;login:

FALL 2019 VOL. 44, NO. 3

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

UPCOMING EVENTS

SREcon19 Europe/Middle East/Africa
October 2–4, 2019, Dublin, Ireland
www.usenix.org/srecon19emea

LISA19
October 28–30, 2019, Portland, OR, USA
www.usenix.org/lisa19

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
Paper submissions due September 26, 2019
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

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
Fall paper titles and abstracts due September 12, 2019
https://www.usenix.org/conference/nsdi20

SREcon20 Americas West
March 24–26, 2020, Santa Clara, CA, USA

HotEdge '20: 3rd USENIX Workshop on Hot Topics in Edge Computing
April 30, 2020, Santa Clara, CA, USA

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
Fall Quarter paper submissions due Friday, November 15, 2019
www.usenix.org/sec20

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

OSDI '20: 14th USENIX Symposium on Operating Systems Design and Implementation
November 4–6, 2020, Banff, Alberta, Canada

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

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

;login:

FALL 2019 VOL. 44, NO. 3

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
Happenstance Type-O-Rama
happenstance.net

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

www.usenix.org

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

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

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

EDITORIAL

Musings

RIK FARROW

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

When I decided to work with computers, I resolved to help make computers easier for people to use. I had already witnessed through my university classes just how difficult and actually inscrutable computers were, and so I hoped that I could make things better.

I failed. I was waylaid by the usual factor: peer pressure. I wanted to be liked and admired by my peer group, and they were programmers and engineers. We loved coming up with elegant solutions, whether it was in the hidden underpinnings of products or in the user interface.

I wrote one of my first Musings about this in 1998 [1]. In that column, I describe the magic of state machines, beloved of programmers and anathema for just about anybody else. My friends and I would wonder why people couldn't program VCRs or set digital watches when we could figure them out without resorting to manuals!

Things today are different. Instead of state machines, we have graphical interfaces with ever-changing sets of symbology. Three vertically arranged dots sometimes means, "Here's that menu you've been searching for!" but sometimes just leads you off on a wild goose chase instead. You are supposed to learn how your new smartphone works from members of your peer group. And by the time you've figured out how to answer your phone, the interface has been updated and you no longer know how to answer your phone.

The desktop metaphor could be called the visual-spatial interface, as it builds on skills familiar to our ancient ancestors. We locate the icon on the screen and manipulate it using a pointing device. Apple made much out of this interface in the '80s, with Microsoft embracing it in the mid-'90s. Visual-spatial design works well because we are familiar with seeing and pointing.

Consider the modern, touchscreen interface as a counter-example. Instead of pointing to what we want others to notice, imagine that the number of fingers we used while pointing was terribly significant, as was the direction we swiped our pointing fingers afterward. Yes, the two-fingered swipe to the left means "Danger, lion!" Or was that just one finger, meaning, "Food item, attack!" Somehow, I am not surprised that finger gestures never caught on with our not very distant ancestors.

The Lineup

Keeping with my theme of making things easier, more efficient, and definitely cooler, we have gg. Fouladi et al., from Stanford University, have created a suite of tools for converting tasks such as large compilations, running tests, and video processing into thousands of cloud functions. While even the concept of a lambda is certain to bewilder mere mortals, I believe this project will prove a godsend to many of the people who read *;login:*. For anyone using lambdas, I strongly recommend you read Hellerstein et al., the fifth cite in this article.

Jangda et al. built Browsix, a browser extension that extends more complete access to the operating system for applications written in WebAssembly (Wasm). Their purpose was to be able to run standard benchmarking tools, and they have done that and noticed that the performance they get from Wasm is not quite what was promised. Reading this article will teach you more about Wasm, but don't go installing Browsix on the browser you use for everyday tasks.

John Koh, Jason Nieh, and Steve Bellovin created E3, a tool for encrypting email while it is stored on mail servers. Instead of relying on knowledge that Johnny doesn't have and wouldn't understand anyway [2], they have built an interface for adding public-private key pairs and transparently encrypting and decrypting mail messages.

Periwinkle Doerfler is researching the intersection between our apps, the device they run on, and our interpersonal relations. She spoke on this at Enigma 2019 [3], where I met her at lunch and decided I should dig deeper by interviewing her. It shouldn't surprise any of us that our inscrutable devices can be used to further abuse by intimate partners, parents, coworkers, and even others we barely know at all.

Dave Dittrich has been in the trenches, reverse engineering malware and DDoS agents since the late 1990s. More recently, Dave has ventured into policy realms as co-author of the Menlo Report. I borrowed from one of Dave's early projects, and basked in my 15 minutes of fame, when I predicted attacks against the Internet giants of the year 2000 days before the attack began. Dave never stopped, creating, for example, the first Forensic Challenge [4].

Anwar et al., from IBM at Almaden, explain why the manner in which Docker creates containers is inefficient. Docker makes creating container images easy—perhaps too easy, leading to bloated images, wasted storage, and slower startup times with much duplication between layers. They explain why and suggest solutions.

Laura Nolan examined complexity in her previous SRE column and takes on reliability this time. We all want our software to be reliable, and Laura explains some of the key features for building reliable software and provides a detailed checklist you can use to help you and your team do so.

Peter Norton tells us that it's past time to move on to Python 3. Then Peter explains a way to add type checking to Python, both why (if you don't already know) and how it can be done in Python 3.

Dave Josephsen shares some tricks he learned about anomaly detection from Monitorama, and explains how you can use Prometheus's query language (PromQL) to do this yourselves.

Mac McEniry decided it was time for us to learn how to access databases with SQL interfaces from within Go programs. As usual, you can do this fairly simply by using preexisting Go modules, but you still need to understand SQL.

Dan Geer and Brian Wade consider the question: are your Internet-facing hosts more secure on-premises or in the cloud? Using data acquired from a vendor, they provide an intriguing answer.

Robert G. Ferrell considers layers. Applications are layered over libraries and the operating system, and the network consists of some number of layers—just how many and what you name them depends on how you slice things.

Mark Lamourine has written reviews of an older book about continuous delivery and two books on deep learning. I review Neal Stephenson's *Fall*.

I really don't intend to come across like a Luddite. I just hope to remind people who write user-interfacing code that your users will likely not be members of your peer group. Instead, they may be average people interested, even anxious, to partake in the wonderful technology you have created. Perhaps now is the time for a newer, more natural, interface metaphor, or your potential users may be using just one middle finger with which to salute your newest creation.

References

[1] R. Farrow, Musings, *;login:*, vol. 24, no. 4, August 1998, pp. 59–61: http://rikfarrow.com/farrow_aug98.pdf.

[2] A. Whitten and J. D. Tygar, "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0," in *Proceedings of the 8th USENIX Security Symposium (USENIX Security '99)*, USENIX Association, pp. 169–184: https://people.eecs.berkeley.edu/~tygar/papers/Why_Johnny_Cant_Encrypt/USENIX.pdf.

[3] P. Doerfler, "Something You Have and Someone You Know—Designing for Interpersonal Security," Engima 2019: https://www.usenix.org/conference/enigma2019/presentation/doerfler.

[4] The Forensic Challenge: http://old.honeynet.org/challenge/index.html.

Save the Dates!

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. Paper submissions are due Thursday, September 26, 2019.

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 will focus 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. Fall paper titles and abstracts are due Thursday, September 12, 2019.

The full program and registration will be available in December.

Outsourcing Everyday Jobs to Thousands of Cloud Functions with gg

SADJAD FOULADI, FRANCISCO ROMERO, DAN ITER, QIAN LI, ALEX OZDEMIR, SHUVO CHATTERJEE, MATEI ZAHARIA, CHRISTOS KOZYRAKIS, AND KEITH WINSTEIN

We introduce gg, a framework that helps people execute everyday applications—software compilation, unit tests, video encoding, or object recognition—using thousands of parallel threads on a "serverless" platform to achieve near-interactive completion times. We envision a future where instead of running these tasks on a laptop, or keeping a warm cluster running in the cloud, users push a button that spawns 10,000 parallel cloud functions to execute a large job in a few seconds from start. gg is designed to make this practical and easy.

A third of a century ago, interactive personal computing changed the way the computers were used and markedly increased global productivity. Nevertheless, even today, many applications remain far from interactive: compiling a large software package can take hours; processing an hour of 4K video typically needs more than 30 CPU-hours; and a single frame from the animated movie *Monsters University* takes 29 hours to render [8]. Users who wants to explore or tinker and desire feedback in seconds need to harness thousands of cores in parallel, far exceeding the available compute power in laptops and workstations and leading users towards rented compute resources in large-scale datacenters—the cloud.

However, outsourcing a job to thousands of threads in the cloud presents its own challenges. For one, maintaining a warm cluster of thousands of CPU cores in the form of VMs is not cost-effective for occasional short-lived jobs. Provisioning and booting a cluster of VMs on current commercial services can also take several minutes, leaving end users with no practical option to scale their resource footprint *on demand* in an efficient and scalable manner.

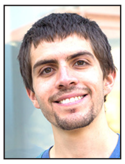
Meanwhile, a new category of cloud-computing resources has emerged that offers finer granularity and lower latency than traditional VMs: cloud functions, also called *serverless computing*. Amazon's Lambda service will rent a Linux container for a minimum of 100 ms, with a startup time of less than a second and no charge when idle. Google, Microsoft, Alibaba, and IBM have similar offerings.

Cloud functions were intended for asynchronously invoked microservices, but their granularity and scale sparked our interest for a different use: as a burstable supercomputer-on-demand. As part of building our massively parallel, low-latency video-processing system, ExCamera [4], we found that thousands of cloud functions can be invoked in a few seconds with inter-function communication over TCP, effectively providing something like a rented 10,000-core computer billed by the second. ExCamera's unorthodox use of a cloud-functions service has been followed by several subsequent systems, including PyWren [6], Sprocket [2], Cirrus, Serverless MapReduce, and Spark-on-Lambda. These systems all launch a *burst-parallel* swarm of thousands of cloud functions, all working on the same job, to provide results to an interactive user.

Challenges of Building Burst-Parallel Applications

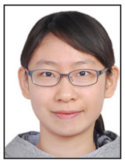
Despite the above, building new burst-parallel applications on thousands of cloud functions has remained a difficult task. Each application must overcome a number of challenges endemic to this environment: (1) workers are stateless and may need to download large

Sadjad Fouladi is a PhD candidate in computer science at Stanford University, working with Keith Winstein on topics in networking, video systems, and distributed computing. His current projects include general-purpose lambda computing and massively parallel ray-tracing systems. sadjad@cs.stanford.edu



Francisco Romero is a PhD student in electrical engineering at Stanford University. His interests are in computer architecture and computer systems. He has recently worked on in-memory database systems for emerging storage technologies, serverless computing, machine learning inference systems, and datacenter resource scheduling. faromero@stanford.edu

Dan Iter is a PhD student at Stanford University. He is advised by Professor Dan Jurafsky and is a member of the NLP Group and AI Lab. He is interested in generative models for text representation, relation extraction, knowledge-base construction, and mental health applications. Previously, Dan also worked on lambda computing and virtualized storage for datacenters. daniter@stanford.edu



Qian Li is a PhD student in computer science at Stanford University, advised by Professor Christos Kozyrakis. She has broad interests in computer systems and architecture. Her current research focuses on efficient resource management and scheduling for heterogeneous cloud computing platforms. Before coming to Stanford, Qian received her Bachelor of Science from Peking University. qianli@cs.stanford.edu

Alex Ozdemir is a PhD student in computer science at Stanford University. His research interests span much of theoretical computer science and computer systems. aozdemir@stanford.edu

Shuvo Chatterjee currently works at Google on account security. Previously, he worked at Square and Apple. In between, he was a visiting researcher at Stanford. His focus is primarily on user security and privacy in large-scale systems. He is a graduate of MIT. shuvo@alum.mit.edu

Matei Zaharia is an Assistant Professor of Computer Science at Stanford University and Chief Technologist at Databricks. He works on computer systems for data analysis, machine learning, and security as part of the Stanford DAWN lab. Previously, Matei started the Apache Spark project during his PhD at UC Berkeley in 2009 and co-started the Apache Mesos cluster manager. Matei's research work was recognized through the 2014 ACM Doctoral Dissertation Award for the best PhD dissertation in computer science, an NSF CAREER Award, the VMware Systems Research Award, and best paper awards at several conferences. matei@cs.stanford.edu

Christos Kozyrakis is a Professor in the Departments of Electrical Engineering and Computer Science at Stanford University. His research interests include resource-efficient cloud computing, energy-efficient computing and memory systems for emerging workloads, and scalable operating systems. Kozyrakis has a PhD in computer science from the University of California, Berkeley. He is a Fellow of the IEEE and ACM. kozyraki@stanford.edu

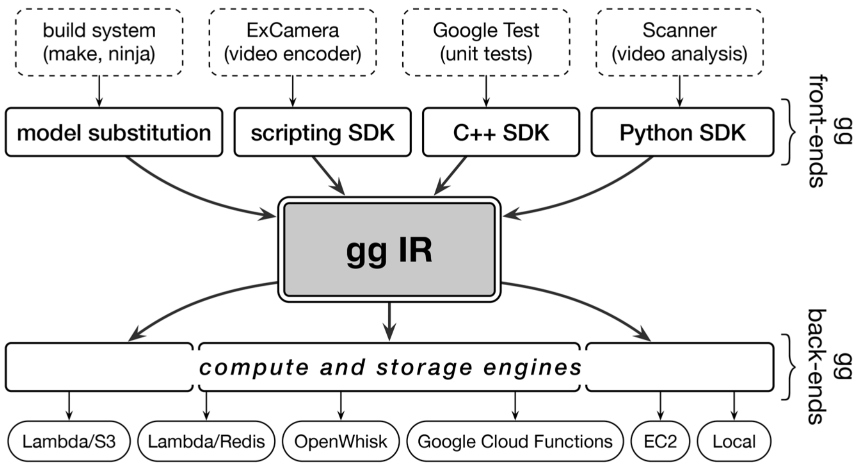


Figure 1: gg helps applications express their jobs in an intermediate representation that abstract the application logic from its placement, schedule, and execution, and provides back-end engines to execute the job on different cloud-computing platforms.

amounts of code and data on startup; (2) workers have limited runtime before they are killed; (3) on-worker storage is limited but much faster than off-worker storage; (4) the number of available cloud workers depends on the provider's overall load and can't be known precisely upfront; (5) worker failures are more likely to occur when running at large scale; (6) libraries and dependencies differ in a cloud function compared with a local machine; and (7) latency to the cloud makes roundtrips costly.

