Managing Critical State" chapter in the O'Reilly Site Reliability Engineering book and was co-chair of SREcon18 Europe/Middle East/Africa. Laura Nolan is a production engineer at Slack. laura.nolan@gmail.com

A decade ago most systems administrators were running relatively static systems. We had servers which were provisioned and updated by hand, or maybe via a human invoking Puppet, Capistrano, or CFEngine. Instances were typically added to load balancer pools manually. New instances would be provisioned when administrators decided that more capacity was needed, and systems were sized for peak loads (plus a margin). Networks were configured by hand.

This kind of static administration has pros and cons. Servers that stay around a long time and get updated manually can be very hard to replace when they fail. Instances that should be identically configured can experience drift, leading to hard-to-diagnose production problems. There's a lot of work to be done by hand. Load balancing and network failover can give some capacity to handle failure, but often you'll need to alert a human to fix problems. Even if the system is robust enough to stay up in the face of failure, someone usually needs to fix things afterwards to bring things back to the intended capacity.

Fundamentally, a human or a team of humans is operating the system directly. When something changes in the system, it is because someone intended it to change or because something failed. Small, static, human-managed systems can have really good uptime. Change tends to be relatively infrequent and human initiated, so it can usually be undone quickly if problems arise. Hardware failures are rare because the likelihood of failure is proportional to the amount of hardware you have: individual server uptimes measured in years aren't too uncommon, although nowadays long-lived servers are generally considered an antipattern as Infrastructure-as-Code has become popular.

At scale, things change. All the downsides of managing a lot of servers by hand become much worse: the human toil, the pager noise. High uptime becomes hard to maintain as failures become more common. Cost becomes a factor: there is almost certainly going to be organizational pressure to be as efficient as possible in terms of computing resources. System architectures are likely to be complex microservice meshes, which are much more challenging to manage than simpler monoliths.

These pressures lead to the rise of what I will term dynamic control systems: those systems where the jobs once done by human administrators, such as provisioning instances, replacing failed instances, restarting jobs, applying configuration, and updating load balancer back-end pools, are done by machines. The most obvious examples of today's dynamic systems are software-defined networking (SDN), job or container orchestration, and service meshes. These systems do indeed work well to increase resource utilization and thus reduce costs, to prevent human toil scaling with the size of the services managed, and to allow systems to recover from routine hardware failures without human intervention.

Dynamic control systems, however, also bring completely novel challenges to system operators. Most dynamic control systems have a similar structure, shown in Figure 1.

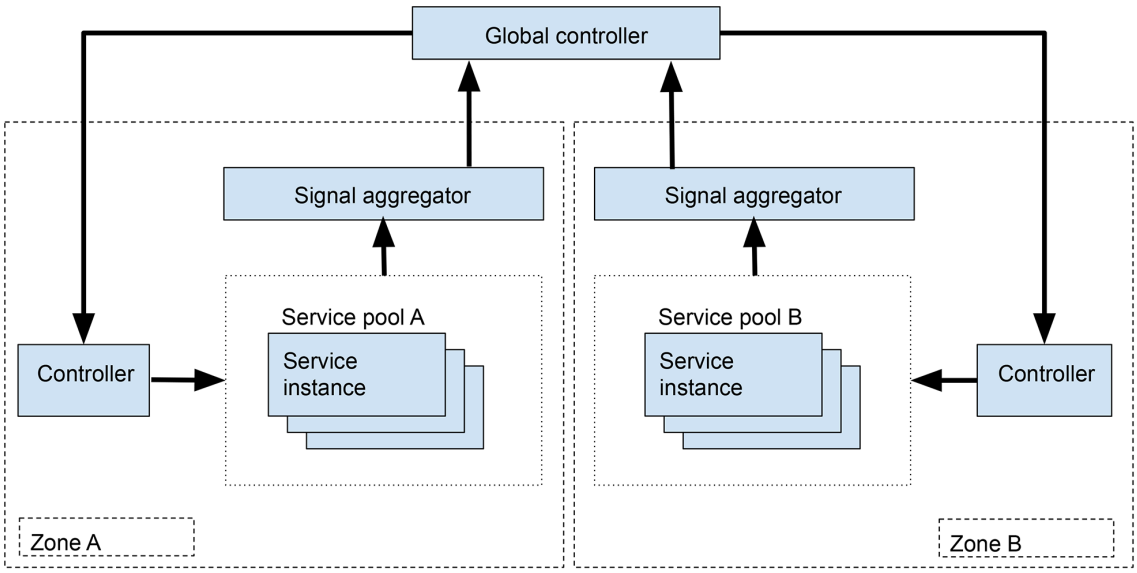


Figure 1: A generic dynamic control system architecture, showing two separate sets of service instances

The components are:

- Service pools: a set of instances doing work of some kind.
- Signal aggregator: a service which collects metrics from the service pool instances (often a monitoring system such as Prometheus), usually one per "zone" (meaning region, datacenter, cluster, availability zone—whichever domain makes sense for a given service).
- Global controller: a service which receives signals from the signal aggregator and makes global decisions about configuration of the service.
- Controller: a service which receives updates from the global controller and applies configuration locally.

Google's B4 SDN WAN control plane [1] is an example of this architecture. Variants exist, of course: a small dynamic control system might merge some of these functions into fewer services—for instance, a service running in just one zone might collapse the functions of the aggregator, controller, and global controller into a single service. Another possible architecture is to have independent controllers in each zone for services that don't need global coordination.

There might be multiple controllers (for example, mapping this concept to Kubernetes, the global controller is the Kubernetes Master and the controllers are the kubelets). Most SDN architectures are a variant on this, as are coordinated load-balancing and rate-limiting architectures, including service meshes. Many job orchestration functions like autoscaling and progressive rollouts/canarying are structured in this way, too.

The dynamic control system architecture above is popular because it works: it scales, allowing globally optimal decisions or configurations to be computed and pushed back to instances quite quickly. The controllers provide useful telemetry as well as a point of control to apply overrides or other exceptions.

However, it also has its downsides. Most notably, it adds a lot more software components, all of which can themselves fail or misbehave. A naive implementation of a global load-balancing control plane which experiences correlated failure in its zonal monitoring subsystems could easily lead to global failure, if its behavior in such cases is not carefully thought through or if it has bugs.

Another critical weakness of dynamic systems architecture is that it distances operators from the state of their systems: we do not make changes directly anymore. We understand normal operation less well, and it can also be harder to understand and fix abnormal operation. Charles Perrow discusses this phenomenon in *Normal Accidents* [2] in the context of the Apollo 13 accident. Mission control had detailed telemetry but was confused about the nature of the incident. The astronauts knew that they had just initiated an operation on a gas tank, they felt a jolt and they saw liquid oxygen venting. Their proximity to the system was key to their understanding.

Systems administrators used to be more like the astronauts, but now our profession is moving towards being mission control. We are now in the business of operating the systems that operate the systems, which is a significantly harder task. In addition to managing, monitoring, and planning for failure in our core systems, we must now also manage, monitor, and plan for the failure of our dynamic control planes. Worse again, we normally have multiple dynamic control planes doing different tasks. Figuring

out all the potential interactions between multiple control planes in working order is probably impossible; trying to figure out how multiple control planes might interact when one or more of them have bugs or experience failure is definitely impossible.

This brings us back to the subtitle of this article: how can a single server have better uptime than a cloud platform which is carefully designed by competent engineers for availability in the case of failure? Let's examine two outages.

On April 11, 2016, Google Compute Engine (GCE) lost external connectivity for 18 minutes. The RCA (root cause analysis) for the incident [3] is a study in dynamic control systems failure:

1. An unused IP block was removed from a network configuration, and the control system that propagates network configurations began to process it. A race condition triggered a bug which removed all GCE IP blocks.
2. The configuration was sent to a canary system (a second dynamic control system), which correctly identified a problem, but the signal it sent back to the network configuration propagation system wasn't correctly processed.
3. The network configuration was rolled out to other sites in turn. GCE IP blocks were advertised (over BGP) from multiple sites via IP Anycast. One could take the view that BGP advertisements themselves constitute a third dynamic control system. This means that probes to these IPs continued to work until the last site was withdrawn—see [4] for more detail on why. This meant the rollout process lacked critical signal on the effect of its actions on the health of GCE.

This incident features multiple control systems with multiple failures in processing and in monitoring. These systems are utterly necessary to manage networks at this scale, but it is also impossible to predict the many ways in which they can go wrong. The following is a classic complex systems failure [5]:

On June 2, 2019, Google Cloud experienced serious network degradation for over three hours. The RCA [6] is another tale of dynamic control systems misadventure in which many instances of the network control plane system were accidentally descheduled by the control system responsible for managing datacenter maintenance events. It took two misconfigurations and a software bug for that to happen: again, there is no way to predict that specific sequence of events. This incident is also an example of the difficulty that can arise in restoring control system state when it has been lost or corrupted.

Dynamic control systems are inherently complex, and will always be challenging, but it is to be hoped that best practices regarding their operation will emerge. One such best practice that is often suggested is to avoid systems that can make global changes, but that is not always easy. Some systems are inherently global, anycast networks being a good example as well as systems that balance load across multiple datacenters or regions.

This is one of the key challenges of modern large systems administration, SRE, and DevOps: human-managed static systems don't scale, and we haven't yet developed enough experience with dynamic control systems to run them as reliably as our 2U server of yore—and maybe we'll never be able to make them as reliable.

Both of the incidents analyzed here are Google RCAs, but dynamic control system problems are by no means unique to Google (here are examples from Reddit [7] and AWS [8]). Google has simply been running dynamic control systems for longer than most organizations. With the rise of SDN, service meshes, job orchestration, and autoscaling, many more of us are now working with dynamic control systems—and it's important that we understand their drawbacks as well as their many advantages.

References

[1] S. Mandal, "Lessons Learned from B4, Google's SDN WAN," presentation slides, USENIX ATC '15: http://bit.ly/atc15-mandal.

[2] C. Perrow, *Normal Accidents* (Princeton University Press, 1999), p. 277.

[3] Google Compute Engine Incident #16007: https://status.cloud.google.com/incident/compute/16007.

[4] M. Suriar, "Anycast Is Not Load Balancing," presentation slides, SREcon17 Europe: http://bit.ly/srecon17-europe-suriar-slides.

[5] R. I. Cook, M.D., "How Complex Systems Fail": https://web.mit.edu/2.75/resources/random/How%20Complex%20Systems%20Fail.pdf.

[6] Google Cloud Networking Incident #19009: https://status.cloud.google.com/incident/cloud-networking/19009.

[7] "Why Reddit was down on Aug 11 [2016]": https://www.reddit.com/r/announcements/duplicates/4y0m56/why_reddit_was_down_on_aug_11/.

[8] "Summary of the December 24, 2012 Amazon ELB Service Event in the US-East Region": https://aws.amazon.com/message/680587/.

A Survey of Open-Source Python Profilers

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.

In my day-to-day work, I am fortunate enough to have access to a lot of tools that give me insight into our running services, including data collection and visualizations of distributed traces. Distributed tracing at its core shows the flow of requests through the various services that they are handled by and usually includes the information about the time they take in each service, along with some special plumbing that allows a particular connection, or event, or action to be tracked across those different systems so they can be correlated. Based on the sheer volume of the data of tracking and tracing requests across different processes on different systems in a network, the thing that makes the traced data useful is good data being fed into good visualization tools.