In this article, we present gg, a general system designed to help application developers manage the challenges of creating burst-parallel cloud-function applications. Instead of directly targeting a cloud-functions infrastructure, application developers express their jobs in gg's *intermediate representation* (gg IR), which abstracts the application logic from its placement, schedule, and execution. This portable representation allows gg to run the same application on a variety of compute and storage platforms, and provides runtime features that address underlying challenges, such as dependency management, straggler mitigation, placement, and memoization. Figure 1 illustrates the overall architecture of gg.

gg can containerize and execute existing programs, e.g., software compilation, unit tests, video encoding, or searching a movie with an object-recognition kernel. gg does this with thousands-way parallelism on short-lived cloud functions. In some cases, this yields considerable benefits in terms of performance. For example, compiling the Inkscape graphics editor on AWS Lambda using gg was almost 5x faster than an existing system (icecc) running on a 384-core cluster of warm VMs.

```
{ function: {
    hash: 'VDSo_TM',
    args: [ 'gcc', -E', 'hello.c', '-o', 'hello.i' ],
    envars: [ 'LANG=us_US' ] },
  objects: [ 'VLb1SuN=hello.c', 'VDSo_TM=gcc', 'VAs.BnH=cpp', 'VB33fCB=/usr/stdio.h' ],
  outputs: [ 'hello.i' ] }
                                                              thunk hash: T0MEiRL
```

Figure 2: An example thunk for preprocessing a C program, hello.c. The thunk is named by the hash of its content, T0MEiRL. The hash starts with T to mark it as a thunk rather than a primitive value. Other thunks can refer to its output by using this hash.

Keith Winstein is an Assistant Professor of Computer Science (and, by courtesy, of Electrical Engineering) at Stanford University. He and his students and colleagues made the Mosh (mobile shell) tool, the Mahimahi network emulator, the Sprout and Remy systems for computer-generated congestion control, the Lepton functional-compression tool used at Dropbox, the ExCamera, Salsify, and Puffer systems for video coding and transmission, the Pantheon of Congestion Control, and gg. keithw@cs.stanford.edu

Thunks: Transient Functional Containers

The heart of gg IR is an abstraction that we call a *thunk*. In the functional-programming literature, a thunk is a parameterless closure that captures a snapshot of its arguments and environment for later evaluation. The process of evaluating the thunk—applying the function to its arguments and saving the result—is called *forcing* it [1].

Building on this concept, gg represents a thunk with a description of a container that identifies, in content-addressed manner, an x86-64 Linux executable and all of its input data objects. The container is hermetically sealed and meant to be referentially transparent; it is not allowed to use the network or access unlisted objects or files. The thunk also describes the arguments and environment for the executable and a list of tagged output files that it will generate—the results of forcing the thunk. Figure 2 shows an example thunk for preprocessing a C source file. Since the thunk captures the full functional footprint of a function, it can be executed in any environment capable of running an x86-64 Linux executable.

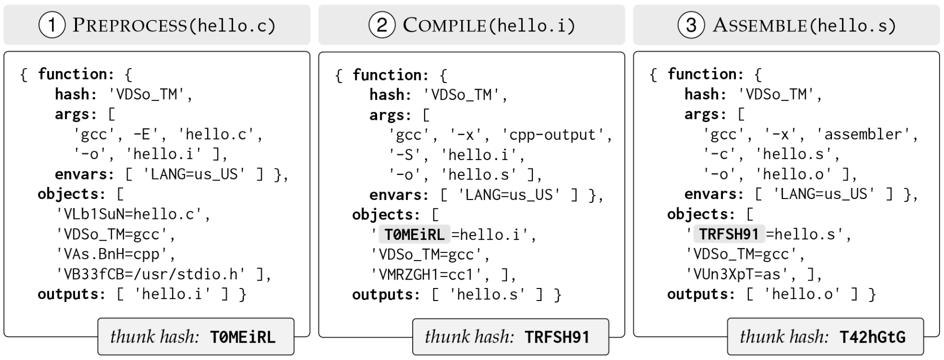
All the objects, including the input files, functions, and thunks are named by their hashes. More precisely, the name of an object has four components: (1) whether the object is a primitive value (hash starting with V) or refers to the result of forcing some other thunk (hash starting with T), (2) a SHA-256 hash of the value's or thunk's content, (3) the length in bytes, and (4) an optional tag that names an object or a thunk's output.

Because the objects are content-addressed, they can be stored on any mechanism capable of producing a blob that has the correct name: durable or ephemeral storage (e.g., S3, Redis, or Bigtable), a network transfer from another node, or by finding the object already available in RAM from a previous execution.

From our experiences of working with the system, we expect gg thunks to be simple to implement and reason about, straightforward to execute, and well matched to the statelessness and unreliability of cloud functions.

gg IR: A Lazily Evaluated Lambda Expression

The structure of interdependent thunks—essentially a lambda expression—is what defines the gg IR. This representation exposes the computation graph to the execution engine, along with the identities and sizes of objects that need to be communicated between thunks. For example, the IR representing the expression `Assemble(Compile(Preprocess(hello.c)))` consists of three thunks, as depicted in Figure 3. Each stage refers to the previous stage's output by using the thunk's hash.



① PREPROCESS(hello.c)

```
{ function: {
    hash: 'VDSo_TM',
    args: [
      'gcc', -E', 'hello.c',
      '-o', 'hello.i' ],
    envars: [ 'LANG=us_US' ] },
  objects: [
    'VLb1SuN=hello.c',
    'VDSo_TM=gcc',
    'VAs.BnH=cpp',
    'VB33fCB=/usr/stdio.h' ],
  outputs: [ 'hello.i' ] }
```

thunk hash: **T0MEiRL**

② COMPILE(hello.i)

```
{ function: {
    hash: 'VDSo_TM',
    args: [
      'gcc', '-x', 'cpp-output',
      '-S', 'hello.i',
      '-o', 'hello.s' ],
    envars: [ 'LANG=us_US' ] },
  objects: [
    ' T0MEiRL =hello.i',
    'VDSo_TM=gcc',
    'VMRZGH1=cc1', ],
  outputs: [ 'hello.s' ] }
```

thunk hash: **TRFSH91**

③ ASSEMBLE(hello.s)

```
{ function: {
    hash: 'VDSo_TM',
    args: [
      'gcc', '-x', 'assembler',
      '-c', 'hello.s',
      '-o', 'hello.o' ],
    envars: [ 'LANG=us_US' ] },
  objects: [
    ' TRFSH91 =hello.s',
    'VDSo_TM=gcc',
    'VUn3XpT=as', ],
  outputs: [ 'hello.o' ] }
```

thunk hash: **T42hGtG**

Figure 3: An example of gg IR consisting of three thunks for building a "Hello, World!" program that represents the expression `Assemble(Compile(Preprocess(hello.c)))` → `hello.o`. To produce the final output `hello.o`, thunks must be forced in order from left to right. Other thunks, such as the link operation, can reference the last thunk's output using its hash, `T42hGtG`. Hashes have been shortened for display.

The IR allows gg to schedule jobs efficiently, mitigate the effect of stragglers by invoking multiple concurrent thunks on the critical path, recover from failures by forcing a thunk a second time, and memoize thunks to avoid repetitive work. This is achieved in an application-agnostic, language-agnostic manner. Based on the data exposed by the IR, back ends can schedule the forcing of thunks, place thunks with similar data-dependencies or an output-input relationship on the same physical infrastructure, and manage the storage or transfer of intermediate results, without roundtrips back to the user's own computer.

Front-End Code Generators and Back-End Execution Engines

Front ends are the programs that emit gg IR (Figure 1). Most of the time, we expect the applications to write out thunks by explicitly providing the executable and its dependencies. This can be done through a command-line tool provided by gg (i.e., `gg create-thunk`) or by using the C++ and Python SDKs that expose a thunk abstraction and allow the developer to describe the application in terms of thunks. For one application, software compilation, we developed a technique called *model substitution* that is designed to extract gg IR from an *existing* build system, without actually compiling the software. In the next section, we will describe the details of this technique.

The execution of gg IR is done by the back ends and requires two components: an *execution engine* for forcing the individual thunks, and a content-addressed *storage engine* for storing the named blobs referenced or produced by the thunks. We implemented five compute engines (a local machine, a cluster of warm VMs, AWS Lambda, IBM Cloud Functions, and Google Cloud Functions) and three storage engines (S3, Google Cloud Storage, and Redis).

gg's approach of abstracting front ends from back ends allows the applications and the back-end engines to evolve and improve independently. The developers can focus on building new applications on top of gg abstractions and, at the same time, benefit from the improvements made to the execution back ends. Moreover, special-purpose execution engines can be built to match the unique characteristics of a certain job without changing the IR description of the application.

As an example, our default AWS Lambda/S3 back end invokes a new Lambda for each thunk. Upon completion, a Lambda uploads its outputs to S3 for other workers to download and use. However, for applications like ExCamera that deal with large input/output objects, the roundtrips to S3 can negatively affect the performance. To improve the performance of such applications, we made a "long-lived" AWS Lambda engine, where each worker stays up until the whole job finishes and seeks out new thunks to execute. The execution engine keeps an index of objects present on each worker's local storage and uses that information to place thunks on workers with the most data available, in order to minimize the need to fetch dependencies from the storage back end.

Software Compilation with gg

Software compilation has long been a prime example of non-interactive computing. For instance, compiling the Chromium Web browser, one of the largest open-source projects, takes more than four hours on a 4-core laptop. Many solutions have been developed to leverage warm machines in a local cluster or cloud datacenter (e.g., `distcc` or `icecc`). We developed such an application on top of gg that can outsource a compilation job to thousands of cloud functions.

Build systems are often large and complicated. The application developers have spent a considerable amount of time crafting `Makefiles`, `CMakeLists.txt` files, and `build.ninja` files for their projects, and manually converting them to gg IR is virtually impossible. We developed a technique called model substitution that can automatically extract a gg IR description from an existing build system.

We run the build system with a modified `PATH` so that each stage is replaced with a stub: a *model* program that understands the behavior of the underlying stage well enough so that when the model is invoked in place of the real stage, it can write out a thunk that captures the arguments and data that will be needed in the future; forcing the thunk will then produce the exact output that would have been produced during actual execution. We used this technique to infer gg IR from the existing build systems for several large open-source applications, including

	Local (make)		Distributed (icecc)	**Distributed (gg)**
	1 core	48 cores	48 cores	AWS Lambda
FFmpeg	06m 9s	20s	01m 03s	44s±04s
GIMP	06m 48s	49s	02m 35s	01m 38s±03s
Inkscape	32m 34s	01m 40s	06m 51s	01m 27s ±07s
Chromium	15h 58m 20s	38m 11s	46m 01s	18m 55s ±10s

Table 1: Comparison of cold-cache build times in different scenarios. gg on AWS Lambda is competitive with or faster than using conventional outsourcing (`icecc`) and, in the case of the largest programs, is 2–5× faster. This includes both the time required to generate gg IR from a given repository using model substitution and the time needed to execute the IR.