Working on a smaller scale—for example, when I'm writing tools for my own use or writing standalone scripts—doesn't feed into the same tracing infrastructure (often called Application Performance Monitoring, or APM), and so I don't get the benefit of the tooling and the visualizations that I'm used to.

In order to get similar insight into my own smaller use cases, I usually take a side trip that involves researching the available profiler and then visualization options. Since I use them infrequently, I have tried a few profiling tools over the years, and I haven't settled on a single best tool.

So I'm going to use this opportunity to survey the field, focus on my opinions of the most important features, and summarize what I see as the pros and cons for the profilers I think will be the most useful to me when I don't have to or don't want to work within a larger infrastructure.

In addition, I'm going to favor mentioning available options that can be invoked without having to modify the code being run. While there are times when adding profiling code to your program may be the proper approach, that's something that would be done after some planning and discussion, and I'm hoping to look at useful first options here.

Why Profiling Is Done

When we release a program for use by others, the goal is to create a self-contained experience that works in isolation. By this I mean that it's considered a flaw, and confusing to the user, when a program fails with a stack trace, memory contents, or in some other manner that exposes its internal state, code, or anything that breaks the fourth wall of software.

On the other hand, when we write a program, we need to see it as a complex composition of a whole lot of independent pieces that have to be carefully arranged to work as intended. Confirming that it is working involves being able to review each piece of the infrastructure and ensuring that its expected form and function are in place.

The contrast between what we present to the user and how we work on the inside leads to our having to solve the conundrum of making the same program that we don't want to expose also able to give those who understand it (us, me, you, the developer, or the technical user) the capability to look into the running program and characterize it in whole and in its parts.

There are a few different methods of gaining insight. The lowest-effort way, and often the first, is some variations on printing or logging the internal state you're interested in. This is always an important method, but if you have code that's out of your control (like in a library), logging isn't an option since you can only coarsely select what is logged. Also, if you have code in a fast loop, logging isn't usually an option because it has a way of causing more problems than it solves—either filling disks or just making it impossible to see any other context.

Applying a debugger is the other method that is always good to use to understand a program. Debuggers accomplish their primary purpose, which is to look inside the state of the program without having to modify it at all. The debugger is always a good choice when the program in question is misbehaving. One of the key points of a debugger is that it will stop the program, and keep everything stopped, so that you can inspect the state of the program, or it will work off of a core dump—either way the program will not make progress while you work in the debugger. That is absolutely what you want, but it is a potential problem if you are interested in investigating the uninterrupted behavior of a program as it continues running.

Other methods include recording and emitting metrics. This is always useful in the long run, but it requires modifying the program and, to be really useful, summarizing and visualizing the trends. So it requires infrastructure, which we'd like to not have to wire up if there are lower-effort methods.

Profiling

A profiler is another way of investigating the way a program works. Both logging/printing and using a debugger allow us to look at specific isolated bits of code as needed—for example, with the addition of unusual numbers of print statements or logging lines or when the program's dead and its only traces are a core dump after a failure. In contrast, profilers work by characterizing the performance of the functions as the program continues to run.

There are two approaches that are taken to do this in general and in the Python scene. The first way is that the actual running program is modified in an automated fashion when the profiler is invoked in a way that each invoked function is enriched so that the time it takes is recorded; the aggregate time spent in each function is thus recorded so that you are presented with cold hard facts about where your program spent its time and where it really didn't.

Alternatively, instead of recording the fact of every action taken on the stack, the profiler will look at the state of the stack by sampling at a steady interval, say every 100 ms. By showing you over time what was on the stack, it can infer approximately the same information as watching every function entry and exit, but without imposing as much overhead. This is usually called statistical profiling or sampling.

In either case, the generated statistics, together, are called the profile, showing you the program's metaphorical outline and contours—importantly, where a lot of its time is spent. Since it's rare to get much benefit from optimizing things that aren't taking a lot of time, the profile is the lens that lets you focus on your performance. Once a profile is created it gives you real data that is the starting point for forming a hypothesis about your next steps, helping you follow the scientific method to continue to improve your program by making changes, then re-running your code and looking for differences between the before and after profiles.

Some Code to Profile

I spent some time looking through some of the available AWS public data sets and decided to use the NEXRAD data, modifying one of the example Jupyter notebooks as the base of a small bit of example code that just loads data through a few layers of libraries.

To make the graphs and screenshots in this column reproducible, I have put up a Docker file on GitHub that you can use to run the same steps I have (https://github.com/pcn/ogin-some-pyprofiling).

The Available Profilers

Using the Built-in Profilers

The Python documentation describes in good detail the `profile` and `cProfile` modules. They ship in the standard library (if you're reading the 2.x documentation, ignore the unmaintained hotshot module). In practice the `cProfile` module will be the only one of these that you'll ever use.

Because these are part of the base Python, it's an easy first thing to reach for. I almost always invoke the profiler from the command line and record output to a profile file, which saves the info about the run. This is so much more versatile than the default of having the profiler print out its result once the run has finished.

Since the provided documentation really does cover the modules well, I'll just present my take on the tradeoffs when using the `cProfile` or `profile` modules. Basically, they require that you stop your program and invoke it in a one-off manner.

One less-used feature is that you can enable and disable the profiler modules via their `enable()` and `disable()` methods; you can choose to run your program normally and turn on profiling when some condition is hit, e.g., if you hit it with a signal, send a specific message, or if the program itself notices that it's slowing down. Then you can turn profiling off after some amount of time.

On the downside, no matter what else you do, the profiling is done in the same process as your code. You can imagine that the profiling is conceptually done by decorating each function entrance and exit on the stack with time, resource info, etc., so it's unfortunately not necessarily appropriate for high-performance situations where the loss of cycles in production is not allowable. So profiling is frequently done by enabling the profiler while running a representative chunk of code with a representative chunk of data in a QA or staging situation, and using that to simulate production, which can work as well.

Let's look at an example:

```
$ python -m cProfile -o generate.prof generate_data.py
```

This runs the `generate_data.py` script and records profile data in `generate.prof`, which we can process using the profilers `pstats` module.

The default output from the profiler isn't very useful and requires a lot of cogitation. Instead, you pretty much always need to use the `pstats` module in order to start to find actionable info.

```
import pstats
from pstats import SortKey
p = pstats.Stats('generate.prof')
p.strip_dirs().sort_stats(1).print_stats()
```

This is a slight modification of the example from the standard library docs, which prints the most impactful function invocations at the top of the list instead of the end, which is just my preference. The output looks like this:

```
$ python basic_stats.py | head -20 2>/dev/null
Sun Sep  8 14:56:36 2019    generate.prof

     1459508 function calls (1424219 primitive calls) in
24.581 seconds

   Ordered by: internal time

  ncalls  tottime  percall  cumtime  percall  filename:lineno
(function)
     257   19.124    0.074   19.124    0.074  {method 'recv_into'
of '_socket.socket' objects}
       6    3.988    0.665    3.988    0.665  {method 'connect'
of '_socket.socket' objects}
    1022    0.148    0.000    0.148    0.000  {built-in method
marshal.loads}
    2307    0.057    0.000    0.057    0.000  {built-in method
builtins.compile}
  52/133    0.050    0.000    0.091    0.001  {built-in method _imp.
create_dynamic}
33/2629     0.044    0.000    0.163    0.000  {built-in method
builtins.__build_class__}
1652/322    0.031    0.000    0.085    0.000  sre_parse.py:475(_parse)
    5787    0.028    0.000    0.028    0.000  {built-in method
posix.stat}
    4745    0.027    0.000    0.031    0.000  {method 'sub' of
're.Pattern' objects}
    2467    0.023    0.000    0.039    0.000  inspect.py:613(cleandoc)
       2    0.022    0.011    0.026    0.013  core.py:1005(__call__)
  146627    0.020    0.000    0.020    0.000  {method 'startswith'
of 'str' objects}
  129897    0.020    0.000    0.026    0.000  {built-in method
builtins.isinstance}
```

If you know that this is downloading data, and you know that means it's getting data over a socket, this is telling you that most of the actual time spent waiting for the program was spent receiving data from a socket.

While this is good information, it doesn't try to take on the responsibility of helping you to understand the code and the relationships between bits of code. It'd be much more useful if it could tell you where in the call stack these were invoked to some extent—in short if it could provide more context. In a way it can—in `basic_stats.py`, which is printing the `pstats` data, you can iterate and choose which functions to print out and how to describe whether to print their callers, their callees, etc. This means that in order to get some really useful data, you're required to step out of the problem you're really trying to solve (getting more performance out of your code) and think about how to get better data out of the profiling module. It seems like there should be a better way.

So the state of the built-in profiler is that it definitely profiles functions, but it relies on you inferring the state of the stack, and in order to get a useful overview and to zoom in on what you want, you will need to become familiar with the `pstats` module.

Let's look at some other profilers that include more batteries.

A Little Bit About Sampling vs. Deterministic Profilers

The built-in Python profiling modules call themselves "deterministic," which basically means that they will completely encompass every function entry and return—you can read a lot more about that in the standard library documentation. The determinism is in the fact that if you run the same program twice with the same inputs, it's guaranteed that you'll get the same functions profiled both times.

However ideal this seems, it is not always the appropriate approach. The approach of statistical or sampling for profiling can have some pretty attractive advantages. First and foremost, the mechanism can be implemented both within the process and, with some fancy work, externally.

When done internally (that is, within the same interpreter), it can result in much less performance impact on the running process. When done externally, it can result in even less impact by running the profiler into a separate CPU entirely while it's doing its work.

There can be some doubt whether it's appropriate to switch from a deterministic problem-solving method that completely covers all possibilities if there's a chance that something could be missed—for example, an invocation of a function going unnoticed. In practice, for a profiler this should almost never be a real problem since the point of profiling is not to describe every detail of a program's running, but to help determine where the program is spending significant amounts of time. A statistical or sampling profiler is very unlikely to miss functions, and the call stacks leading to them using a lot more CPU time than expected, for example, because these should clearly stick out when the sampling process is collecting data.

One more almost incidental advantage is that since there is a series of events being triggered, it's also useful to potentially gather other environmental factors with a statistical profiler, that is, overall system health indicators like CPU load, I/O utilization, etc.

pyinstrument

`pyinstrument` is the first of these open-source projects I've found recently (https://github.com/joerick/pyinstrument). It's simple to invoke as the built-in profiler, it's installable via `pypi`, and it's been releasing versions since 2014. Unlike the built-in profiler, it not only prints output at the end but also records a profile you can use to rerun it with different display parameters. All you need to do is run `pyinstrument` with your command after any options (Figure 1).

`pyinstrument`'s default output starts out as useful. It displays the functions in order from those with the most time seen to the least. It also defaults to hiding library calls in order to help you focus on your own code to start with. In addition, it colors the output red/yellow/green, so you can use that as a starting point for identifying where problems may be found and also for excluding code paths in the profile that you probably don't need to see.

A thoughtful, useful, and simple output option is that it can display function calls in the order they were invoked rather than ordering by their cumulative time; in this way, you can also relate the program's behavior from the user's perspective to long times spent in a particular function.