OpenSSH, the FFmpeg video system, the GIMP image editor, the Inkscape vector graphics editor, and the Chromium browser, with no changes to the original build system or user intervention. Table 1 shows a summary of the results for four open-source projects. gg on AWS Lambda is about 2–5× faster than a conventional tool (icecc) in building medium- and large-sized software packages.

As an example, we will go through the steps of building the FFmpeg video system with gg. First, the user clones the repo and execute ./configure script to generate the Makefiles:

```
sadjad@~$ git clone https://git.ffmpeg.org/ffmpeg.git
sadjad@~$ [install ffmpeg build dependencies]
sadjad@~$ cd ffmpeg
sadjad@~/ffmpeg$ ./configure --disable-doc --disable-x86asm
```

Next, the user runs gg init in the program's root, which will create a directory named gg. This directory will contain the generated thunks and local cache entries:

```
sadjad@~/ffmpeg$ gg init
```

To compile the project with gg, first we need to extract an IR description from the build system, which is done by running the normal build command (make in this case), prefixed by gg infer:

```
sadjad@~/ffmpeg$ gg infer *make -j$(nproc)*
```

This command will execute the underlying build system, but it modifies the PATH so that each stage of the build is replaced with a *model* program, which writes out a thunk for that stage. After the IR generation is done, the build targets are created, but their contents are not what we would normally expect:

```
sadjad@~/ffmpeg$ cat ffmpeg
#!/usr/bin/env gg-force-and-run
Te6aLo5FtpPyyGY.CsF8PHGY5WS61AlmbcUNGA1tG9Cs00000179
```

This is a *placeholder*, and it expresses that the actual ffmpeg binary is the output of the thunk with the hash Te6aLo5F... (the content of this thunk can be inspected by using the gg describe utility). Running this script forces this thunk, replaces itself with the output, and then executes it. The user can also manually force this thunk by using the gg force utility:

```
sadjad@~/ffmpeg$ gg force --jobs *1500* --engine *lambda*
ffmpeg
* Loading the thunks... done (233 ms).
* Uploading 4663 files (81.8 MiB)... done (6985 ms).
  ...
* Downloading output file (16.7 MiB)... done (1131 ms).
```

This command specifies that the user wants to run this job with 1500-way parallelism on AWS Lambda. First, all the necessary input files are uploaded to the storage engine in one shot. Then the program forces all the necessary thunks recursively until obtaining the final result. After the output is downloaded, the ffmpeg binary can be executed, as if it were built on the local machine:

```
sadjad@~/ffmpeg$ ./ffmpeg
ffmpeg version N-94028-gb8f1542dcb Copyright (c) 2000-2019
the FFmpeg developers
```

Unit Testing with gg

Software test suites are another set of applications that can benefit from massive parallelism, as each test is typically a standalone program that can be run in parallel with other tests, with no inter-dependencies. Using gg's C++ SDK, we implemented a tool that can generate gg IR for unit tests written with Google Test, a popular C++ test framework used by projects like LLVM, OpenCV, Chromium, Protocol Buffers, and the VPX video codec library.

For code bases with large numbers of test cases, this can yield major improvements. For example, the VPX video codec library contains more than 7,000 unit tests, which take more than 50 minutes to run on a 4-core machine. Using the massive parallelism available, gg is able to execute all of these test cases in parallel in less than four minutes, with 99% of the test cases finishing within the first 30 seconds. From a developer's point of view, this improves turnaround time and translates into faster discovery of bugs and regressions.

In addition to software compilation and unit testing, we ported a number of other programs to emit gg IR, including an implementation of ExCamera on gg that, unlike the original implementation, supports memoization and failure recovery, an object recognition task with TensorFlow, and a Fibonacci series program that demonstrates gg abilities on handling *dynamic* execution graphs. For the details of these applications, we refer the reader to our USENIX ATC '19 paper [3].

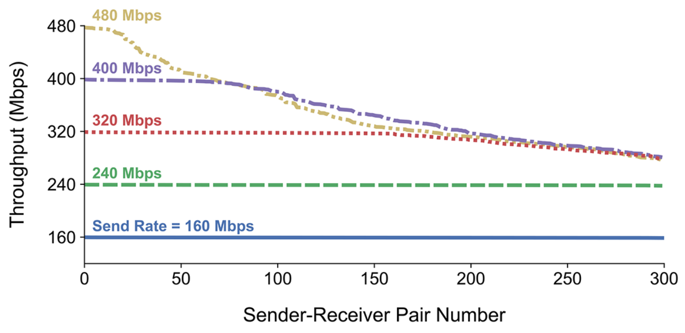


Figure 4: The distribution of achieved network throughputs between pairs of workers at five different send rates on AWS Lambda. Each point corresponds to a sender-receiver pair, and the lines are labeled with their corresponding send rates.

Next Steps: Direct Communication between Workers

Many of the applications that can benefit from *burst-parallel* execution are not *embarrassingly* parallel—they can have complex dataflow graphs and require moving large amounts of data between workers. It has generally been understood that Lambdas cannot accept incoming network connections [5]. As a result, Lambda-computing tools have retreated to exchanging data between workers only indirectly. For example, ExCamera achieves this through TCP connections brokered by a TURN server (each Lambda worker makes an outgoing connection to the server), while PyWren suggests that nodes write to and read from S3, a network blob store. Some of us have developed storage systems like Pocket [7] for ephemeral data storage between workers. However, the latency and throughput limitations introduced by indirect communication (and by mediating inter-node communications through a network file system) are a disqualifier for many applications.

Our preliminary results suggest a more hopeful story for the ability of swarms of cloud functions to tackle communication-heavy workloads, even on current platforms. We have found that on AWS Lambda, workers *can* establish direct connections between one another, and have been able to communicate at up to 600 Mbps using standard NAT-traversal techniques. Figure 4 shows a distribution of achieved network throughputs at five different send rates. For each send rate, we started 600 workers divided into 300 sender-receiver pairs, and each sender transmits UDP datagrams to its pair at that rate for 30 seconds. To be sure, these results indicate variable and unpredictable network performance, but we believe that by designing appropriate protocols and abstractions and failover strategies, direct worker communication can enable a myriad of HPC applications on top of cloud-function platforms.

Our main motivation for this investigation is to build a 3D ray-tracing engine on gg, with the goal of rendering complex scenes with low latency. Currently, the artists who work on 3D scenes rely on high-end machines to iterate on their work—scenes that require tens or sometimes hundreds of gigabytes of memory and take hours to render. Often, the artists must limit the complexity of these scenes (geometry and texture data) by the amount of RAM it is feasible to put in one workstation. For rendering the same scene on RAM-constrained cloud functions, the scene data has to be spread over the workers, which in turn requires low-latency, high-throughput communication between workers to achieve the desired performance. Only further work will tell whether this application can successfully be parallelized to thousands of parallel cloud functions.

Conclusion

We have described gg, a framework that helps developers build and execute burst-parallel applications. gg presents a lightweight, portable abstraction: an intermediate representation (IR) that captures the future execution of a job as a composition of lightweight containers. This lets gg support new and existing applications in various languages that are abstracted from the compute and storage platform and from runtime features that address underlying challenges: dependency management, straggler mitigation, placement, and memoization.

We suspect that cloud functions, as a computing substrate, are in a similar position to that of graphics processing units in the 2000s. At the time, GPUs were designed solely for 3D graphics, but the community gradually recognized that they had become programmable enough to execute some parallel algorithms unrelated to graphics. Over time, this “general-purpose GPU” (GPGPU) movement created systems-support technologies and became a major use of GPUs, especially for physical simulations and deep neural networks.

Cloud functions may tell a similar story. Although intended for asynchronous microservices, we believe that with sufficient effort by the community, the same infrastructure is capable of broad and exciting new applications. Just as GPGPU computing did a decade ago, nontraditional “serverless” computing may have far-reaching effects.

For more information on this project, including our research paper, the code, and quick-start guides, please visit the gg website at https://snr.stanford.edu/gg.

Acknowledgments

We thank the USENIX ATC reviewers and our shepherd, Ed Nightingale, for their helpful comments and suggestions. We are grateful to Geoffrey Voelker, George Porter, Anirudh Sivaraman, Zakir Durumeric, Riad S. Wahby, Liz Izhikevich, and Deepti Raghavan for comments on versions of our research paper. This work was supported by NSF grant CNS-1528197, DARPA grant HR0011-15-20047, and by Google, Huawei, VMware, Dropbox, Facebook, and the Stanford Platform Lab.

References

[1] H. Abelson, G. J. Sussman, with J. Sussman, *Structure and Interpretation of Computer Programs,* 2nd ed. (MIT Press, 1996).

[2] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A Serverless Video Processing Framework," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18),* ACM, pp. 263–274.

[3] S. Fouladi, R. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, "From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers," in *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC '19),* USENIX Association, 2019.

[4] S. Fouladi, R. S. Wahby, B. Shacklett, K.V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17),* USENIX Association, 2017, pp. 363–376.

[5] J. M. Hellerstein, J. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless Computing: One Step Forward, Two Steps Back," in *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research,* 2019.

[6] E. Jonas, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the Cloud: Distributed Computing for the 99%," in *Proceedings of the 8th Symposium on Cloud Computing (SoCC 2017).*

[7] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic Ephemeral Storage for Serverless Analytics," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)* USENIX Association, 2018, pp. 427–444.

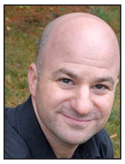
[8] D. Takahashi, "How Pixar Made Monsters University, Its Latest Technological Marvel," VentureBeat, December 2018: https://venturebeat.com/2013/04/24/the-making-of-pixars-latest-technological-marvel-monsters-university/.

Not So Fast
Analyzing the Performance of WebAssembly vs. Native Code

ABHINAV JANGDA, BOBBY POWERS, EMERY BERGER, AND ARJUN GUHA

Abhinav Jangda is a PhD student in the College of Information and Computer Sciences at the University of Massachusetts Amherst. For his research, Abhinav focuses on designing programming languages and compilers. He loves to write and optimize high performance code in his leisure time.
aabhinav@cs.umass.edu

Bobby Powers is a PhD candidate at the College of Information and Computer Sciences at the University of Massachusetts Amherst (in the PLASMA lab), and he is a Software Engineer at Stripe. His research spans systems and programming languages, with a focus on making existing software more efficient, more secure, and usable in new contexts.
bobbypowers@gmail.com



Emery Berger is a Professor in the College of Information and Computer Sciences at the University of Massachusetts Amherst, where he co-directs the PLASMA lab (Programming Languages and Systems at Massachusetts), and he is a regular visiting researcher at Microsoft Research, where he is currently on sabbatical. His research interests span programming languages and systems, with a focus on systems that transparently increase performance, security, and reliability.
emery@cs.umass.edu

WebAssembly is a new low-level programming language, supported by all major browsers, that complements JavaScript and is designed to provide performance parity with native code. We developed Browsix-Wasm, a "UNIX kernel in a web page" that works on unmodified browsers and supports programs compiled to WebAssembly. Using Browsix-Wasm, we ran the SPEC CPU benchmarks in the browser and investigated the performance of WebAssembly in detail.

Web browsers have become the most popular platform for running user-facing applications, and, until recently, JavaScript was the only programming language supported by all major web browsers. Beyond its many quirks and pitfalls from the perspective of programming language design, JavaScript is also notoriously difficult to execute efficiently. Programs written in JavaScript typically run significantly slower than their native counterparts.

There have been several attempts at running native code in the browser instead of JavaScript. ActiveX was the earliest technology to do so, but it was only supported in Internet Explorer and required users to trust that ActiveX plugins were not malicious. Native Client [2] and Portable Native Client [3] introduced a sandbox for native code and LLVM bitcode, respectively, but were only supported in Chrome.

Recently, a group of browser vendors jointly developed the WebAssembly (Wasm) standard [4]. WebAssembly is a low-level, statically typed language that does not require garbage collection and supports interoperability with JavaScript. WebAssembly's goal is to serve as a portable compiler target that can run in a browser. To this end, WebAssembly is designed not only to sandbox untrusted code, but to be fast to compile, fast to run, and portable across browsers and architectures.

WebAssembly is now supported by all major browsers and has been swiftly adopted as a back end for several programming languages, including C, C++, Rust, Go, and several others. A major goal of WebAssembly is to be faster than JavaScript. For example, initial results showed that when C programs are compiled to WebAssembly instead of JavaScript, they run 34% faster in Chrome [4]. Moreover, on a suite of 24 C program benchmarks that were compiled to WebAssembly, seven were less than 10% slower than native code, and almost all were less than twice as slow as native code. We recently re-ran these benchmarks and found that WebAssembly's performance had improved further: now 13 out of 24 benchmarks are less than 10% slower than native code.

These results appear promising, but they beg the question: are these 24 benchmarks really representative of WebAssembly's intended use cases?

The Challenge of Benchmarking WebAssembly

The 24 aforementioned benchmarks are from the PolybenchC benchmark suite [5], which is designed to measure the effect of polyhedral loop optimizations in compilers. Accordingly, they constitute a suite of small scientific computing kernels rather than full-fledged

Arjun Guha is an Assistant Professor in the College of Information and Computer Sciences at the University of Massachusetts Amherst, where he co-directs the PLASMA lab (Programming Languages and Systems at Massachusetts). His research interests include web programming, web security, network configuration languages, and system configuration languages.
arjunguha@umass.edu

applications. In fact, each benchmark is roughly 100 lines of C code. WebAssembly is meant to accelerate scientific kernels, but it is explicitly designed for a wider variety of applications. The WebAssembly documentation highlights several intended use cases, including scientific kernels, image editing, video editing, image recognition, scientific visualization, simulations, programming language interpreters, virtual machines, and POSIX applications. In other words, WebAssembly's solid performance on scientific kernels does not imply that it will also perform well on other kinds of applications.

We believe that a more comprehensive evaluation of WebAssembly should use established benchmarks with a diverse collection of large programs. The SPEC CPU benchmarks meet this criterion, and several of the SPEC benchmarks fall under WebAssembly's intended use cases. For example, there are eight scientific applications, two image and video processing applications, and all the benchmarks are POSIX applications.

Unfortunately, it is not always straightforward to compile a native program to WebAssembly. Native programs, including the SPEC CPU benchmarks, require operating system services, such as a file system, synchronous I/O, processes, and so on, which WebAssembly does not itself provide.

Despite its name, WebAssembly is explicitly designed to run in a wide variety of environments, not just the web browser. To this end, the WebAssembly specification imposes very few requirements on the execution environment. A WebAssembly module can import externally defined functions, including functions that are written in other languages (e.g., JavaScript). However, the WebAssembly specification neither prescribes how such imports work, nor prescribes a standard library that should be available to all WebAssembly programs.

There is a separate standard [7] that defines a JavaScript API to WebAssembly that is supported by all major browsers. This API lets JavaScript load and run a Wasm module, and allows JavaScript and Wasm functions to call each other. In fact, the only way to run WebAssembly in the browser is via this API, so all WebAssembly programs require at least a modicum of JavaScript to start. Using this API, a WebAssembly program can rely on JavaScript for I/O operations, including drawing to the DOM, making networking requests, and so on. However, this API also does not prescribe a standard library.

Emscripten [6] is the de facto standard toolchain for compiling C/C++ applications to WebAssembly. The Emscripten runtime system, which is a combination of JavaScript and WebAssembly, implements a handful of straightforward system calls, but it does not scale up to larger applications. For example, the default Emscripten file system (MEMFS) loads the entire file-system image in memory before execution. For the SPEC benchmarks, the file system is too large to fit into memory. The SPEC benchmarking harness itself requires a file system, a shell, the ability to spawn processes, and other UNIX facilities, none of which Emscripten provides.

Most programmers overcome these limitations by modifying their code to avoid or mimic missing operating system services. Modifying well-known benchmarks, such as SPEC CPU, would not only be time-consuming but would also pose a serious threat to the validity of any obtained results.

Our Contributions

To address these challenges, we developed Browsix-Wasm, which is a simulated UNIX-compatible kernel for the browser. Browsix-Wasm is written in JavaScript (compiled from TypeScript) and provides a range of operating system services to Wasm programs, including processes, files, pipes, and blocking I/O. We have engineered Browsix-Wasm to be fast, which is necessary both for usability and for benchmarking results to be valid [1].

Using Browsix-Wasm, we conducted the first comprehensive performance analysis of WebAssembly using the SPEC CPU benchmark suite (both 2006 and 2017). This evaluation confirms that Wasm is faster than JavaScript (1.3× faster on average). However, contrary to prior work, we found a substantial gap between WebAssembly and native performance. Code compiled to Wasm ran on average 1.55× slower in Chrome and 1.45× slower in Firefox.

Digging deeper, we conducted a forensic analysis of these results with the aid of CPU performance counters to identify the root causes of this performance gap. For example, we found that Wasm produced code with more loads and stores, more branches, and more L1 cache misses than native code. It is clear that some of the issues that we identified can be addressed with engineering effort. However, we also identified more fundamental performance problems that appeared to arise from the design of WebAssembly, which will be harder to address. We provided guidance to help WebAssembly implementers focus their optimization efforts in order to close the performance gap between WebAssembly and native code.

In the rest of this article, we present the design and implementation of Browsix-Wasm and give an overview of our experimental results. This article is based on a conference paper that appeared at the 2019 USENIX Annual Technical Conference, which presents Browsix-Wasm, our experiments, our analysis, and related work in detail [1].

Overview of Browsix-Wasm

Browsix-Wasm mimics a UNIX kernel within a web page with no changes or extensions needed to a browser. Browsix-Wasm supports multiple processes, pipes, and the file system. At a high-level, the majority of the kernel, which is written in JavaScript, runs on the main thread of the page, whereas each WebAssembly process runs within a WebWorker, which runs concurrently with the main thread. In addition, each WebWorker also runs a small amount of JavaScript that is necessary to start the WebAssembly process and to manage process-to-kernel communication for system calls.

In an ordinary operating system, the kernel has direct access to each process's memory, which makes it straightforward to transfer data to and from a process (e.g., to read and write files). Web browsers allow a web page to share a block of memory between the main thread and WebWorkers using the SharedArrayBuffer API. In principle, a natural way to build Browsix-Wasm would be to have each WebAssembly process share its memory with the kernel as a SharedArrayBuffer.

Unfortunately, there are several issues with this approach. First, a SharedArrayBuffer cannot be grown, which precludes programs from growing the heap on demand. Second, browsers impose hard memory limits on each JavaScript thread (2.2 GB in Chrome), and thus the total memory available to Browsix-Wasm would be 2.2 GB across all processes. Finally, the most fundamental problem is that WebAssembly programs cannot access SharedArrayBuffer objects.

Instead, Browsix-Wasm adopts a different approach. Within each WebWorker, Browsix-Wasm creates a small (64 MB) SharedArrayBuffer that it shares with the kernel. When a system call references strings or buffers in the process's heap (e.g., writev or stat), the runtime system copies data from the process memory to the shared buffer and sends a message to the kernel with locations of the copied data in auxiliary memory. Similarly, when a system call writes data to the auxiliary buffer (e.g., read), its runtime system copies the data from the shared buffer to the process memory at the memory specified. Moreover, if a system call specifies a buffer in process memory for the kernel to write to (e.g., read), the runtime allocates a corresponding buffer in auxiliary memory and passes it to the kernel. If a system call must transfer more than 64 MB, Browsix-Wasm breaks it up into several operations that only transfer 64 MB of data. The cost of these memory copy operations is dwarfed by the overall cost of the system call invocation, which involves sending a message between process and kernel JavaScript contexts.

Using Browsix-Wasm, we are able to run the SPEC benchmarks and the SPEC benchmarking harness unmodified within the browser. The only portions of our toolchain that work outside the browser are (1) capturing performance counter data, which cannot be done within a browser, and (2) validating benchmark results, which we do outside the browser to avoid errors.

Performance Evaluation

Browsix-Wasm provided what we needed to compile the SPEC benchmarks to WebAssembly, run them in the browser, and collect performance counter data. We ran all benchmarks on a 6-Core Intel Xeon E5-1650 v3 CPU with hyperthreading and 64 GB of RAM. We used Google Chrome 74.0 and Mozilla Firefox 66.0. Our ATC paper describes the experimental setup and evaluation methodology in more detail.

Reproducing Results with PolybenchC

Although our goal was to conduct a performance evaluation with the SPEC benchmarks, we also sought to reproduce the results by Haas et al. [4] that used PolybenchC. We were able to run these benchmarks (which do not make system calls): the most recent implementations of WebAssembly are now faster than they were two years ago.

Measuring the Cost of Browsix-Wasm

It is important to rule out the possibility that any slowdown that we report is due to poor performance by the Browsix-Wasm

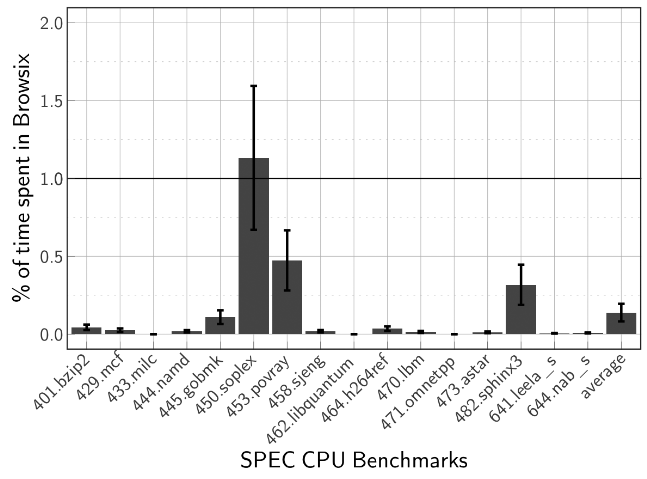


Figure 1: Percentage of time spent (in %) in Browsix-Wasm calls in Firefox

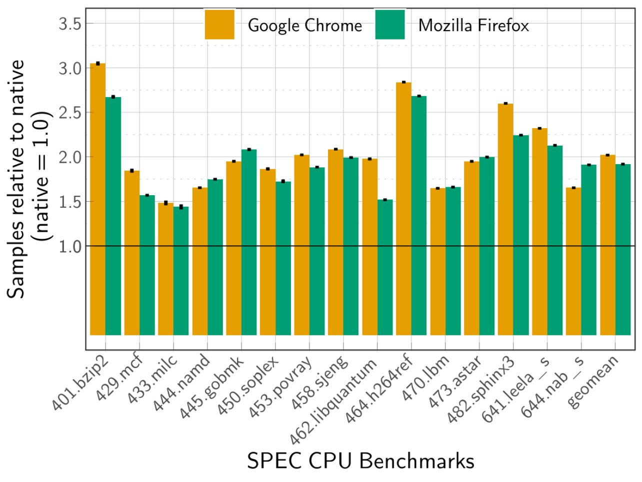


Figure 2: Ratio of the number of load instructions retired by WebAssembly over native code

kernel. In particular, since Browsix-Wasm implements system calls without modifying the browser, and system calls involve copying data, there is a risk that a benchmark may spend the majority of its time copying data in the kernel. Fortunately, our measurements indicate that this is not the case. Figure 1 shows the percentage of time spent in the kernel on Firefox when running the SPEC benchmarks. On average, each SPEC benchmark only spends 0.2% of its time in the kernel (the maximum is 1.2%); we conclude that the cost of Browsix-Wasm is negligible.

Benchmark	Native	Google Chrome	Mozilla Firefox
401.bzip2	370	864	730
429.mcf	221	180	184
433.milc	375	369	378
444.namd	271	369	373
445.gobmk	352	537	549
450.soplex	179	265	238
453.povray	110	275	229
458.sjeng	358	602	580
462.libquantum	330	444	385
464.h264ref	389	807	733
470.lbm	209	248	249
473.astar	299	474	408
482.sphinx3	381	834	713
641.leela	466	825	717
644.nab_s	2476	3639	3829
Slowdown:geomean	—	1.55x	1.45x
Slowdown:jmedian	—	1.53x	1.54x

Table 1: Detailed breakdown of SPEC CPU benchmarks execution times (of 5 runs) for native (Clang) and WebAssembly (Chrome and Firefox); all times are in seconds.

Measuring the Performance of WebAssembly Using SPEC

Finally, we are ready to consider the performance of the SPEC suite of benchmarks. Specifically, we used the C/C++ benchmarks from SPEC CPU2006 and SPEC CPU2017 (the new C/C++ benchmarks and the speed benchmarks). These benchmarks use system calls extensively and do not run without the support of Browsix-Wasm. We were forced to exclude four benchmarks that either failed to compile with Emscripten or allocated more memory than WebAssembly allows in the browser.

In Table 1 we show the absolute execution times of the SPEC benchmarks when running in Chrome, Firefox, and natively. All benchmarks are slower in WebAssembly, with the exception of 429.mcf and 433.milc, which actually run faster in the browser. Our ATC paper presents a theory of why this is the case. Nonetheless, most benchmarks are slower when compiled to WebAssembly: the median slowdown is nearly 1.5× in both Chrome and Firefox, which is considerably slower than the median slowdowns for PolybenchC. In our ATC paper, we also compare the performance of WebAssembly and JavaScript (asm.js) using these benchmarks, and confirm that WebAssembly is faster than JavaScript.