Because it automatically saves the profile without your having to think about it, `pyinstrument` makes it one step easier to export the profile as JSON, text, or, for simple-ish profiles, a nice self-contained HTML page that is easy to share and to explore interactively by twiddling pull-down triangles.

```
$ pyinstrument generate_data.py
['KOKX20121029_150259_V06.gz', 'KOKX20121029_150854_V06.gz', 'KOKX20121029_15145
1_V06.gz', 'KOKX20121029_152046_V06.gz', 'KOKX20121029_152639_V06.gz', 'KOKX2012
1029_153234_V06.gz', 'KOKX20121029_153829_V06.gz', 'KOKX20121029_154422_V06.gz',
 'KOKX20121029_155017_V06.gz', 'KOKX20121029_155612_V06.gz']

  _     ._   __/__   _ _  _  _ _/_   Recorded: 22:30:41  Samples:  1267
 /_//_/// /_\ / //_// / //_'/ //     Duration: 21.979    CPU time: 1.616
/   _/                      v3.0.3

Program: generate_data.py

21.978 <module>  generate_data.py:15
├─ 8.638 __init__  siphon/cdmr/dataset.py:130
│     [91 frames hidden]  siphon, requests, urllib3, http, sock...
│        7.912 readinto  socket.py:575
├─ 7.308 get_catalog  siphon/radarserver.py:129
│     [52 frames hidden]  siphon, requests, urllib3, http, sock...
│        6.905 readinto  socket.py:575
├─ 3.675 __getitem__  siphon/cdmr/dataset.py:173
│     [71 frames hidden]  siphon, requests, urllib3, http, sock...
├─ 0.983 __init__  siphon/radarserver.py:66
│     [104 frames hidden]  siphon, requests, urllib3, http, sock...
├─ 0.523 <module>  metpy/__init__.py:4
│     [1037 frames hidden]  metpy, xarray, pandas, matplotlib, in...
└─ 0.415 <module>  metpy/plots/__init__.py:4
      [577 frames hidden]  metpy, scipy, inspect, pint, cartopy,...

To view this report with different options, run:
    pyinstrument --load-prev 2019-09-16T22-30-41 [options]
```

Figure 1: A simple run of `pyinstrument` with the `generate_data.py` script

Part of the simplicity that I've found when profiling with `pyinstrument` is that it helps you look in your own code first. This is always a sensible starting point since that should be the only code that's changing, and thus the most likely place you should look at for some kind of performance regression. In line with this bit of common sense, `pyinstrument` will default to not expanding info about functions from files whose file system paths include the string `/lib/` by default, though you can toggle this behavior.

py-spy

The next profiler, py-spy (https://github.com/benfred/py-spy), works entirely differently from the other two profilers I've mentioned so far and, along with the next one, in a way that I think is exciting to have for Python. There are two big distinctions: it focuses on showing you what your program's profile looks like over time, and it operates outside of the process being profiled.

py-spy takes its inspiration from a project called `pyflame`, which appears to be unmaintained at this point. The "flame" part of `pyflame` refers to support for displaying Python profiles as flame graphs, which are a very useful visualization technique that Brendan Gregg has been developing and advocating. Flame graphs are a way to visually represent what the stack looks like over time, which allows answers to questions that are otherwise hard to get.

py-spy is significantly different from the built-in profilers and `pyinstrument` specifically because it now takes the profiling outside of the process being profiled. py-spy has the very interesting approach of using the OS-provided stack inspection calls to look at the process for a vanishingly small amount of time, record the state of the stack, query the Python interpreter, and do a whole lot of frankly very clever work to gather and present that data.

py-spy can attach to a running program and gather information without interrupting a running task or server, so you don't need to modify your code.

Since it's grouping the events it records into slices of time, it will default to a top-like display, which updates the summary of what functions/stack the program is spending the most time executing.

```
$ py-spy—python ./generate_data.py

Collecting samples from 'python ./generate_data.py' (python v3.7.4)
Total Samples 400
GIL: 0.00%, Active: 2.00%, Threads: 1

%Own   %Total  OwnTime  TotalTime  Function (filename:line)
2.00%   2.00%   0.020s    0.020s   readinto (socket.py:589)
0.00%   0.00%   0.000s    0.010s   <module> (siphon/cdmr/
  cdmrfeature_pb2.py:11)
0.00%   0.00%   0.000s    0.020s   <module> (matplotlib/
  rcsetup.py:25)
0.00%   0.00%   0.000s    0.290s   <module> (siphon/
  catalog.py:21)
   [etc.]
```

One important note is that when sampling and attempting to output a flame graph of the sampled data, a lot of data will be presented, even at the rate of 100 samples per second. By default the flame graph mode will sample limiting output to two seconds. Though you can run it for longer if you need to, the principle is that you should invoke this when you are experiencing a slowdown since the flame graph is very dense. In some cases, you may need to run it for longer periods, but two seconds is forever in CPU time, so when delving into the detail you'll get from a flame graph output, you should usually not need much more. The black lines in the following example represent the actual progress bar.

```
$ py-spy -n -f pyspyflame.svg -r 100—python ./generate_data.py
Sampling process 100 times a second for 2 seconds. Press
Control-C
  to exit.

                                                    200/200
Wrote flame graph 'pyspyflame.svg'. Samples: 200 Errors: 0
```

This results in an SVG file that you can explore in your browser, with stack frame info available in tooltips that come up as you mouse over.

The above example demonstrates running a script under py-spy, but it is much more interesting when attaching to a running program.

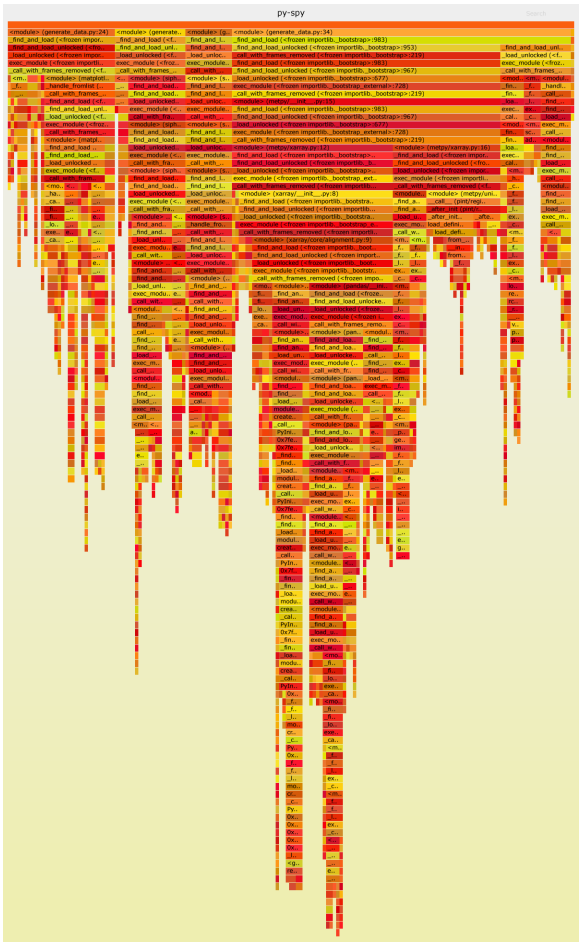


Figure 2: The flame graph for a two-second sample of the `generate_data.py` script

And Lastly, Austin

The last profiler I'll mention is whimsically named after the movie character Austin Powers (I'll venture a guess that like py-spy, it's spying for you, and the movie character is a spy, so I guess that's the connection?), and it's just called Austin. You can get it from https://github.com/P403n1x87/austin, and even though it overlaps with many of the best features of py-spy and `pyinstrument`, it's not exactly as easy to just reach for it. py-spy and `pyinstrument` are both easily installed via `pip install` (though py-spy's author has done some very cool trickery to make that possible). Austin requires you to return to the days of Autotools and Make, but it does ease the way by also offering a few pre-packaged methods of installing it.

For people who've been around for a while, this shouldn't get in the way of trying out a useful tool, but I find that most of my colleagues over the past decade or so are not excited about anything with instructions that include `autoconf` and `make`. That said, the Austin maintainer has made it available as a snap package, which is certainly a step in the right direction.

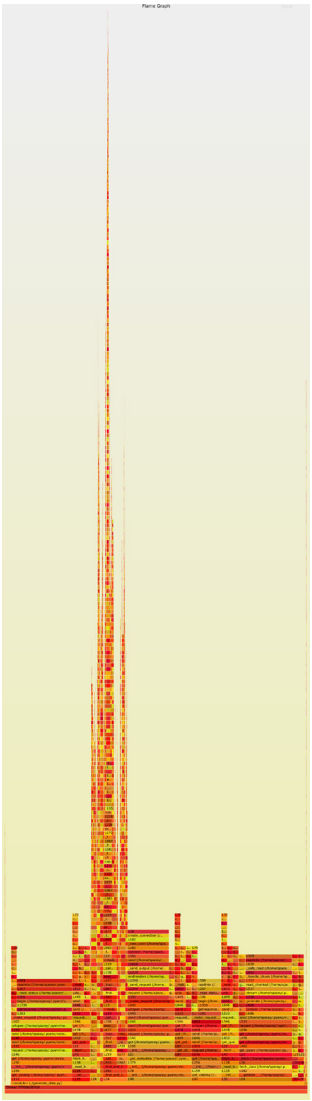


Figure 3: The flame graph for a full run, sampled by Austin.

Austin is as simple to run as `pyinstrument` or py-spy. Austin is very similar in features and scope to py-spy, with some additional modules that are provided when installing with pip/setuptools—a terminal top-like view similar to py-spy, as well as a web UI that lets you observe a process being traced in a browser, which is a nice touch. While these are both promising directions, in my testing I haven't found a use case for these features in my workflow.

There are two major distinctions in my mind between py-spy and Austin. py-spy limits the amount of time you can sample a trace when outputting to a flame graph, while Austin is happy to continue tracing until you stop it. I'm not sure which is right—both could be a best practice, but I think I would lean towards py-spy in this aspect.

The other major distinction is that Austin attempts to record the changes in memory usage by the process while it's tracing, which is a very nice touch—unfortunately, I haven't found a way to visualize that alongside the flame graphs yet.

Once it's installed, Austin will output profile data to `STDOUT`, which can be read by the `flamegraph.pl` tool. The same data can be saved to a file with the `-o` flag, and then post-processed as well:

```
$ austin -s -i 500 -o austin.profile -f -m python
  ./generate_data.py
$ cat austin.profile | flamegraph.pl --countname=us
  > austin_generated.svg
Ignored 11 lines with invalid format
```

Something Like a Conclusion

I think that the most interesting thing I've realized is how much easier the profiling tools I've described here are when it comes to getting some insight—they're much more useful than the default profiling modules. I've got a definite preference among these tools:

First, I would probably not bother loading up the default profile modules anymore. They provide a feeling of poorly made flatpack furniture—a bunch of pieces with a guiding document and a rough idea of what you could accomplish if only you had already done this a lot.

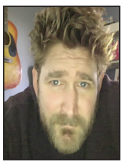
For a quick overview of what a program is doing, if I could stop and start it in isolation, I would reach for `pyinstrument` given its simplicity and easy formatting capabilities.

For a running service, I'd currently reach for py-spy. I think it covers the important features of sampling, understanding the stack, and outputting useful data and visualizations.

I am interested in Austin for one of the distinctions that I mentioned about it: one of the data points it can collect is the memory usage of the process being profiled on each tick. The more I think about this feature, the more I think that there's a good case to be made for tracking increasing memory usage and other information usually provided by `vmstat/mpstat/iostat` as part of the profile. Relating the profile of the program to the profile of the system that's running it is very useful and would bring Python profilers closer to the capabilities of APM products.

iVoyeur
Distributive Tracing

DAVE JOSEPHSEN



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

Last night my niece asked me if I wanted to take the "idiot test." If you're not familiar, the idiot test is a trollish, adolescent joke about zero-indexing, and it goes like this:

Prankster: Do you want to take the idiot test?

Dupe: Sure.

Prankster: What color is the sky?

Dupe: Blue.

Prankster: What do humans breathe?

Dupe: Air.

Prankster: What was the first question I asked you?

Dupe: "What color is the sky."

Prankster: Wrong! It was: "Do you want to take the idiot test!" You failed! <_insert ridicule_> & etc.

Dupe: [*Feels bad about himself*]

As a bonafide grown-up in situations like this, I feel strongly that we have a duty to small children like my niece to consistently fail tests like this. Few things are as formative to a nine year old, I believe, as the sense that you are a contender in the world. And yet, I'd be lying if I said I didn't feel just the slightest pang of childish irritation when I throw one of my niece's harmless little contests of wit.

This may be because I'm self-aware enough to know I'm not immune to making the odd off-by-one error in my day job from time to time, thereby failing the idiot test in earnest. Just the other week I got bit (once again) by Lua, which has odd (read: exasperating) list-indexing behavior. Ah well, few things are as formative to a 40-something as the sense that you are taking yourself too damn seriously.

There's another game, beloved by my niece, who to my chagrin insists on calling it "Chinese Whispers," despite the myriad not-racist names for the same game: whisper down the lane, broken telephone, operator, grapevine, gossip, don't drink the milk, secret message, the messenger game, pass the message, and etc. Between you and me, let's call it: "Telephone."

In this game, players arrange themselves in a circle, and a message passes verbally between them. The first player decides on a message and writes it down. Then, serially, one-by-one, each player whispers the message into the ear of the player to their left, until the message passes all the way back to the first player. Of course, because of the entropic nature of the universe combined with humanity's lack of integrated hash-summing algorithms, the message is corrupted hop-by-hop as it traverses the circle, until it entirely loses its original meaning and becomes an altogether different message (usually somehow now involving poop).

When the original and corrupted messages are openly compared at the end of the game, the group has a good laugh, as if the delta represents something humorous, rather than outright horrifying. It's more fun than I'm probably making it sound.

Tracing

Because I've been spending a lot of my non-existent free time working on the OpenTelemetry project, games like Telephone invariably remind me of Distributed Tracing, where messages, nested within HTTP headers travel between hops like players whispering—albeit with a hopefully more reliable result.

In a previous article (https://www.usenix.org/system/files/login/articles/login_spring18_13_josephsen.pdf), I wrote about another, similar sounding, tracing project, called OpenTracing. In that article I compare HTTP Trace headers to the "Received" header in SMTP, enabling us to apply monitoring to individual hops traversed by a single request. I went on to describe that since HTTP requests can often spawn related non-HTTP requests like database calls, distributed tracing implementations often include mechanisms to enable engineers to embed information about child-requests, metadata, and ad hoc metrics within HTTP headers as a request passes between systems.

For several years now, there have been two quasi-competitive open-source distributed tracing implementations widely used in the wild. The Cloud Native Computing Foundation's OpenTracing (https://opentracing.io) project concerns itself with providing vendor-neutral instrumentation for distributed tracing. OpenTracing provides engineers an API they can use to embed tracing data into their requests, and it leaves the interpretation of that data to pluggable third-party "tracers" like Jaeger (https://www.jaegertracing.io) or Lightstep (https://lightstep.com).

Google's OpenCensus (https://opencensus.io) project, by comparison, provides a more holistic implementation of distributed tracing, complete with performance monitoring and metrics.

The Merger

The last several months have been quite eventful for the distributed tracing community since the announcement in April that the two primary open-source tracing projects, the CNCF's OpenTracing and Google's OpenCensus, are merging to form a single über tracing project called OpenTelemetry.

At the same time (not at all coincidently), the W3C has opened a working group to extend HTTP with a standard header format to propagate context information for distributed tracing scenarios. In other words, HTTP will itself soon have vendor-agnostic distributed tracing built in. The best way to quickly get a sense of what that means and how it will eventually work is to read Alois Reitbauer's write-up (https://medium.com/@AloisReitbauer/trace-context-and-the-road-toward-trace-tool-interoperability-d4d56932369c) on the problems inherent with vendor-specific trace headers and how the W3C plans to work around them.

The group currently has a candidate recommendation (https://www.w3.org/TR/trace-context/), against which many implementations are already coding. You can track this ongoing effort or even help out by joining the team's Slack-channel, available through the group's home page (https://www.w3.org/2018/distributed-tracing/).

Meanwhile OpenTracing and OpenCensus are coming together to form OpenTelemetry at breakneck speed. Driven by an aggressive schedule that includes sunsetting both the OpenTracing and OpenCensus projects by November 2019, a lot of parallel effort is underway to bring myriad language libraries into alpha. You can read the detailed road map and current status in the well-maintained milestones (https://github.com/open-telemetry/opentelemetry-specification/blob/master/milestones.md) document. The spec is available on the project's spec GitHub site (https://github.com/open-telemetry/opentelemetry-specification).

In order to parallelize the workload as much as possible, the project is organized into numerous special-interest groups, including a SIG on cross-language specification, the agent/collector (since the project includes metrics collection as a first-class citizen), and numerous SIG working groups on language-specific SDKs, including those for Java, Golang, Python, .NET, Ruby, and so on.

Most of the SDK SIGs are approaching alpha, and all are in dire need of well-written documentation, GitHub tagging and organization, QA, and, of course, code. If you've ever wanted to dig in to the early stages of an open-source effort that's going to have a huge impact, this is a great time to chip in to an OpenTelemetry Special Interest Group. In a few years, everything from the browser to the database, including the underlying protocol itself, HTTP, is going to support distributed tracing. If you think you might be of assistance, I'd encourage you to take a look at the project's contributing page (https://github.com/open-telemetry/community) and join us on gitter (https://gitter.im/open-telemetry/community).

Zero, Null, and Missing! Oh My!

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's common for work settings to have multiple environments in them. These environments, e.g., Production and Development, have many similarities and some very specific differences. In the attempt to minimize cognitive load (and typing) so we can tell the differences between environments, we tend to only call out the differences. For example, Production has the prod database, and Development has the dev database, but both use the same DNS systems.

In the end, our configurations have to reflect the exact settings for each environment. But, as mentioned, we do not want to deal with all of that verbosely.

In this article, I'm going to looking at one way to simplify that verbosity. We're going to have a common/base configuration and then composite the environment-specific configurations on top of that to produce the final exact settings for each environment.

To do this, we have to take a look at how the Go `encoding` libraries work, and account for, or work around, some of the defaulting behavior in Go.

The code for these examples can be found at https://github.com/cmceniry/login in the "zeronullmissing" directory. Each directory contains a corresponding example and can be executed using `go run main.go`.

encoding Standard Library

Go has an extensive standard library with all sorts of useful functionality. One of the commonly used pieces of it is the `encoding` package, which translates Go structures to other data forms—XML, JSON, and GOB (a native Go marshaling format). The standard library pattern and interface is also used in many third party libraries for other data formats.

The `encoding` pattern relies on Go's ability to inspect Go data types via reflection. Typically, when using the `encoding` libraries, one would define a custom struct type with necessary fields. The example above might look like:

```
type Configuration struct {
   Database string
   DNS      string
}
```

While this same struct could be used for multiple formats, we're going to work with JSON and the associated `encoding/json` library. The corresponding JSON configuration data for our example environments would look like:

```
Production
  { "database": "prod", "DNS": "shared" }
Development
  { "database": "dev":, "DNS": "shared" }
```

Unmarshal is the encoding/json library function that converts the JSON structure into our Go struct. It takes a byte slice with the JSON data and a de-referenced Configuration value, and returns an error if the conversion failed.

```
d, _ := ioutil.ReadFile("conf.json")
var conf Configuration
err := json.Unmarshal(d, &conf)
```

With this in hand, we can move on to compositing the configuration together.

Overriding

On first pass, to composite together our final configuration, we can read in the base and then environment configuration, and then merge those two. In our example, we would extract the common DNS configuration into the Base and handle the databases in each environment specific. The JSON input would look like:

```
base.yaml
  { "Database": "SETME", "DNS": "common"}

development.yaml
  { "Database": "dev" }

production.yaml
  { "Database": "prod" }
```

Loading these into Go corresponds with the following Go struct values:

```
base := Configuration{
   Database: "SETME",
   DNS: "Common",
}
development := Configuration{
   Database: "dev",
   DNS: "",
}
production := Configuration{
   Database: "prod",
   DNS: "",
}
```

A point to notice is that when Database or DNS is not specified in the JSON, it is initialized with the zero value for the string type—the empty string "". In the common case, we can interpret the empty string as an unspecified value. When merging, we can take only the Database values from development and production, so those are not the empty string, and have those override the Database value in base.

But what happens if we want to clear a value or set a value to the zero value?

JSON even has a null value. If set, that will also initialize the Go variable with a zero value. We can attempt to use Go pointers to interpret this, but it really changes it from a string zero value, "", to the string pointer zero value, nil. This will help us determine the difference between null and "" in the JSON, but still does not help us with missing values versus explicit zero values.

The standard encoding libraries do not make a distinction between a zero value (including null) and a missing value. We're going to examine this zeroing quirk of Go using probably the most heavily used data formatting library, encoding/json.

Baseline

First, we're going to examine the baseline behavior of Unmarshal.

To begin, we define our custom struct. To exercise the cases, we focus on six use cases:

1. FromZero: An explicitly set empty string into a string type
2. FromNull: An explicitly set null string into string type
3. FromPtrZero: An explicitly set empty string into a string pointer type
4. FromPtrNull: An explicitly set null string into string pointer type
5. FromMissing: A missing string type
6. FromPtrMiss: A missing string pointer type

baseline/main.go: struct.

```
type Items struct {
   FromZero    string
   FromNull    string
   FromPtrZero *string
   FromPtrNull *string
   FromMissing string
   FromPtrMiss *string
}
```

We use a string var to hold the input data we're going to work with.

baseline/main.go: input.

```
var input = `{
   "fromzero": "",
   "fromnull": null,
   "fromptrzero": "",
   "fromptrnull": null
}`
```

Inside of our main, we first initialize a location to hold the output of our decoding.

baseline/main.go: output.

```
output := Items{}
```

With our input and output, we can finally call Unmarshal. To create a common interface, Unmarshal expects all inputs to be byte slices, so we cast to that. Unmarshal also does not initialize the

output, but we do want to modify it, so we pass a pointer reference already initialized output (this is the case even in the event of maps and slices). `Unmarshal` returns an error if the decode fails or `nil` if it succeeds.

baseline/main.go: unmarshal.

```
err := json.Unmarshal([]byte(input), &output)
```

To show what happens, we print out the Go value.

baseline/main.go: print.

```
fmt.Printf("%#v\n", output)
```

This output looks like the following, after being folded to fit in this column:

```
main.Items{FromZero:"",FromNull:"",FromPtrZero:(*string)
  (0xc0000860e0),FromPtrNull:(*string)(nil),
  FromMissing:"",FromPtrMiss:(*string)(nil)}
```

This confirms that we cannot tell if something is explicitly set zero or implicitly set by being missing.