Explaining Why the SPEC Benchmarks Are Slower with WebAssembly

Using CPU performance counters, our ATC paper explores in detail why the SPEC benchmarks are so much slower when compiled to WebAssembly. We summarize a few observations below.

Register pressure. For each benchmark and browser, Figure 2 shows the ratio of the number of load instructions retired by WebAssembly over native code. On average, Chrome and Firefox retire 2.02× and 1.92× as many load instructions as native code, respectively. We find similar results for store instructions

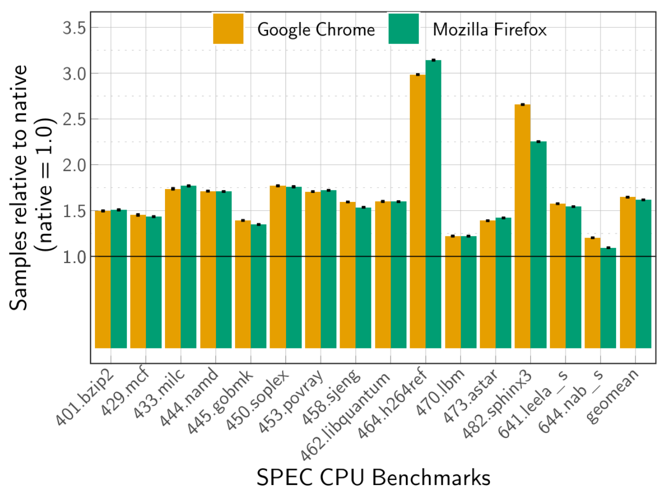


Figure 3: Ratio of the number of conditional branch instructions retired by WebAssembly over native code

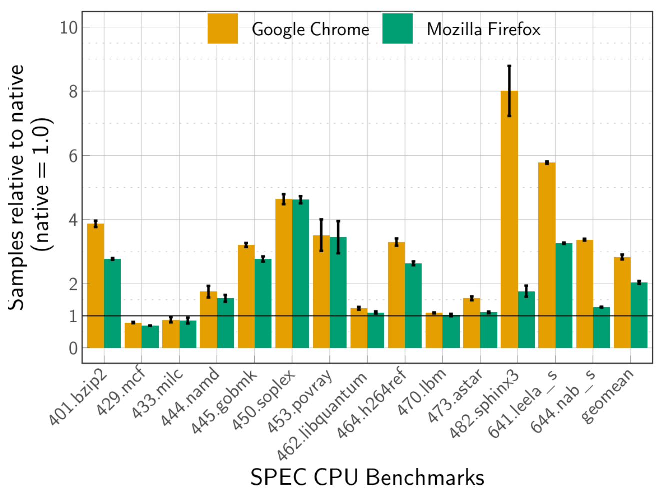


Figure 4: Ratio of the number of L1 instruction cache misses by WebAssembly over native code

retired. Our paper presents two reasons why this occurs. First, we find that Clang's register allocator is better than the register allocator in Chrome and Firefox. However, Chrome and Firefox have faster register allocators, which is an important tradeoff. Second, JavaScript implementations in Chrome and Firefox reserve a few registers for their own use, and these reserved registers are not available for WebAssembly either.

Extra branch instructions. Figure 3 shows the ratio of the number of conditional branch instructions retired by WebAssembly over native code. On average, both Chrome and Firefox retire 1.7× more conditional branches. We find similar results for the number of unconditional branches too. There are several reasons why WebAssembly produces more branches than native code, and some of them appear to be fundamental to the way the language is designed. For example, a WebAssembly implementation must dynamically ensure that programs do not overflow the operating system stack. Implementing this check requires a branch at the start of each function call. Similarly, WebAssembly's indirect function call instruction includes the expected function type. For safety, a WebAssembly implementation must dynamically ensure that the actual type of the function is the same as the expected type, which requires extra branch instructions for each indirect function call.

More cache misses. Due to the factors listed above, and several others, the native code produced by WebAssembly can be considerably larger than equivalent native code produced by Clang. This has several effects that we measured using CPU performance counters. For example, Figure 4 shows that WebAssembly suffers 2.83× and 2.04× more cache misses with the L1 instruction cache. Since the instruction cache miss rate is higher, the CPU requires more time to fetch and execute instructions, which we also measure in our paper.

Conclusion

We built Browsix-Wasm, a UNIX-compatible kernel that runs in a web page with no changes to web browsers. Browsix-Wasm supports multiple processes compiled to WebAssembly. Using Browsix-Wasm, we built a benchmarking framework for WebAssembly, which we used to conduct the first comprehensive performance analysis of WebAssembly using the SPEC CPU benchmark suite (both 2006 and 2017). This evaluation confirms that Wasm is faster than JavaScript. However, we found that WebAssembly can be significantly slower than native code. We investigated why this performance gap exists and provided guidance for future optimization efforts. Browsix-Wasm has been integrated into Browsix; both Browsix and Browsix-SPEC can be found at https://browsix.org.

Acknowledgments

Browsix-Wasm builds on earlier work by Powers, Vilk, and Berger (Powers and Berger are co-authors of this article). That work did not support WebAssembly and had performance issues that Browsix-Wasm addresses. This work was partially supported by NSF grants 1439008 and 1413985.

References

[1] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code," in *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '19):* https://www.usenix.org/conference/atc19/presentation/jangda.

[2] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," 30th IEEE Symposium on Security and Privacy (Oakland '09), *Communications of the ACM*, vol. 53, no. 1, January 2010, pp. 91–99.

[3] A. Donovan, R. Muth, B. Chen, and D. Sehr, "PNaCl: Portable Native Client Executables," 2010: https://css.csail.mit.edu/6.858/2012/readings/pnacl.pdf.

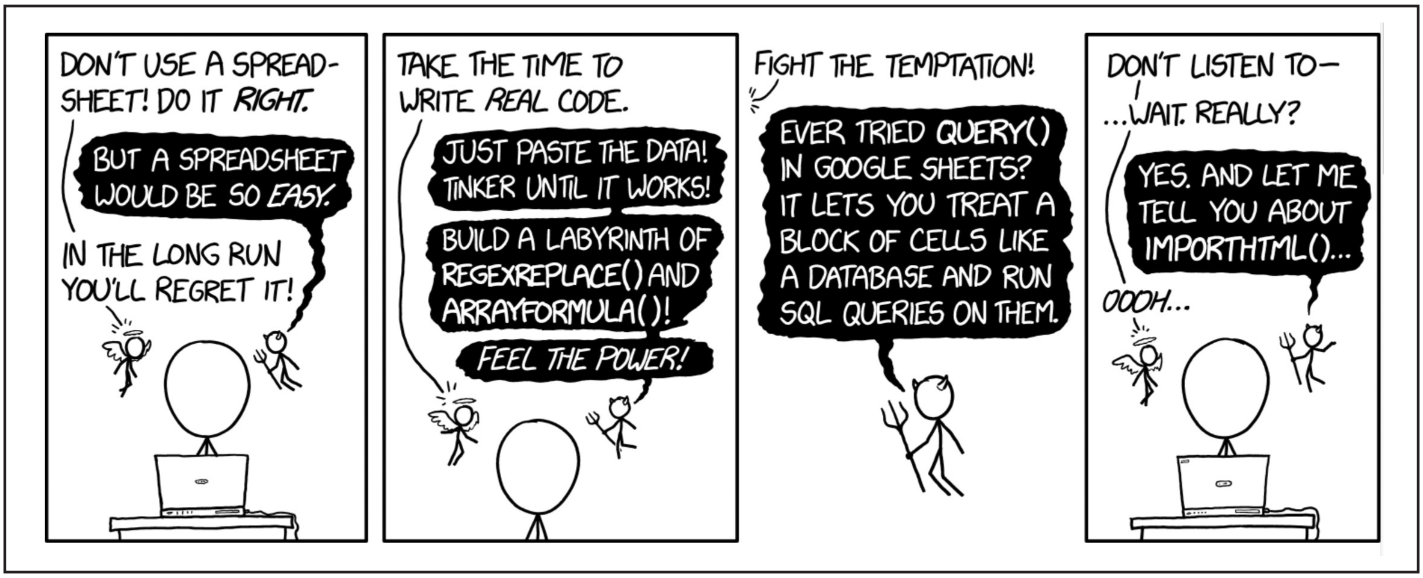
[4] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien, "Bringing the Web Up to Speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*, ACM, 2017, pp. 185–200.

[5] PolyBenchC: The Polyhedral Benchmark Suite, 2012: http://web.cs.ucla.edu/~pouchet/software/polybench/.

[6] A. Zakai, "Emscripten: An LLVM-to-JavaScript Compiler," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA '11)*, ACM, 2011, pp. 301–312.

[7] WebAssembly JavaScript Interface, 2019: http://webassembly.github.io/spec/js-api/index.html.

XKCD **xkcd.com**


DON'T USE A SPREADSHEET! DO IT RIGHT.
BUT A SPREADSHEET WOULD BE SO EASY.
IN THE LONG RUN YOU'LL REGRET IT!
TAKE THE TIME TO WRITE REAL CODE.
JUST PASTE THE DATA! TINKER UNTIL IT WORKS!
BUILD A LABYRINTH OF REGEXREPLACE() AND ARRAYFORMULA()!
FEEL THE POWER!
FIGHT THE TEMPTATION!
EVER TRIED QUERY() IN GOOGLE SHEETS? IT LETS YOU TREAT A BLOCK OF CELLS LIKE A DATABASE AND RUN SQL QUERIES ON THEM.
DON'T LISTEN TO—
...WAIT. REALLY?
YES. AND LET ME TELL YOU ABOUT IMPORTHTML()...
OOOH...

ENIGMA

A USENIX CONFERENCE

SECURITY AND PRIVACY IDEAS THAT MATTER

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

Enigma is committed to fostering an open, collaborative, and respectful environment. Enigma and USENIX are also dedicated to open science and open conversations, and all talk media is available to the public after the conference.

PROGRAM CO-CHAIRS

Ben Adida
VotingWorks

Daniela Oliveira
University of Florida

The full program and registration will be available in November.

enigma.usenix.org

JAN 27–29, 2020

SAN FRANCISCO, CA, USA

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

SECURITY

Making It Easier to Encrypt Your Emails

JOHN S. KOH, STEVEN M. BELLOVIN, AND JASON NIEH

John S. Koh is a PhD candidate in computer science at Columbia University. John's interests lie in the intersection of applied cryptography, usability, and systems security with practicality in mind. koh@cs.columbia.edu

Steven M. Bellovin is Professor of Computer Science at Columbia University and affiliate faculty at its law school. His research specializes in security, privacy, and related legal and policy issues. He co-authored *Firewalls and Internet Security*, the first book on the subject. He is a member of the National Academy of Engineering and received the USENIX "Flame" award for co-inventing Netnews. smb@cs.columbia.edu

Jason Nieh is Professor of Computer Science and Co-Director of the Software Systems Laboratory at Columbia University. Professor Nieh has made research contributions in software systems across a broad range of areas, including operating systems, virtualization, thin-client computing, cloud computing, mobile computing, multimedia, web technologies, and performance evaluation. nieh@cs.columbia.edu

We've known for decades how difficult it is to encrypt email. We've developed E3, a client-side system that encrypts email at rest on mail servers to mitigate the most common cases of attacks today. E3 also demonstrates techniques for making key management simple enough for most users, including those who use email on multiple devices.

Email privacy is of crucial importance. Although email accounts and servers contain troves of valuable private information dating back years, they are easy to compromise. This makes them attractive targets for adversaries. Attackers often use methods such as spear-phishing, password recovery and reset, and social engineering attacks to obtain a victim's email credentials. With login details in hand, attackers then simply authenticate to the appropriate mail service like a normal user and siphon off all of the victim's emails.

We have seen this situation repeatedly in the news such as with the John Podesta, Sarah Palin, and John Brennan email hacks, among many more. Email encryption using a key inaccessible to the email service provider would have mitigated all these attacks. But none of these victims used encrypted email. If even prominent VIPs with access to top-notch advice are failing to use any kind of encrypted email, then everyday non-technical users are very unlikely to adopt email encryption. What makes this even worse is that a single breach of this kind is enough to compromise the entire history of affected users' emails. With the explosive growth of cloud storage, it is easy to keep gigabytes of old emails at no cost forever.