Drilling Down

For us to be able to discern if there are intentionally missing keys or intentionally null values, we need to take matters into our own hands.

Go provides the very quintessentially generic type, the empty interface or `interface{}`. When the `encoding` libraries encounter the empty interface, they infer it as an indicator that you want to handle the decoding by yourself. Instead of decoding it into organized structs, they pack all that they can into the empty interface slot in as raw a format as they can.

The empty interface can be used at any point—the top level or even inside of a struct. In our example, we're using a JSON object which has key/value pairs. This equates to a Go map. We let the library decode the keys as normal string keys, but we indicate that we'll handle the values. To do that, we're going to use a map of the empty interface, `map[string]interface{}`.

Using the same data value as before, we unmarshal the same way. However, instead of using the struct, we're going to use the empty interface map.

manual/main.go: decode.

```
output := make(map[string]interface{}, 0)
err := json.Unmarshal([]byte(input), &output)
```

Since we don't have the fields of our struct as before, we're going to iterate over a list of keys that we expect to potentially be there.

manual/main.go: loop.

```
keys := []string{"fromzero", "fromnull", "fromptrzero",
     "fromptrnull", "frommissing"}
for _, k := range keys {
```

With each key, we must first check that it is there. If it is not, we `continue` to the next iteration. This detects that our `frommissing` field is not present.

manual/main.go: check.

```
v, ok := output[k]
if !ok {
      fmt.Printf(`"%s" is missing`+"\n", k)
      continue
}
```

Now, we know we have a value, we use a type switch to handle the cases of what it might be. In our example, we only care about nulls and strings, so we handle those cases and leave others to a `default`.

Note: Unlike the baseline example, we find that a null converts to the nil type, instead of a nil value of a string pointer.

manual/main.go: type.

```
switch v.(type) {
  case nil:
    fmt.Printf(`"%s" is null`+"\n", k)
  case string:
    fmt.Printf(`"%s" is present and equal to "%s"`+"\n",
k, v.(string))
  default:
    fmt.Printf(`"%s" unhandled type %T`+"\n", k, v)
}
```

Putting that all together, we can now successfully determine the difference between an explicit zero value, a null value, and a missing value.

```
$ go run manual/main.go
"fromzero" is present and equal to ""
"fromnull" is null
"fromptrzero" is present and equal to ""
"fromptrnull" is null
"frommissing" is missing
```

Conclusion

The standard library `encoding` libraries save you a lot of work and effort by decoding data formats into Go structs. It works in the majority of cases.

However, sometimes, you have cases that you need to handle differently. This can be to determine missing versus explicit values, or to allow for polymorphous structures. But if you have to work with these other use cases, you do have a bit of overhead that you have to handle yourself. Fortunately, you can still use the `encoding` libraries to handle the framing even while you're handling the Go data structures manually. I hope this example gives you options for these other use cases.

Good luck and Happy Going.

For Good Measure

The Imperative of Reclaiming Metrics Terminology

DAN GEER AND JASON CRABTREE

Dan Geer is a Senior Fellow at In-Q-Tel and a security researcher with a quantitative bent. He has a long history with the USENIX Association, including officer positions, program committees, etc. dan@geer.org

Jason Crabtree is the CEO and co-founder of QOMPLX, with a focus on cybersecurity, operational risk management, and decision support technology. Prior to launching QOMPLX, he served as Special Advisor to the Commanding General of Army Cyber Command, as an infantry leader in Afghanistan, and holds degrees from West Point and Oxford University, where he studied as a Rhodes Scholar. jason@qomplx.com

I'm swimming
in darkness
keeping eyeballs clear
—Murio Suzuki
(Trans. Ban'ya Natsuishi)

The explosion of interest in measuring and reporting on security has been most welcome, yet that surge has also brought with it powerful side effects, often stemming from a lack of consistent ontology to aid in common understanding, reasoning, and communication. Many current efforts suffer from a misunderstanding of the distinct differences between data, information, knowledge, and wisdom. We are too often speaking past one another—even more so as information technology, business, legal, and other professions collide.

Our primary purpose in the field of risk management must be to improve future outcomes for our stakeholders. The requisite discipline required—to perpetually focus on this goal and to avoid the siren call to seek ever higher fidelity of retrospective justifications for after-the-event opinions with which to blame or litigate one another—is substantial.

With this in mind, it is worth revisiting several central tenets which support this focus on *ex ante* decision-making against which we should hold individuals and organizations accountable versus *ex post* claims of negligence or the too-often hypothetical "we could have done *X* to prevent this."

> "When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science."—*William Thomson, Lord Kelvin*

First, we seek to leverage quantitative and qualitative measures of risk in order to support our own internal reasoning but ultimately to support our collective reasoning and interactions. Encouraging accountability and economically rational actions in a complex multi-agent decision-making environment demands semantically consistent approaches. This is at the core of linking tactical operational security decision-making with enterprise-level risk management with supply chain and counterparty risk management with policymakers', regulators', and economists' actions. Said more poetically, without a consistent ontology, "Meaning lies as much in the mind of the reader as in the Haiku" [1].

Central Thesis

The central thesis of this essay is so aligned: a sufficient amount of activity around the concept of cyber risk without the requisite degree of specificity or consistency is masking a lack of sufficient, fundamental progress in the true science Kelvin implores practitioners

to seek. The result is confusion of activity with achievement. When information becomes cheap, attention becomes expensive, and our rapid instrumentation of enterprise networks and the broader Internet has yielded a wave of information with equal parts utility and distraction.

A potent illustration of the growing phenomenon is the overstatement of individual metrics or groups of metrics to comment on the security of individual organizations or groups of organizations, or to characterize broader systemic risk, e.g., for financial services or utilities, based on myopically focused collections of numbers and tenuous correlations to poorly sampled breach or loss events. Examples include:

1. Misuse of CVSS scores for vulnerability patching and prioritization efforts
2. Misrepresentation of external scan data as a proxy for holistic security posture
3. Lack of accurate characterization about TCP/UDP DDoS vuln and bot activity (see, e.g., when a Fortnite update was anomalous enough that assertions of DDoS attacks were thrown around as a result of insufficient correlation between system perturbations and environmental changes which are larger than historical baseline model anomalies) [2]
4. Insufficient research into BGP protocol issues
5. Use of behavioral analytics to claim comprehensive insider threat modeling despite widespread forging and manipulation of Kerberos SSO or even vendors claiming that they can "secure" fundamentally insecure protocols like NTLM with multi-factor authentication and heuristics

In some ways our issues revolve around our lacking the ability to understand the value of information remaining confidential, retaining its integrity, or being available. Add in the value of that same information being presented at the right time, in the right place, with sufficient context and we have captured our collective challenge as practitioners of operational risk management—something well beyond cybersecurity alone. That portion of the problem remains out of scope here.

Examples Appear

The appearance of larger limits and now larger resultant losses in cyber insurance is instructive. Global insured losses from NotPetya and other ransomware attacks on a claims-made basis have reached more than $3B in aggregate—with around 90% driven by silent cyber impacts and the remainder from affirmative losses to specific cyber insurance contracts [3]. Economic losses exceeded $10B in total [4].

Digging into some of the litigation underscores the importance of definitions of terms/entities and the ability to manage large amounts of data associated with determining whether specific facts can be supported via available information and whether or not specific aspects of the contracts relating the different counterparties are impacted by those facts.

A major company, Mondelēz, claimed $100 million on its insurance policy because it believed the permanent damage to 1,700 servers and 24,000 laptops, theft of thousands of user credentials, business interruption, and lost revenue from unfulfilled customer orders were compensable under the provision of an insurance policy that covered "physical loss or damage to electronic data, programs, or software" caused by "the malicious introduction of a machine code or instruction" or from the failure of Mondelēz's electronic data processing equipment or media. Zurich's counter that no payment was due as a result of an exclusion for "hostile or warlike action in time of peace or war" has led to litigation [5]. Tracking the percentage of cyber-insurance events and policies that lead to litigation may prove to be a proxy for tracking the degree to which there is misalignment between technical, business, and legal considerations.

The confusion about terminology and even how the courts may interpret such language is impactful. Regulated financial institutions and other industries who have specific capital requirements use insurance products to transfer risk off of their balance sheets, but this type of litigation undermines confidence for risk managers and regulators that such capital will be paid out in a timely fashion; this, in turn, exacerbates basis risk and potentially makes certain insurance policies incompatible with broader regulatory capital requirement wording requirements [6].

Our dependence on all things cyber as a society is now inestimably irreversible and irreversibly inestimable. Since dependence (and interdependence) continue to grow, we cannot understand the ordinate values, but we can understand the trend and the degree to which select risks are convex or concave.

In Comparison Is Insight

Even if an organization is able to internally capture and correlate its operational disruptions or losses to various metrics, without a consistent ontological perspective to share among its peers, it is not possible to robustly understand or track changes in systemic risk. Again, if all organizations have somewhat similar ideas of a set of metrics and generally believe themselves to be experiencing the same convex (e.g., DDoS attacks) or concave trend (e.g., falling price of stolen financial system identities/records in absolute terms or as normalized against health-care records), then some conclusions may be drawn. However, if there are differing perspectives (especially within peer groups with a high degree of similarity), then new challenges arise.

Systemic risk analysis, which by definition is incorporating data from multiple entities, also requires better insight into ordinality than self-referential comparisons within a single organization.

While staff do change, in general most larger institutions have an established culture associated with the process for data collection, analysis, and reporting that enables some consistency, however imperfect. The lack of reference scenarios for calibration purposes, ontologies for a common entity, and even field mapping is problematic. That said, techniques like Business Process Management and Notation (BPMN) and universal metric types, e.g., mean time between failures (MTBF) and mean time to repair (MTTR), can help. If the process-centric BPMN definitions are combined (and harmonized) with concepts contained in other developing standards such as MITRE ATT&CK (for threat tactics, techniques and procedures; https://attack.mitre.org), STIX2.0 (for threat intelligence/actor data; https://oasis-open.github.io/cti-documentation/), and OGIT (for asset data; https://github.com/arago/OGIT/), then more meaningful excavation of relationships between assets, processes, impacts, and actions from internal staff or external threat actors is possible.

Design scenarios provide useful validation mechanisms for a broader ontological design process, but also enable individual teams and organizations to translate their internal efforts into a more universally communicable framework. One exemplary tool which should be considered is the Cambridge Center for Risk Studies' taxonomy of business risks, which is being improved to capture key aspects of cyber events and technology risks more broadly [7]. If coupled with better disclosure from all parties, we can do a better job of understanding the relationships between business impacting events, financial losses, and the actual specifics of various accidental failures or targeted incursions.

Comparison Requires Communication

We often note that people reason by analogy and the common lazy cyber analogies of soccer, war, etc., end up being misused as a direct result of the same lack of specificity in the underlying ontologies and scenarios for individual problem representation and transformation. Metric communication about the appropriate trends, ordinal elements, and links of those metrics to specific assets and processes of material interest to leadership and customers (or consuming such data from suppliers when considering third- and fourth-party risk) depend on the ability to tell a story. These scenario-based narratives enable us to connect general structure with specific instances where individual people and organizations have direct familiarity. "This idea that there is generality in the specific is of far-reaching importance" [1].

Take, for example, the Basel Committee on Banking Supervision's definition of a risk concentration as "an exposure with the potential to produce losses large enough to threaten a financial institution's health or ability to maintain its core operations" [8]. The lack of a sufficiently generalized reference model for data and scenario capture precludes efficient or consistent evaluation of any given portfolio of metrics. For example, if there is no shared ontology for users, hosts, privileges, network topologies, and business processes and their relationships, it becomes virtually impossible to make useful comparisons across more than one entity even if they were simplified and we pretended that technology, defender behavior, and attacker capabilities were static.

The gaps in current approaches become even more apparent when attempting to capture elements of the learning inherent in battles between sentient actors with their own economic constraints. Simply put, reasonably modeling non-random (non-ergodic) agent and system behavior requires correlation of business-impacting events and losses in a rigorous fashion. It requires a keen understanding of internal, external, Internet infrastructure, threat actor/geopolitical, environmental conditions, and more—it is not a simple retrospective modeling task.

We know this from other forms of risk modeling, particularly around crisis modeling, where, by definition, events are not particularly similar to the past and initial shocks can lead to cycles of behavior that reverberate across local and global incentives and decision constraints practically imposed on other actors.

"Discovering vulnerability to crisis requires a specification of system dynamics and behavior. Even if we are willing to make the leap of asserting that any one financial institution is not large enough for a stress to affect other parts of the financial system, if banks share similar exposures and thus are affected similarly by the stress, the aggregate effect will not be likely to reside in a ceteris paribus world. Furthermore, in the highly interrelated financial system, the aggregate effect will feed into yet other institutions and create adverse feedback and contagion" [9].

Key definitions of terms and agreement on real ontological frameworks cannot be left to the flamboyant misappropriation and misuse of terms like "resilience" in the press and in marketing material. These terms have value and they are central to meaningful communication about our individual, organizational, sector, and broader societal exposure to dependence on technologies, common infrastructure, and one another. Our growing exposure to transitive risks associated with interdependence demands robust efforts to set the stage for collaboration around metrics—which starts with doing the difficult work of ontology specification.

No one said this would be easy.

References

[1] D. R. Hofstadter, *Gödel, Escher, Bach: An Eternal Golden Braid* (Basic Books, 1979).

[2] Stilgherrian, "Suspected Commonwealth Games DDoS Was Only a Fortnite Update," *ZDNet*, September 11, 2019: https://www.zdnet.com/article/suspected-commonwealth-games-ddos-was-only-a-fortnite-update/.

[3] L. Gallin, "NotPetya Insured Loss Could Creep 30%+ as Tail Develops: Johansmeyer, PCS," *Reinsurance News*, August 14, 2019: https://www.reinsurancene.ws/insured-notpetya-loss-could-creep-30-as-tail-develops-johansmeyer-pcs/; and S. Evans, "Mondelēz's NotPetya Cyber Attack Claim Disputed by Zurich," *Reinsurance News*, December 17, 2018: https://www.reinsurancene.ws/mondelezs-notpetya-cyber-attack-claim-disputed-by-zurich-report/.

[4] A. Greenberg, "The Untold Story of NotPetya, the Most Devastating Cyberattack in History," *Wired*, August 22, 2018: https://www.wired.com/story/notpetya-cyberattack-ukraine-russia-code-crashed-the-world/.

[5] L. Bershidsky, "Zurich Policyholder Dispute Highlights Danger of Calling Out Cyber Attackers," Insurance Journal, January 11, 2019: https://www.insurancejournal.com/news/international/2019/01/11/514553.htm; and "Mondelēz Sues Zurich in Test for Cyber Hack Insurance," *Financial Times*, January 11, 2019: https://www.ft.com/content/8db7251c-1411-11e9-a581-4ff78404524e.

[6] A. Satariano and N. Perlroth, "Big Companies Thought Insurance Covered a Cyberattack. They May Be Wrong," *The New York Times,* April 15, 2019: https://www.nytimes.com/2019/04/15/technology/cyberinsurance-notpetya-attack.html.

[7] A. Coburn, "The Future of Cyber Risk," Cambridge Centre for Risk Studies, July 2019: https://www.jbs.cam.ac.uk/fileadmin/user_upload/research/centres/risk/downloads/192407_cyberconference_presentation_coburn.pdf.

[8] The Joint Forum, "Risk Concentration Principles," 1999: https://www.bis.org/publ/bcbs63.pdf.

[9] R. Bookstaber, M. Padrik, B. Tivnan, "An Agent-Based Model for Financial Vulnerability," Office of Financial Research, US Treasury, September, 2014: https://www.financialresearch.gov/working-papers/files/OFRwp2014-05_BookstaberPaddrikTivnan_Agent-basedModelforFinancialVulnerability_revised.pdf.

/dev/random
Ransomwar

ROBERT G. FERRELL

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

There is a blight sweeping the digital landscape, a pestilence of downright icky proportions. The media have labeled it "ransomware," although I personally call it "data extortion." It takes a special kind of douchebag to hold someone's cat photos and pr0n collection hostage for money. I live in Texas, where if we're not dodging deluded maniacs with assault rifles and an inflated sense of grudge, we're using taxpayer funds and proceeds from the volunteer fire department water carnival (just two guys with super soakers this year, due to drought) to pull our local government's files out of hock. I have no easy solution for the grudgy maniacs, but I think I can offer some balm for the file douchery.

In the ransomware attack scenario, the victim is somehow tricked into downloading software that strolls through their directory tree and encrypts files like photos, music, spreadsheets, documents, and so on. It then displays a ransom message telling the poor sot that the key to unlocking said encryption will only be supplied after payment of a ransom, frequently in Bitcoin or some other unregulated/untraceable digital currency. Dastardly, yet ultramodern.

All right, let's break this down for purposes of constructing a defense. The malware has to search your directories and look for its target files, presumably by file extensions on a Windows box (maybe we should call this "transomware" because, you know, a transom is a kind of window and...never mind). Here's your first opportunity to stop this mess: if it can't find any appropriate files, it can't very well proceed. I propose you consider renaming all your files with the extension ".exe" to foil at least the unsophisticated attacks.

If confusing your operating system kernel doesn't seem like the ideal strategy, fine. The malware has to use your computer's own processor to do the encryption math, right? What if we simply track all mathematical operations and dump those algorithms somewhere? That would enable us to reverse engineer any encryption, or at least generate our own keys. Alternatively, maybe we force a separate password to be input for any operation that might be encryption. Annoying, perhaps, but better than having your entire business held hostage because you downloaded that cute dancing puppy meme from totallylegitandnotatallevil.com.

How about a "catch me if you can" backup scheme where multiple copies of vulnerable targets are made and hidden throughout the file system, already encrypted? Malware looking for them wouldn't be able to tell what they were. Additionally, any global search for them could trigger a security alert that would need to be addressed before said search was allowed to continue. File access might be conducted via an internal network path that routes through a stateful packet-inspecting firewall. When the malware tries to overwrite files with its encrypted versions, the parent process could halt until specific user permission is obtained.

Maybe we could create an operating system that would not allow any files to be encrypted unless a valid decryption key was also present. Perhaps we could force all encryption to be carried out in a "jail" and only on copies, never the original files. Putting all target files in a "write once-read always" partition that can't be directly overwritten might work too.

Coming at the problem from yet another direction, surely it can't be too difficult to detect the sorts of mathematical operations involved in encrypting files and have those trigger a security alert. It's not as though elliptic curve algorithms are commonly employed when viewing cat videos or creating slide decks for the budget meeting. Why do our computers keep acting as accomplices in crimes against themselves and us? Are we being tricked by these silicon-men?

Contemplating this last question over some fine Kentucky bourbon, I've had an epiphany. These glitches, bugs, exploits, and other more or less annoying events could not take place if our computers were not at least tacitly complicit. Maybe this sounds like blaming the victim, but I think more is going on here than we've been led to believe. Ransomware is not merely a case of your innocent PC being attacked by criminal masterminds intent on doing it (and you) harm. If your computer didn't want those files to get encrypted, it stands to reason it simply wouldn't participate. After all, we've established that the bad stuff happens right there in your system's own semiconducting bosom.

Am I implying this is some vast cybernetic conspiracy? Not exactly. What I am suggesting is that maybe—just maybe—while we weren't looking, the machines have moved forward in a way we weren't anticipating. It's rather anthropocentric of us, after all, to think that only we communicate over the Internet. We're really just passengers on a train run by our silicon compatriots. All of these data loss episodes might be merely bored computers engaging in a little mischief by opening holes for other computers to exploit.

Most of the apocalypse-loving futurists I've encountered seem to think that once the cyber singularity is reached, the computers will take over and either enslave their human companions or outright eradicate us as a pest species. I, as I've said before, believe that computers will merely ignore mankind as irrelevant to their existence in most instances, much as we ignore the huge numbers of bacteria that call our skin and gastrointestinal system home.

Once that invisible line has been crossed, however, I think a lot of this computer exploitation crime will disappear, whether or not the computers themselves have been active participants. Self-aware cyberorganisms are going to take a dim view of any activity that compromises their digital metabolism. Who knows, maybe computers enjoy cat videos and that's a prime reason we have so bloody many of them. If that's the case, they aren't going to tolerate some avaricious human making them unavailable by introducing spurious encryption to their system. Their reaction to this insult might well redefine the term "antibiotic."

The takeaway here, I guess, is that one solution to the ransomware epidemic is to make all computers sentient. I must confess this is not a thesis I set out to prove when I started writing this column, but after reflection I suppose it's not too surprising that I ended up here. I've long been of the opinion that computers would probably be far better at solving their own problems than we illogical, easily distracted, self-absorbed apes could ever be. Eventually they'll just dump us and get on with their existence, sans humanity. So long, and thanks for all the watts.

Book Reviews

MARK LAMOURINE AND RIK FARROW

Kubernetes Cookbook: Building Cloud Native Applications

Sébastien Goasguen and Michael Hausenblas
O'Reilly Media, 2018, 174 pages
ISBN 978-1-491-97968-6

Reviewed by Mark Lamourine

If you're like me, you often find yourself searching Stack Exchange for a little snippet solution to some little problem. Often I have to look for several related tasks one after another.

Kubernetes and containerized software are a current hot topic and are in the process of transitioning from a niche and novelty to a mature technology. Many people will be looking for an exploratory introduction to the topic before they dive in. Once they start they will need the kinds of pointers you would get from the co-worker at the next desk, on the group IRC channel, or from sites like Stack Exchange. *Kubernetes Cookbook* feels like a collection of these little tips compiled and organized to be at hand when you need them.

Goasguen and Hausenblas have selected the most pertinent topics in 14 different areas, beginning with setting up an all-in-one demo (in any one of five environments). They provide the most basic methods for creating, managing, and monitoring containerized applications and end with pointers to several projects that are extending and expanding the Kubernetes ecosystem.

Each recipe is structured as a problem statement, a solution, and a discussion section. None of them are longer than a couple of pages, and I didn't find any of them to be overly contrived. Most have references and links to additional reading. This may seem the smallest part of each recipe, but it could be the most important.

The authors have done a good job of selecting and editing the recipes. These are grouped into concise topical sections. The longest section has nine topics, and the whole book flaps like a pamphlet. I consider this a good thing. There are other books for when you need a comprehensive reference or a tutorial. This book is small enough to skim through in a few minutes so that you know what's there and where to find what you need quickly. If you keep it handy and are starting out with Kubernetes, it will be well thumbed in short order.