Existing email encryption approaches are comprehensive and effective against attackers but are seldom used due to their complexity and inconvenience. Examples include Pretty Good Privacy (PGP) [1] and Secure/Multipurpose Internet Mail Extensions (S/MIME), which are end-to-end encrypted email solutions. They are frankly too complicated to use, yet they represent the state of the art for secure email. The current paradigm for secure email places too much of a burden on its users, especially senders of email, who must correctly encrypt emails, manage keys, understand public key cryptography, and coordinate with other potentially non-technical users [5, 6]. The result is even technical users rarely encrypt their email.

End-to-end encrypted email is overkill for most users. Mail services are increasingly using SSL/TLS for email in transit between SMTP and IMAP servers, and are forcing clients to use SSL/TLS or STARTTLS. One example is Google's Gmail service, which completely disables plain IMAP and therefore requires clients to use TLS connections. This makes a large part of end-to-end encryption's benefits redundant since emails are already being encrypted in transit. What users are vulnerable to is an adversary who steals email account credentials, such as via a database leak or a phishing attack, or who compromises entire mail servers, such as when governments issue subpoenas for and seize entire servers belonging to mail services. But end-to-end encryption for email protects against a vast array of rarely encountered attacks other than these. This comes at the cost of usability, creating a chasm between end-to-end encryption's absolute security, which almost nobody uses, and regular plaintext email with no encryption, which everybody uses. There is thus room for change.

Designing for End Users

Any new secure email solution needs to be easy to use and also platform independent to help make it as amenable as possible to users. This has historically been a difficult problem. Various approaches, both academic and commercial, have tried to make it easier to use secure email but at the cost of sacrificing platform independence. They only work within closed ecosystems, such as Lavabit and Posteo, or with other people using the same solution, such as with traditional end-to-end encrypted email. But perhaps even more importantly, the more widely used secure mail services often encrypt emails or users' individual private keys on their servers using master private keys accessible to them.

What we need is a secure email solution that works on any mail service (yes, even Gmail) and that uses a private key that is inaccessible to the mail service but is accessible to all of a user's multiple devices for reading email. At the same time, users shouldn't need to know about key management concepts, public key cryptography, and public key infrastructure (PKI). Just as important is that this solution must work nearly identically to a regular email client to minimize the learning curve.

We developed Easy Email Encryption (E3) [4] as the first step to filling the void between unusable but secure email encryption and usable but insecure plaintext email. E3 provides a client-side encrypt-on-receipt mechanism that makes it easy for users since they do not need to rely on PKI or coordinate with recipients. The onus is no longer on the sender to figure out how to use PGP or S/MIME. Instead, email clients automatically encrypt received email without user intervention. E3 protects all emails received prior to any email account or server compromise for the emails' lifetime, using threat models similar to those of more complex schemes such as PGP and S/MIME.

E3 is designed to be compatible with existing IMAP servers and IMAP clients to ease adoption. No changes to any IMAP servers are necessary. Users require only a single E3 client program to perform the encryption, but multiple E3 clients are supported as well. Existing mail clients do not need to be modified and can be used as is alongside a separate E3 background app or add-on. If desired, existing mail clients can be retrofitted with E3 instead of relying on a separate app or on an add-on.

Users are free to use their existing, unmodified mail clients to read E3-encrypted email if they support standard encrypted email formats. The vast majority of email clients support encrypted emails either natively or via add-ons. Other than the added security benefits of encryption, all functionality looks and feels the same as a typical email client, including spam filtering and having robust client-side search capability.

Key management, including key recovery, is simplified by a scheme we call per-device key (PDK) management, which provides significant benefits for the common email use case of having two or more devices for accessing email, e.g., desktop and mobile device mail clients. Users with multiple devices leverage PDK with no reliance on external services. Users who truly only use a single device still benefit from PDK's key configuration and management capabilities but rely on free and reliable cloud storage for recovery. E3 as a whole is a usable solution for encrypted email that protects a user's history of emails while also providing a simple platform-independent key management scheme.

Encrypt on Receipt

Encrypt on receipt can be described as follows: when a user's E3 client detects that the mail server has received a new plaintext email, it downloads it, encrypts it, and replaces the original plaintext email with the encrypted version. In practice this is implemented entirely on the client side through the use of several existing IMAP commands, so E3 requires no modifications to the IMAP server and protocol. The encryption format is either standard PGP or S/MIME depending on implementation preference. Encrypt on receipt confers many benefits for usability while still retaining important security properties.

Self-generated, self-signed key pairs. Since the user isn't sending encrypted email but simply storing it for himself, the key pairs used for encrypting, decrypting, and signing don't need to be trusted by others. The user doesn't need to know about PKI and complicated key exchanges with other confused users. Self-generated and self-signed key pairs are also useful for E3's key management approach.

Support for all IMAP services. Encrypt on receipt is compatible with any IMAP mail service with no server modifications, including Gmail, Yahoo!, AOL, Yandex, and so on. It is also compatible with server-side spam filters, anti-virus scanners, and even indexing for ad-based services since emails are encrypted after they are received, giving the server a window of time to process email before it is encrypted.

Client implementation and compatibility. Encrypt on receipt requires only modest implementation changes for existing IMAP mail clients. We implemented E3 on multiple platforms, including on a popular open-source Android mail client, K-9 Mail, to show this. Furthermore, since E3 uses standard encrypted email formats, emails can be read on any unmodified mail client that supports them. Examples for S/MIME include Apple Mail, Mozilla Thunderbird, and Microsoft Outlook.

Secure against future compromises. Since all emails are encrypted on receipt, they remain secure against any future compromise of a user's account or IMAP server. To the attacker, all old emails would be encrypted and therefore unusable. However, if the attacker retains access to the account, newly arriving emails will be vulnerable. Encrypt on receipt therefore represents a much better-than-nothing approach to security. The current norm for email security is no security, so protecting a user's thousands of old emails is much better than protecting absolutely none of her emails.

Security against wiretapping. Encrypt on receipt is not end-to-end encryption, so email is not sent in encrypted form. This is actually not an issue. These days, especially after the Snowden revelations of widespread government surveillance of the Internet, practically all mail services use TLS for both client-server and server-server connections to protect email in transit.

Users don't need to know crypto. The user doesn't manually encrypt email because the client handles all encryption and decryption automatically. This is an issue observed in user studies, including our own—sometimes users can't figure out that they need to press the encrypt button when sending encrypted email to others.

Crypto algorithms can be updated. Once in a blue moon, a crypto algorithm or key length is discovered to have problems or simply has become too weak. Re-encrypting E3 emails to a newer crypto standard is simple: re-encrypt all emails using the user's new key. In contrast, this situation poses a problem for traditional PGP and S/MIME because re-encrypting emails received from other PGP or S/MIME users may not be possible. Perhaps the original sender can no longer be reached to re-sign the new copy, and thus his signature data would be lost. Even if he were reachable, the process of asking someone else to sign your old emails is a tedious task and also requires expert knowledge from all participants of how end-to-end encryption works.

Per-Device Keys

E3 eliminates manual public key exchanges. This simplifies the key management by removing half of it. What remains is the problem of private keys when using multiple devices. Traditional security best practices advise users to never transport private keys because doing so is insecure. This advice is almost never followed in practice because users often access email from multiple devices, all of which need the same private key when using common secure email usage models.

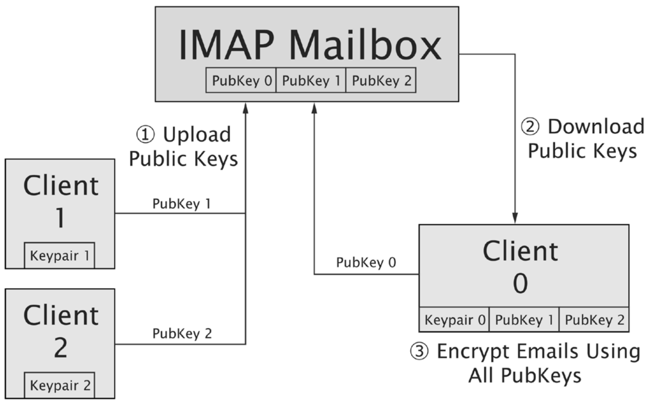


Figure 1: The per-device key (PDK) architecture

E3 returns to the traditional security advice of never transporting private keys. In contrast to most secure email schemes, which assume a user has a single private key, E3 asserts that a user should have a unique private key for every device. Then each device makes its public key available to the others. We call this the per-device key (PDK) scheme as depicted in Figure 1. PDK provides numerous benefits compared to traditional end-to-end encrypted email:

Complements self-generated, self-signed keys. One of the strengths of encrypt on receipt is that it can leverage self-generated, self-signed keys because it does not need to worry about third-party trust. PDK complements this scheme because each of a user's devices can generate its own self-signed key pair. This also greatly simplifies the process of adding a new device to a user's E3 ecosystem since it can just generate its own key pair.

Avoids moving private keys around. As we mentioned, traditional security best practices advise users to never move private keys around. Not only is this insecure, but most users have no concept of what a private key is and how it differs from its public key. With PDK, users don't need to know about these concepts and only know that their devices are encrypting their emails for them.

Eliminates manual public key exchanges. Instead of moving private keys, each of the user's E3 clients automatically makes available its public key to his other devices. Then any E3 client can encrypt the user's emails using the public keys from all of his devices. The principle is similar to when a traditional PGP or S/MIME user encrypts an email to multiple people. The email is not encrypted multiple times for each public key but is encrypted only once using a symmetric key, which in turn is encrypted to each public key. E3 takes this paradigm and applies it in a new way by encrypting emails on receipt using every verified public key belonging to the user. When a new key is added, clients re-encrypt already-encrypted emails to the new keys.

Requires no secondary communication channel. E3 maintains its requirement for platform independence even for its public-key exchanges. E3 clients upload their public keys to the mailbox as ordinary emails with the keys as attachments. Other E3 clients detect these key emails and store the public keys locally. These key emails contain a number of metadata fields to identify them but also to ensure security: for example, to support a secure key verification process.

Turns complicated key verification into simple device verification. In traditional end-to-end encrypted email, users must verify public keys, usually via trusted third parties. This places a burden on non-technical users who don't understand PKI. PDK also asks users to verify keys, but since E3 has different trust requirements compared to traditional end-to-end encrypted email, verifying PDK public keys is a much simpler process. Verifying a PDK public key means checking whether that key really belongs to one of the user's devices. E3 presents this as asking the user to verify whether she is adding a new device. Ideally, the method to do this should be compatible with any kind of device whether a desktop or mobile one. We therefore developed a process, which we refer to as a *two-way verification process*. A given client periodically scans for new keys, and when a new key is detected, the user is prompted to perform the two-way verification step.

The two-way verification process leverages a verification phrase that is easy for humans to recognize and match. When a client uploads its key for other devices to discover, it adds a randomly generated verification phrase to the key email, which is prominently displayed. The user then needs to confirm this verification phrase on one of his existing E3 clients. Once he completes the verification on any existing client, it will display a second verification phrase. The user then needs to confirm this second phrase on his new client to complete the two-way verification.

The catch is that when the user confirms a verification phrase, it must be selected from among two randomly generated incorrect phrases. The user must select the correct verification phrase in order to verify the key. This multiple-choice confirmation reduces the chances of a user accidentally accepting a key that isn't hers. The words in the phrases are selected from a curated pool such as the PGP Word List [3]. As shown in [2], this technique is effective and usable for quickly authenticating identities even with only three words.

Conclusion

Easy Email Encryption (E3) introduces new client-side encrypt-on-receipt and per-device key (PDK) mechanisms compatible with the existing IMAP standard and servers. E3 email clients automatically encrypt received email without user intervention, making it easy for users to protect the confidentiality of all emails received prior to any email account or server compromise. E3 uses keys that are self-generated and self-signed, and PDK makes it easy to use them to access encrypted email across multiple devices. Users no longer need to understand or rely on public key infrastructure, coordinate with recipients, or figure out how to use PGP or S/MIME.

E3 is also easy to implement, and we developed versions of it on a variety of platforms, including Android, Windows, Linux, and even Google Chrome. We also ensured that it works with popular IMAP-based email services, including Gmail, Yahoo!, AOL, and Yandex. Further, we conducted a user study to evaluate E3 usability, and results show that real users, even non-technical ones, consider E3 easy to use even when compared to using regular unencrypted email clients and vastly easier to use over the state of the art for PGP.

Twenty years ago, Whitten and Tygar's "Why Johnny Can't Encrypt" introduced Johnny to the research community as a representation of the average non-technical user who finds end-to-end encrypted email impossibly difficult to use [6]. However, we have seen an explosive growth of consumer-oriented technology since then. Always-on, always-connected mobile devices are ubiquitous, providing the necessary foundation for putting a new and usable spin on the idea of receiver-controlled encryption. Johnny may have been unable to encrypt, but Joanie in the modern age certainly can.

References

[1] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer, *OpenPGP Message Format*, IETF, RFC 4880, November 2007: http://www.rfc-editor.org/rfc/rfc4880.txt.