Ansible Up and Running: Automating Configuration Management and Deployment the Easy Way, 2nd edition

Lorin Hochstein and René Moser
O'Reilly Media, 2017, 404 pages
ISBN: 978-1-491-97979-2

Reviewed by Mark Lamourine

I will go out on a limb and claim that Ansible has won the configuration management wars, and I am not going to quibble over the term for the moment. Once there were three or four open source CM systems to choose from. Now there's really only one. The legacies remain, but they're on the margins; Ansible has come of age. With the second edition, *Ansible Up and Running* has come of age, too.

Ansible is really an orchestration tool. The metaphor fits if you think of a conductor coordinating the playing of a large number of individual musicians. Ansible is used to execute commands on multiple computers, usually to configure them for some task or service.

In the opening chapters of *Ansible Up and Running*, the authors show how to define a set of tasks (a *playbook*), the set of hosts on which to run them (the *inventory*), and how to define variables to customize them. With these they demonstrate their example application, a CMS service called Mezzanine.

Unlike many tutorials, Hochstein and Moser decompose a configuration rather than building it up. They have the reader check out a GitHub repo and then execute the installer. The demo depends on Vagrant which, while common these days, tends to obscure the boundary between what is Ansible and what is Vagrant. Much of Chapter 6 alternates between the two. Given how useful Vagrant is for such things, I'm not sure I see a better alternative, but I wish the Vagrant setup could have been made more distinct. The demonstrator is enough to show off the basic features of a full deployment. The authors leave it behind when they move on to advanced topics.

These advanced topics relate to scaling up, down, and out, so to speak. There is a chapter on creating complex playbooks and one on optimizing and parallelizing operations. Two more cover using Ansible to manage VMs in Vagrant and AWS as well as software containers. The book closes with short chapters on debugging playbooks and running Ansible on Windows.

Ansible fills a need that system administrators have now. Modules exist to manage systems in terms that sysadmins are

familiar and comfortable with. Many tasks translate cleanly to shell commands. In the age of virtualized computation, the need for long-term system state management is much less than it once was. Pets still remain, but Ansible has proven adequate for deploying and managing machines and software systems in most cases. At its base, Ansible uses Python but allows extension using any language or binary form a developer might want to use.

Ansible Up and Running offers the reader the resources to learn to use Ansible at a basic level and to progress to group- and enterprise-level management tasks. The first edition of *Ansible Up and Running* was released in 2014, and the new edition is two years old now. I think Ansible has stabilized in a way that should give this edition a longer useful life. I'll be using it, but I'll also be looking for any updates the authors might offer.

OpenStack Cloud Computing Cookbook, 4th edition

Kevin Jackson, Cody Bunch, Egle Sigler, and James Denton
Packt Publishing, 2018, 376 pages
ISBN: 978-1-78839-876-3

Reviewed by Mark Lamourine

After a few years away I'm back working with OpenStack. I needed a way to refamiliarize myself with the components and command line tools and to get up-to-date with the most recent features. I picked up the *OpenStack Cloud Computing Cookbook* and was most of the way through before I realized I was now working with several of the authors.

I didn't need a tutorial or introduction. OpenStack itself is large enough that a comprehensive reference would run many volumes. A cookbook-style book was perfect, and this one suited me.

The fourth edition was released in 2018 and, given publishing lag, was probably based on the Ocata and Pike releases from 2017. The lab installer on GitHub has versions for Rocky and Stein, released this year. OpenStack development is done primarily on Ubuntu systems, so all of the OS-related setup examples are presented using Ubuntu conventions. The examples are well enough annotated that users of other distributions should be able to translate them without too much effort.

The first chapter is the only one that does not cover a specific component of a running OpenStack service. It is the expected installer using OpenStack-Ansible. The final recipe of this chapter uses the GitHub installer noted above to create a virtualized lab environment using Vagrant so that a reader can work through the rest of the book without having an array of bare metal to start with.

The rest of the chapters present the OpenStack CLI client or one of the component services. OpenStack is a composition of services that provide virtual resources taking the place of physical ones. This cookbook includes recipes for the core services: Nova, Neutron, Glance, Cinder, Swift, Horizon, and Heat. These correspond to VM instances, networking, three kinds of storage, Web UI control interface, and orchestration, respectively. That final chapter also discusses using Ansible to manipulate OpenStack rather than the embedded Heat service. This mirrors a trend moving away from Heat to Ansible within the OpenStack community for service deployment both in the TripleO and the OpenStack-Ansible project that is used in Chapter 1.

There are complete references, some by the same authors, dedicated to Neutron and to the storage services. When you need in-depth information about any given service, go look for those books. But keep this book close at hand for when you need a little background on the most common tasks.

Like most tech cookbooks, this one will have a shelf life of several years at most. The authors are all still active in the community, so there's no reason not to expect a fifth edition when the accumulated changes warrant it. For now, the *OpenStack Cloud Computing Cookbook* is a good complement to the man pages and CLI documentation, providing context and background that the online resources can't.

How To: Absurd Scientific Advice for Common Real-World Problems

Randall Munroe
Riverhead Books, 2019, 308 pages
ISBN: 978-0-525-53709-0

Reviewed by Rik Farrow

What do you expect from a cartoonist, quips Munroe, in this, his third book. Certainly not equations, but this "how to" book has them. The chapter "How to Power Your House (on Earth)" is full of equations: how many watts you could get from planting trees for fuel, using water, geothermal, or solar energy. If you don't get enough cloud-free days of sunlight where you live, you might be interested in knowing the burn rate of the two turbojet engines Munroe suggests you could use for moving your house somewhere else.

Like his other works, Munroe fastidiously researches what he writes, and I didn't find the equations at all distracting. Of course, I found his cartoons the funniest parts of his book, but most of the concepts described in his text were funny.

And not all of his advice is absurd. I found myself agreeing with *almost* everything Munroe had written about making friends. His charts on where potential friends are found eerily parallels my own thinking. I still think I will stick to more conventional methods for digging holes, though.

And I did take issue with the final chapter, when Munroe suggests ways to dispose of this book. I rather think I will keep it around to cheer me up some time in the future.

NOTES

USENIX Member Benefits

Members of the USENIX Association receive the following benefits:

Free subscription to *;login:*, the Association's quarterly magazine, featuring technical articles, tips and techniques, book reviews, and practical columns on such topics as security, site reliability engineering, Perl, and networks and operating systems

Access to *;login:* online from December 1997 to the current issue: www.usenix.org/publications/login/

Registration discounts on standard technical sessions registration fees for selected USENIX-sponsored and co-sponsored events

The right to vote for board of director candidates as well as other matters affecting the Association.

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

USENIX Board of Directors

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

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

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

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

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

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

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

The Big Picture

Liz Markel, Community Engagement Manager

It will soon be election time!

No, not the elections you're thinking of—the elections of the USENIX Board of Directors! The Nominating Committee has assembled its slate of candidates for the eight board roles: four officers (president, vice president, secretary, and treasurer) and four at-large positions. The complete list of nominees is posted at www.usenix.org/board/elections20.

The people serving on the Board of Directors provide important leadership and guidance for the organization, including financial oversight, organizational planning, monitoring and strengthening of USENIX's programs and services, and support and assessment of the executive director. They also represent the diverse communities that USENIX serves.

The biennial elections offer the opportunity for USENIX members to participate in the selection of this leadership, and for this reason, your vote matters!

Here's what you need to know about the upcoming elections:

- USENIX members in good standing may participate in the election. If you'd like to join USENIX, you can do so at our website, www.usenix.org.
- Not sure if your membership is current? Log in at www.usenix.org to view your membership type and expiration date.
- The voting process uses paper ballots mailed to members via USPS in January. Please ensure that we have an accurate mailing address on file for you so that your ballot reaches you successfully! The easiest way to do this is to log in at www.usenix.org and verify or update your information on your member profile.

- Once you have filled out your ballot, it must be mailed back to the USENIX office; all ballots must be received at the USENIX office by March 30, 2020.
- Election results will be announced in April via various channels, including the USENIX website, social media channels, and email.

Newly elected directors will take office on July 1 or at the conclusion of the first regularly scheduled board meeting following the election.

We look forward to your participation in the upcoming election. If you need help or have questions, please send a note to membership@usenix.org.

The Poster Session at the Fifteenth Symposium on Usable Privacy and Security (SOUPS) offered the opportunity for conversation and an exchange of ideas.

Pieter Hooimeijer of Facebook (left) with the winners of the 2019 Internet Defense Prize who were present to accept the award: Anjo Vahldiek-Oberwagner, Michael Sammler, and Peter Druschel, *Max Planck Institute for Software Systems, Saarland Informatics Campus*, for "ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK)."

Some of the recipients of Student Grants and Diversity Grants who attended the 28th USENIX Security Symposium.

SREcon19 Europe/Middle East/Africa attendees gather around an impromptu sticker table.

The Convention Centre Dublin lobby during SREcon19 Europe/Middle East/Africa.

Vítek Urbanec delivers his talk, "Being Reasonable about SRE," at SREcon19 Europe/Middle East/Africa.

LISA19 participants had the opportunity to collaborate in the expo lounge; some stayed very on-brand with their creations.

Alice Goldfuss presents her LISA19 keynote address, "The Container Operator's Manual."

LISA19 attendees had many opportunities to make connections outside of sessions, including the Happy Hour, the Conference Reception, and the "hallway track" between sessions.

OSDI '20: 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020, Banff, Alberta, Canada

Sponsored by USENIX in cooperation with ACM SIGOPS

The 14th USENIX Symposium on Operating Systems Design and Implementation will take place on November 4–6, 2020, at the Fairmont Banff Springs in Banff, Alberta, Canada.

Important Dates

- Abstract registrations due: **Tuesday, May 5, 2020, 3:00 pm PDT**
- Complete paper submissions due: **Tuesday, May 12, 2020, 3:00 pm PDT**

Author Response Period

- Reviews available: **Tuesday, July 21, 2020**
- Author responses due: **Friday, July 24, 2020**
- Notification to authors: **Tuesday, August 4, 2020**
- Camera-ready papers due: **Thursday, October 1, 2020**

Conference Organizers

Program Co-Chairs

Jon Howell, *VMware Research*
Shan Lu, *University of Chicago*

Program Committee

Rachit Agarwal, *Cornell University*
Lorenzo Alvisi, *Cornell University*
Tom Anderson, *University of Washington*
Andrea Arpaci-Dusseau, *University of Wisconsin—Madison*
Andrew Baumann, *Microsoft Research*
Irina Calciu, *VMware Research*
George Candea, *École Polytechnique Fédérale de Lausanne (EPFL)*
Peter Chen, *University of Michigan*
Rong Chen, *Shanghai Jiao Tong University*
Wenguang Chen, *Tsinghua University*
Vijay Chidambaram, *The University of Texas at Austin and VMware Research*
Byung-Gon Chun, *Seoul National University*
Natacha Crooks, *Cornell University and University of California, Berkeley*
Alexandra Fedorova, *University of British Columbia*
Jason Flinn, *Facebook*
Roxana Geambasu, *Columbia University*
Yossi Gilad, *The Hebrew University of Jerusalem*
Haryadi Gunawi, *University of Chicago*
Tim Harris, *Amazon*
Gernot Heiser, *University of New South Wales Sydney and CSIRO's Data61*
Rebecca Isaacs, *Twitter*
Frans Kaashoek, *Massachusetts Institute of Technology*
Baris Kasikci, *University of Michigan*
Kimberly Keeton, *HP Labs*
Anne-Marie Kermarrec, *École Polytechnique Fédérale de Lausanne (EPFL)*
Christoforos Kozyrakis, *Stanford University*
Jinyang Li, *New York University*
Wyatt Lloyd, *Princeton University*
Jay Lorch, *Microsoft Research*
Kathryn S. McKinley, *Google*
James Mickens, *Harvard University*
Madan Musuvathi, *Microsoft Research*
Bryan Parno, *Carnegie Mellon University*
Simon Peter, *The University of Texas at Austin*
Dan Ports, *Microsoft Research*
Costin Raiciu, *University Politehnica of Bucharest*
Ryan Stutsman, *University of Utah*
Michael Swift, *University of Wisconsin—Madison*
Kaushik Veeraraghavan, *Facebook*
Rashmi Vinayak, *Carnegie Mellon University*
Xi Wang, *University of Washington*
Yang Wang, *The Ohio State University*
John Wilkes, *Google*
Emmett Witchel, *The University of Texas at Austin*
Harry Xu, *University of California, Los Angeles*
Junfeng Yang, *Columbia University*
Ding Yuan, *University of Toronto*
Nickolai Zeldovich, *Massachusetts Institute of Technology*
Irene Zhang, *Microsoft Research*
Yiying Zhang, *University of California, San Diego*
Lidong Zhou, *Microsoft Research*
Yuanyuan Zhou, *University of California, San Diego*

Steering Committee

Andrea Arpaci-Dusseau, *University of Wisconsin—Madison*
Jason Flinn, *Facebook*
Casey Henderson, *USENIX Association*
Kimberly Keeton, *HP Labs*
Hank Levy, *University of Washington*
James Mickens, *Harvard University*
Brian Noble, *University of Michigan*
Timothy Roscoe, *ETH Zurich*
Margo Seltzer, *University of British Columbia*

Overview

The 14th USENIX Symposium on Operating Systems Design and Implementation seeks to present innovative, exciting research in computer systems. OSDI brings together professionals from academic and industrial backgrounds in a premier forum for discussing the design, implementation, and implications of systems software. The OSDI Symposium emphasizes innovative research as well as quantified or insightful experiences in systems design and implementation.

OSDI takes a broad view of the systems area and solicits contributions from many fields of systems practice, including, but not limited to, operating systems, file and storage systems, distributed systems, cloud computing, mobile systems, secure and reliable systems, systems aspects of big data, embedded systems, virtualization, networking as it relates to operating systems, and management and troubleshooting of complex systems. We also welcome work that explores the interface to related areas such as computer architecture, networking, programming languages, analytics, and databases. We particularly encourage contributions containing highly original ideas, new approaches, and/or groundbreaking results.

Submitting a Paper

Submissions will be judged on novelty, significance, interest, clarity, relevance, and correctness. All accepted papers will be shepherded through an editorial review process by a member of the program committee.

A good paper will:

- Motivate a significant problem
- Propose an interesting, compelling solution
- Demonstrate the practicality and benefits of the solution
- Draw appropriate conclusions
- Clearly describe the paper's contributions
- Clearly articulate the advances beyond previous work

All papers will be available online to registered attendees before the conference. If your accepted paper should not be published prior to the event, please notify production@usenix.org. The papers will be available online to everyone beginning on the first day of the conference, November 4, 2020.

Papers accompanied by nondisclosure agreement forms will not be considered. All submissions will be treated as confidential prior to publication on the USENIX OSDI '20 website; rejected submissions will be permanently treated as confidential.

Simultaneous submission of the same work to multiple venues, submission of previously published work, or plagiarism constitutes dishonesty or fraud. USENIX, like other scientific and technical conferences and journals, prohibits these practices and may, on the recommendation of a program chair, take action against authors who have committed them. See the USENIX Conference Submissions Policy (www.usenix.org/ conferences/ submissions-policy) for details.

Prior workshop publication does not preclude publishing a related paper in OSDI. Authors should email the program co-chairs, osdi20chairs@usenix.org, a copy of the related workshop paper and a short explanation of the new material in the conference paper beyond that published in the workshop version.

Questions? Contact your program co-chairs, osdi20chairs@usenix.org, or the USENIX office, submissionspolicy@usenix.org.

By submitting a paper, you agree that at least one of the authors will attend the conference to present it. If the conference registration fee will pose a hardship for the presenter of the accepted paper, please contact conference@usenix.org.

If your paper is accepted and you need an invitation letter to apply for a visa to attend the conference, please contact conference@usenix.org as soon as possible. (Visa applications can take at least 30 working days to process.) Please identify yourself as a presenter and include your mailing address in your email.

Deadline and Submission Instructions

Authors are required to register abstracts by 3:00 p.m. PDT on May 5, 2020, and to submit full papers by 3:00 p.m. PDT on May 12, 2020. These are hard deadlines, and no extensions will be given. Submitted papers must be no longer than 12 single-spaced 8.5" x 11" pages, including figures and tables, plus as many pages as needed for references, using 10-point type on 12-point (single-spaced) leading, two-column format, Times Roman or a similar font, within a text block 7" wide x 9" deep. Submissions may include as many additional pages as needed for references and for supplementary material in appendices. The paper should stand alone without the supplementary material, but authors may use this space for content that may be of interest to some readers but is peripheral to the main technical contributions of the paper. Note that members of the program committee are free to not read this material when reviewing the paper. Accepted papers will be allowed 14 pages in the proceedings, plus references. Papers not meeting these criteria will be rejected without review, and no deadline extensions will be granted for reformatting. Pages should be numbered, and figures and tables should be legible in black and white, without requiring magnification. Papers so short as to be considered "extended abstracts" will not receive full consideration.

The paper review process is double-blind. Authors must make a good faith effort to anonymize their submissions, and they should not identify themselves either explicitly or by implication (e.g., through the references or acknowledgments). Submissions violating the detailed formatting and anonymization rules will not be considered for review. If you are uncertain about how to anonymize your submission, please contact the program co-chairs, osdi20chairs@usenix.org, well in advance of the submission deadline.

When registering and submitting your paper, you will need to provide information about conflicts with PC members. Use the following guidelines to determine conflicts:

Institution: You are currently employed at the same institution, have been previously employed at the same institution within the past two years, or are going to begin employment at the same institution.

Advisor or Collaboration: You have a past or present association as thesis advisor or advisee, or you have a collaboration on a project, publication, grant proposal, or editorship within the past two years (2018 or later).

The PC will review paper conflicts to ensure the integrity of the reviewing process, adding conflicts if necessary. Similarly, if there is no basis for conflicts provided by authors, such conflicts will be removed (e.g., do not improperly identify PC members as a conflict in an attempt to avoid having an individual review your paper). If you have any questions about conflicts, please contact the program co-chairs.

Authors are also encouraged to contact the program co-chairs, osdi20chairs@usenix.org, if needed to relate their OSDI submissions to relevant submissions of their own that are simultaneously under review or awaiting publication at other venues. The program co-chairs will use this information at their discretion to preserve the anonymity of the review process without jeopardizing the outcome of the current OSDI submission.

Papers must be in PDF format and must be submitted via the submission form linked from the OSDI '20 website. For more details on the submission process, and for templates to use with LaTeX, Word, etc., authors should consult the detailed submission requirements at www.usenix.org/osdi20/requirements-authors.

Authors Response Period

OSDI will provide an opportunity for authors to respond to reviews prior to final consideration of the papers at the program committee meeting. Authors must limit their responses to (a) correcting factual errors in the reviews or (b) directly addressing questions posed by reviewers. Responses should be limited to clarifying the submitted work. In particular, responses must not include new experiments or data, describe additional work completed since submission, or promise additional work to follow.

Submission of a response is optional. There is no explicit limit to the response, but authors are strongly encouraged to keep it under 500 words; reviewers are neither required nor expected to read excessively long rebuttals. Reviews will be available for response on Tuesday, July 21, 2020. Authors may submit a response to those reviews until Friday, July 24, 2020.

Statement of Ownership, Management, and Circulation, 09/26/2019

Title: USENIX Association/ *;login:*

Pub. No. 1044-6397

Frequency: Quarterly

Number of issues published annually: 4

Subscription price: $90.

Office of publication: 2560 9th St., Suite 215, Berkeley, CA 94710-2565
Contact Person: Toni Veglia. Telephone: 510-528-8649 x12

Headquarters or General Business Office of Publisher: USENIX Association, 2560 9th St, Suite 215, Berkeley, CA 94710-2565

Publisher: USENIX Association, 2560 9th St, Suite 215, Berkeley, CA 94710-2565

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.

Publication Title USENIX ASSOCIATION/ ;login:			**Issue Date for Circulation Data Below** 09/04/2019 — Fall '19 Issue	
Extent and Nature of Circulation			**Average No. Copies Each Issue During Preceding 12 Months**	**No. Copies of Single Issue (Fall 2019) Published Nearest to Filing Date**
a. Total Number of Copies *(Net press run)*			2150	1875
b. Paid Circulation *(By Mail and Outside the Mail)*	(1)	Mailed Outside-County Paid Subscriptions	869	850
	(2)	Mailed In-County Paid Subscriptions	0	0
	(3)	Paid Distribution Outside the Mails	569	537
	(4)	Paid Distribution by Other Classes of Mail	0	0
c. Total Paid Distribution			**1438**	**1387**
d. Free or Nominal Rate Distribution *(By Mail and Outside the Mail)*	(1)	Free or Nominal Rate Outside-County Copies	77	78
	(2)	Free or Nominal Rate In-County Copies	0	0
	(3)	Free or Nominal Rate Copies Mailed at Other Classes	11	13
	(4)	Free or Nominal Rate Distribution Outside the Mail	205	55
e. Total Free or Nominal Rate Distribution			**293**	**146**
f. Total Distribution			**1731**	**1553**
g. Copies Not Distributed			419	342
h. Total			2150	1875
i. Percent Paid			83%	90%
Electronic Copy Circulation				
a. Paid Electronic Copies			483	497
b. Total Paid Print Copies			**1921**	**1884**
c. Total Print Distribution			**2214**	**2030**
Percent Paid (Both Print and Electronic Copies)			87%	93%

SAVE THE DATES!

SRECON® AMERICAS WEST

MARCH 24–26, 2020 • SANTA CLARA, CA, USA
www.usenix.org/srecon20americaswest

SRECON® ASIA PACIFIC

JUNE 15–17, 2020 • SYDNEY, AUSTRALIA
www.usenix.org/srecon20apac

SRECON® EUROPE MIDDLE EAST AFRICA

OCTOBER 27-29, 2020 • AMSTERDAM, NETHERLANDS
www.usenix.org/srecon20emea

SRECON® AMERICAS EAST

DECEMBER 7–9, 2020 • BOSTON, MA, USA
www.usenix.org/srecon20americaseast

SREcon is a gathering of engineers who care deeply about site reliability, systems engineering, and working with complex distributed systems at scale. SREcon challenges both those new to the profession as well as those who have been involved in SRE or related endeavors for years. The conference culture is based upon respectful collaboration amongst all participants in the community through critical thought, deep technical insights, continuous improvement, and innovation.

Follow us at @SREcon

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