[2] M. Farb, Y.-H. Lin, T. H.-J. Kim, J. McCune, and A. Perrig, "SafeSlinger: Easy-to-Use and Secure Public-Key Exchange," in *Proceedings of the 19th Annual International Conference on Mobile Computing & Networking (MobiCom '13)*, ACM, pp. 417–428: https://doi.org/10.1145/2500423.2500428.

[3] P. Juola and P. Zimmermann, "Whole-Word Phonetic Distances and the PGPfone Alphabet," in *Proceedings of the 4th International Conference on Spoken Language Processing (ICSLP '96)*, vol. 1, IEEE, pp. 98–101: https://doi.org/10.1109/ICSLP.1996.607046.

[4] J. S. Koh, S. M. Bellovin, and J. Nieh, "Why Joanie Can Encrypt: Easy Email Encryption with Easy Key Management," in *Proceedings of the 14th EuroSys Conference 2019 (EuroSys '19)*, ACM, 2019, article no. 2: https://doi.org/10.1145/3302424.3303980.

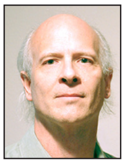
[5] S. Ruoti, N. Kim, B. Burgon, T. Van Der Horst, and K. Seamons, "Confused Johnny: When Automatic Encryption Leads to Confusion and Mistakes," in *Proceedings of the 9th Symposium on Usable Privacy and Security (SOUPS 2013)*, ACM, article no. 5.

[6] A. Whitten and J. D. Tygar, "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0," in *Proceedings of the 8th USENIX Security Symposium (USENIX Security '99)*, USENIX Association, pp. 169–184: https://people.eecs.berkeley.edu/~tygar/papers/Why_Johnny_Cant_Encrypt/USENIX.pdf.

Interview with Periwinkle Doerfler

RIK FARROW

Periwinkle Doerfler is a PhD candidate at New York University's Tandon School of Engineering within the Center for Cybersecurity, advised by Professor Damon McCoy. Her research focuses on the intersection of intimate partner violence and technology. She looks at this issue with regard to abusers and how they come to use technology to perpetuate violence, as well as with regard to survivors and how technology can help or hinder escape from abusive situations. Her past work has also examined cryptocurrency as it relates to human trafficking, doxing communities, and authentication schemes.
periwinkleid@gmail.com


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

I met Peri Doerfler at Enigma 2019 during lunch and wanted to talk to her right away. Peri would be giving the closing talk the next day about interpersonal threats, a very different way of looking at security than any I had considered. In my life, the threats were attacks on my mail or web servers, or disclosure of financial information while I was attending USENIX conferences. Peri was taking on what sounded completely different, but also very relevant to the types of technology people are regularly using today.

I also have a very personal interest in Peri's research topic. All of the women I'd become close to during my life had told me stories of sexual abuse. I don't mean just verbal abuse, but actual assaults or rape. I was and still am astonished and appalled. The current statistics, relying on reported attacks, are one in three women and one in six men in the US have been sexually assaulted [1].

Rik Farrow: To start out with, how did you get interested in the interpersonal threat area? Reading online, I noticed that you interned at Google and worked on authentication issues.

Peri Doerfler: I've actually had a pretty varied set of research experiences that led me to this. The first project I got involved in when I started my PhD involved looking at Bitcoin and human trafficking, and as you noted, I interned at Google and worked on authentication. I had a second internship at Google working on Android permissions.

In doing some work on spyware and domestic violence, I found that there is a whole set of threats that people, but particularly women, are facing from the people they know. I have not continued to be heavily involved with the work that group at Cornell Tech (in NYC) is doing related to domestic violence, but they are doing great work, as are a few other groups, including one at Google. I think where I went from caring about the specific work to more of this vigilante attitude, if you will, is in attending conferences and hearing a lot of the community dismiss these concerns. I'm always frustrated to hear the security and privacy community talk about users as though they are stupid.

Further, I find that when you address what are, to be frank, more female concerns (not at all because men don't face the same technological concerns, but because men tend to have less fear of physical violence), they are even more summarily dismissed. I have often heard people express how "sad it is that that happens to some people" when discussing domestic violence, without realizing that it is such a common problem (transcending socioeconomic barriers, I must add) that it very likely affects someone they know well. So for me, I think that the best way to help the users who are not aware of the risks they may be taking by sharing their iPhone PIN (or similar) is to raise awareness in society at large, but also to try to get the community that controls this technology and its default settings to think about these risks as seriously as they think about risks from hackers and phishers.

RF: Speaking of which, how do you go about researching such sensitive areas? Do you rely on mining public comments? Are you gaining a reputation in this area so that people seek you out?

PD: When studying domestic violence specifically, a lot of good work is already done in collaboration with various governmental and nongovernmental agencies working with survivors. Most of the research on survivors is done from interviews at shelters. In my personal work in that space, I've focused more on studying the abusers and trying to understand how they're acquiring the awareness and know-how to become abusive with smartphones. That work relies on public information in reviews of apps, on Reddit and 4chan, and on the websites and advertising of the software makers.

In studying interpersonal privacy more generally, I think it will be a combination of the two methods. There's honestly not a lot of data out there now about things like password sharing even generally, and especially not specifically in relationships. I'm definitely hoping to gather some in future work.

RF: Let's stick with spyware for the moment. In March 2019, Eva Galperin of the EFF said she was going to speak about "eradicating spyware" at a Kaspersky conference [2]. The story itself is decent, and it relates to your work.

After the conference, Kaspersky Lab announced adding a feature to their Android AV product that pops up warnings, "Privacy Alerts," when it appears spyware is in use, allowing the user to block the theft of information [3]. I would think that helping the person delete the spyware app would be a better idea.

PD: Yes, the *Wired* story [2] does reference some of my work. I think Eva's coming from exactly the same place on this as I am, which is wanting to help in every way possible and being frustrated when others aren't as receptive as they could be. I liked this quote from the article:

> "...often because security researchers don't count spy tools that require full access to a device as 'real' hacking, despite domestic abusers in controlling relationships having exactly that sort of physical access to a partner's phone.

I think she makes another really good point about threat modeling, and that for the average smartphone user, the major threats the security industry tends to focus on don't really hold up:

> The Kaspersky users who worry about domestic abuser spying are rarely the same ones concerned with Russian intelligence. "It's really about modeling your threat. Most victims of domestic violence don't work for the NSA or the US government."

With regards to whether Kaspersky's move is enough, my response is a resounding no. The fact of the matter is that for it to help someone, they have to have Kaspersky antivirus on their phone before the spyware is installed, then whoever installs the spyware has to not know that it's there or not know how to tamper with the antivirus. Further, it appears from the *Wired* article that this feature is going to operate off of a blacklist. A lot of these apps have many, many versions with different hashes, and a blacklist is likely to miss them.

It's also not clear whether this blacklist will include dual-use apps coming from the Play Store. Assuming this chain of events, the victim gets this privacy notification, but the notification isn't as specific as it could be. It's better than the previous "not a virus" warning, but it doesn't articulate the delicacy of the situation, that someone *put this stuff on your phone*, as opposed to it being some awful adware bundled with something else. It doesn't clarify that the information being leaked could be your GPS data, text messages, and recent calls.

And it certainly doesn't do the most important thing in this context, which would be to help the victim understand that if they delete the offending application, the abuser may become aware of that and escalate to physical violence. That's the big problem I could see happening: in the case it does catch something, people are going to remove it without realizing what it was, and then potentially face violence as a repercussion or lose any evidence they may have had.

I will note, however, that Kaspersky has also reached out to me to ask for thoughts/guidance on how to improve this feature, and they have a whole team of people making a genuine effort to address this. That's incredibly reassuring to see, but it's frustrating that the scope of the protection will be limited to their customers. Hopefully, it puts pressure on other industry players to do the same.

RF: In your Enigma talk, you tell the story of someone being embarrassed after allowing someone access to the iPad to play some music. While phones typically autolock, lots of other devices, like iPads and laptops, don't. To be honest, I think of my home as my castle, but it's really not. I have guests sometimes, or workers, in the house. But in your area of interest, it's not the guests that are the problem, correct?

PD: In my research, guests and workers are part of the threat model, though they are less likely to be the source of a threat than a parent, coworker, or intimate partner. I'm generally interested in studying the ways that people perceive their digital privacy and security in relation to the people they know "IRL." Shared devices and accounts are increasingly ubiquitous, so I'm interested in questions ranging from "Do people moderate their viewing habits when sharing their parents' Netflix account?" to "To what extent do people share their devices with their partners, and what are their expectations of their partners' access to their device?"

RF: What are your plans for future work?

PD: One of the next studies I want to do is with respect to online dating, and asking a few questions inspired by true and very creepy anecdotes. First, if you're in a fairly self-contained community, like a college campus, how easily can you find someone on a dating app if you've only seen them, say, in class? What risks does this pose? Second, if you encounter someone on a dating app, how easy is it to find them elsewhere online or IRL? How does this change across apps, geographic density? Beyond studying dating apps, I'm hoping to do a deeper dive on device sharing and credential sharing in romantic relationships.

I'm also still working on some research related to doxing and harassment, as well as trying to understand pieces of the incel/pickup-artist space, and what the connection is between that and domestic violence.

References

[1] National Sexual Violence Resource Center statistics: https://www.nsvrc.org/node/4737.

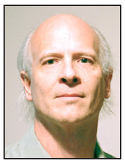
[2] A. Greenberg, "Hacker Eva Galperin Has a Plan to Eradicate Stalkerware," *Wired*, April 2, 2019: https://www.wired.com/story/eva-galperin-stalkerware-kaspersky-antivirus/.

[3] S. Lyngaas, "Kaspersky Lab Looks to Combat 'Stalkerware' with New Android Feature," Cyberscoop, April 3, 2019: https://www.cyberscoop.com/kaspersky-lab-looks-combat-stalkerware-new-android-feature/.

Interview with Dave Dittrich

RIK FARROW

Dave Dittrich is one of those rare people who started college declaring a major in photography and graphic arts but left with a computer science degree and went on to be involved in many "first in the world" cyber events. His incident response experiences, often involving personally identifiable information of both innocents and suspected computer criminals, led him to research the ethical and legal bounds within which "white hat" actors can justifiably act to respond to "black hat" hackers and criminals. He has written extensively on ethics and the "Active Response Continuum," served for six years on a University of Washington Institutional Review Board, and has recently been distilling this all into curricular resources for teaching practical ethical analysis. dave.dittrich@gmail.com



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

I first met Dave Dittrich at USENIX Security in 2000. Dave had been working at University of Washington for many years by then and had made a name for himself with his analysis of malware installed on Internet-connected systems at the university.

I had learned about his work on distributed hacking tools, particularly the ones for carrying out distributed denial of service (DDoS) attacks. Someone within the NSA had kindly pointed me in that direction, and I had fortunately realized the potential impact and managed to get an article published days before MafiaBoy set off his big attack.

Rik Farrow: When did you start working in DFIR (Digital Forensics and Incident Response) at the University of Washington, and what was that like?

Dave Dittrich: My start in security came from the system administration side, out of necessity.

After working for a couple of years in the UW Chemistry Department, I took a position as the frontline UNIX workstation support contact for faculty and staff on campus. At the time, I think there was something like 20,000 UNIX workstations and maybe 3–4 times more Windows systems. But Windows didn't have a standard TCP/IP stack, so if a computer was broken into over the Internet, it would be a UNIX system. There were BSD, SunOS 3 and 4, System V, HP/UX, Irix, Digital UNIX, NeXT, and nascent Linux (Red Hat and Debian, mostly). I had to support them all, being the first (and usually only) person that would interface with the faculty and staff, relying on the University Computing Services system administrators and engineers for their experience when I didn't have it.

There would sometimes be dozens or hundreds of compromised systems at any given time, and I tried to help everyone as efficiently as possible. I took everything I learned and put it on my web page, and added it to the two-day R870 system administration course that I inherited from someone who retired right after I came on board [1]. I got bit-image copies of any interesting computer intrusion and got really efficient at forensic analysis using open source tools like Coroner's Toolkit by Dan Farmer and Wietse Venema, following public guidelines by the FBI and DoD, and developing my own investigative and reporting techniques.

RF: I think that UNIX `strings` was one of my favorite tools for a quick look at a suspicious binary. Coroner's Toolkit was amazing.

DD: Yeah, amazingly just using `strings` would be enough to get a pretty good idea from internal prompts, error messages, and system call identifiers of what a simple piece of malware was supposed to do. A disassembly could then provide some more detail. For example, is it a sniffer? A remote access trojan? A rootkit concealment tool? An exploit?

Another really basic technique, but one that I don't see commonly used by forensic analysts, is using file system Modify/Access/Create (MAC) timelines to develop situational awareness about post-intrusion activity. Forensic analysts often search for "known bads" using hash databases, or exclude programs based on "known goods" hash databases, or search for known Windows Registry keys, etc. In other words, looking for things based on simple signatures (often signatures derived by others at different sites). While this might work, it also might take hours of indexing to come up with nothing, especially if the malware is polymorphic or crafted specifically for that victim, meaning nobody else would see the same binary in their generalized threat intelligence telemetry. Or it might find several artifacts from different unrelated intrusions over time, confusing the analyst. Just finding a hash match or a file name match is the start of an analytic process, not the end.

It is really hard to effectively wipe out all possible evidence of compromise of the integrity of a computer system. I think it's safe to say that most intrusions up to the early 2000s had almost no effort spent on advanced concealment and wiping of fingerprints, so to speak. Rootkits were very common (both user level and kernel level) but were usually pretty easy to defeat if you know how the operating system, file system, and network connections behave. But you need to be able to show your work and prove it to a "preponderance of the evidence" in civil cases, and "beyond a reasonable doubt" in a criminal case.

I have found it far quicker and more useful to leverage initial facts (including time and date of suspected malicious activity) to find the directory where malware was initially dropped or where configuration files and/or log files are stored. In situations where there is no enterprise endpoint protection agent in place, a very common situation, you need to "live off the land" in terms of evidence collection. To increase confidence, you then include external sources of evidence to confirm/refute things like clock skew, missing the year or time zone in system log lines, etc.

I developed a forensic analysis and reporting methodology using the tools and techniques described by Farmer and Venema in the notes from their 1999 IBM forensic training event. I described this technique and how to use it in a guide I published later that year, "Basic Steps in Forensic Analysis of UNIX Systems" [2]. I also used this technique in a two-hour "house call" on the University of Washington campus network to quickly get around a kernel-level rootkit on a Linux server, which became the chapter "Omerta" in Mike Schiffman's book *Hacker's Challenge* [3].

The owner of that system was 100% sure his system was not compromised, since the kernel-level rootkit worked so well. I hooked his computer and my laptop up to a hub and showed him the IRC bot traffic coming from a process that wasn't listed in `netstat`, or `ps` output. I then had him run `dd` using `netcat` to pipe the root partition to my laptop, where I used the Coroner's Toolkit to get a MAC timeline and later to extract and analyze deleted file space. Having obtained a bit-image copy of the root partition to preserve any evidence, it only took a short time, while simultaneously copying the other partitions, to identify and disable the rootkit. All of the malicious processes now showed up!

By the next day I had a full understanding of what had happened, identified all of the other systems around the world being used by the group from network traffic and internal log or rootkit configuration files, and reported to all the other victim sites and to CERT/CC.

Farmer and Venema, two of the voices of reason in the forensic arena, published a much more detailed description of the underlying operating system behaviors and file-system functions that preserve evidence in their book, *Forensic Discovery* [4]. They showed in technical detail how, despite file deletions (or even re-installation of the operating system, if you look hard enough!), you can do the same kind of analysis that geologists do to understand the history of a specific location by examining the composition of soil layers, rock or shell inclusions, discontinuities in soil layers, etc. With an understanding of how kernels running programs affect MAC times in each type of file system in use, you can not only make quantifiable conclusions based on interpretation of MAC timelines, but you can demonstrate through experiments using the same kernel and file system that you can reproduce the results to show proof to back up your theory!

If your objective is to support criminal process, this is very important in order to meet an evidentiary standard known as the "Daubert Standard" (Daubert v. Merrell Dow Pharmaceuticals, 509 U.S. 579 (1993)): Federal Rules of Evidence 702 requires that an expert witness should possess the kind of knowledge as found in Farmer and Venema's book, use that knowledge to help the court understand the evidence or determine facts at issue, base their testimony on sufficient facts or data that were the product of reliable principles and methods, and reliably apply those principles and methods to the facts of the case.

RF: Another thing I recall you were involved with was the Honeynet Project (HP). I asked Lance Spitzner to write about the HP in 2002 [5]. How did you get involved in the HP?

DD: My publications and conference talks on sniffers, rootkits, post-intrusion log alteration and concealment, and DDoS handler/agent tools, pre-cursors to today's "botnets," got me an invitation to the Honeynet Project. The first publications we did referenced many of the whitepapers I published on my UW home page.

People kept saying, "It's great that you mention how to use tools and how to analyze compromised systems, but I don't have a honeypot set up and want a bit image disk copy to work with." So Lance Spitzner asked me to organize the Forensic Challenge so that people could have a real-life compromised Linux system to work with. I spent over a hundred hours in one month doing the reference analysis, setting up the rules, organizing the judges, and managing the judging process. It was the top most-popular download on the HP web site for a few years running!

I also organized the Reverse Challenge, which turned out to be yet-another-DDoS-bot!

RF: How did you get involved with working on the Menlo Report? Is that something you and Erin Kenneally decided to do on your own?

DD: Erin and I both came into our roles in the process from previous DHS and ethics (and legal, in Erin's case) work we had done.

I had been working within the PREDICT project (a research data repository project, now known as the Information Marketplace for Policy and Analysis of Cyber-risk and Trust, or IMPACT) at DHS for many years. Around 2006, I was trying to develop honeypot images and related logs and network traffic for use in research. This kind of sandbox processing of malware artifacts is commonplace today, but not in the mid-2000s. One of the larger such botnet-related dataset collections today is maintained by the Czech Technical University in Prague (https://mcfp.felk.cvut.cz/publicDatasets/).

One of the botnets I had studied, known as "Nugache," was written up in USENIX *;login:* in 2007 [6]. Nugache had some features far in advance of the most visible botnet in the world at the time, the "Storm botnet." I was keeping a close eye on Storm and the differences in Nugache that really had me worried due to the level of apparent sophistication in that botnet (that wouldn't be publicly shown to have been surpassed until the Conficker.C variant came out years later).

I saw the December 2008 CCC presentation "Stormfucker: Owning the Storm Botnet" by researchers from the University of Bonn, inspired by research from the University of Mannheim, where they demonstrated a partially tested implementation of software components necessary for constructing a "white worm" that could be released to clean up Storm botnet-infected nodes. Afterwards, I began writing on the ethics of cleaning up botnets. This followed on the Active Response Continuum research I had done, and my take on the ethics was very applied and focused on the overlap of research and operations (including law enforcement investigations), not just a pure academic research perspective.

My first attempt at publication at the USENIX LEET '09 workshop was rejected, but I was invited to participate on a panel entitled "Ethics in Botnet Research" in April 2009 [7].

That initial rejected paper grew and became a technical report co-authored with Michael Bailey (a PREDICT Principal Investigator) and Sven Dietrich (whom I had been working with on Nugache). We released the technical report the same day as the LEET panel [8].

I had several people I knew with ethical review experience review the paper and case studies to see if they would even require research ethical review. One was Katherine Carpenter, with whom I've subsequently written several articles and papers. The other two were Tanya Matthews and Shannon Sewards, who worked at the University of Washington's Institutional Review Board (IRB). I also joined one of UW's IRB committees to learn how the process works from firsthand experience, serving on the committee for over six years.

Doug Maughan was at the LEET panel and invited me to speak about this paper at the first workshop on ethics in ICT research he was setting up for the next month (May 26-27, 2009). That workshop led to formation of the Menlo Working Group. The technical report I co-authored with Bailey and Dietrich was provided to the Working Group and served as some of the background and case studies for the Companion to the Menlo Report.

RF: I believe you wrote about the process. It's enough to say that many people were involved, but you and Erin created the report that got published in the Federal Record.

DD: The process was covered in an *IEEE Security and Privacy* article [9].

We had a large Working Group, approximately two dozen people, a similarly sized group of external reviewers, and a number of official responses to the publication in the Federal Register that had to be integrated and summarized in an official response in the Federal Register. I learned a lot about the Federal Register and its relationship to federal regulations!

Erin Kenneally was serving as legal counsel to CAIDA (another PREDICT performer). Erin and I both had the capacity to wrangle the report drafting and commenting/editing process. We got closer to a final draft ready for submission to the Federal Register and subsequent public response when Michael Bailey joined us to help out with the final push (and to work with us to start publishing and speaking about the Menlo Report as part of the outreach process).

RF: What else were you doing at UW?

DD: A couple years after that, Mike Eisenberg (Dean of the Information School) and David Notkin (chair of the Computer Science and Engineering Department) bought out half of my time to allow me to reach out to other universities and community colleges, get the UW accepted into the National Security Agency's Center of Academic Excellence in Information Assurance Education (CAE IAE) program, help start up the Center for Information Assurance and Cybersecurity (CIAC), and begin a career as a staff research scientist with permission to be a Principal Investigator on grants, despite only having a BS degree. I owe a great deal to Mike Eisenberg.

Over the next 10+ years, I brought in over $4 million in grants and contracts and covered my salary and that of a few others at various times. My first grant was from Cisco Systems Critical Infrastructure Assurance Group to study *active defense*. I coined the term *Active Response Continuum* (ARC) to make it clear this is not a black/white situation by any means but, rather, a set of ranges or levels (capacity to respond, aggressiveness, intrusiveness, risk, etc.). I collaborated with Kenneth Einar Himma to write one of the early papers on the topic (http://ssrn.com/abstract=790585) and presented first publicly at AusCERT 2005.

We came at the subject from the perspective of private-sector response and framed it in terms of ethical principles, as opposed to the military *law of war* context taken by most publications on the topic to date. The concept of ethics in security operations and research has remained a central part of my research and publications since then. The bulk of the funding I secured at the UW was from Doug Maughan (another person to whom I owe a great deal) at DHS but also included grants or contracts from NSF, DoD, the FTC, and industry.

Over that same period I had permission to work on outside contracts and pro bono projects, including contract support to criminal defense lawyers, federal public defenders, assisting a few DDoS victims, assisting the Federal Trade Commission on a fake-drug civil temporary restraining order (TRO) case, and providing declarations to the court in two of Microsoft's ex-parte TROs in the Waledac and Rustock botnet cases.

RF: What are your plans for the future?

DD: I'll be really honest: I'm figuring that out. Let me explain.

During my last major project as a Principal Investigator at the UW, I worked so hard I was burning myself out. Physically, I have a nerve impingement in my neck that began to cause pain, tingling, and numbness in my back, shoulder, and arm. Emotionally, I was taking on too much stress (which combined with the physical issues to produce a negative feedback loop). My doctor, friends and family were all telling me I had to cut back, change my work habits, and take it easier to begin to recover.

I just read Arthur C. Brooks' *Atlantic* piece, "Your Professional Decline Is Coming (Much) Sooner Than You Think: Here's How to Make the Most of It" [10]. His article really spoke to me and made me realize some things that have been in the back of my mind lately.

I've recently been writing a history of the early days of the Honeynet Project, going back over some of the things I did in the late 1990s and early 2000s. This August 19th is the 20th anniversary of the first massive DDoS (handler/agent style) attack on the University of Minnesota that lead me to write the first DDoS tool analyses. It surprised me a little to realize just how much I did in those days (all the DDoS tool analyses I wrote, computer security incidents I investigated and reported on to CERT/CC and the FBI, projects taking up hundreds of hours over a month or two, trips and talks and publications, all on top of a 40+ hour work week). As Brooks describes, I made my name and reputation in this industry using fluid intelligence and a dedication to serving the public through open source research, digital forensics, malware analysis and threat intelligence, and publication. But I am learning (the hard way) how unsustainable that level of productivity really is. I feel confident I can still identify and solve novel "cyber" problems, but physically I can't put in 12+ hour days any longer.

Over the last two years I've started shifting to, as Brooks puts it, using crystallized intelligence—applying all the lessons I've learned in information security over the decades, the recommendations and predictions I've made, all that I have read and researched, the linkages I'm capable of recognizing—as a contract subject matter expert and an author. I have invested thousands of hours in producing open source tools, documenting ways to solve some basic information security problems that have persisted for decades (like default passwords and secrets leaked through source code repositories), and combining case studies and other material from papers I've written and the Menlo Report effort to produce materials for a full-day applied ethics tutorial/course. Ever since my UNIX workstation support days, I have tried to teach what I have learned by including what-to-do and how-to-do-it information in my publications and have given many talks and guest lectures. So perhaps teaching is my future? After all, my father (before he passed away) and my older brother today both taught college physics as professors for decades each.

I'm excited for the next direction my professional and personal life will take, similar to the way I used to feel in my 30s and 40s when preparing to go on a multi-day backcountry ski trip. I know how to navigate finding a route, the general direction I want to go, and I've done all the preparation and accumulated the requisite knowledge. But I don't know right now precisely what path I will take, what kind of objective hazards I will have to overcome, the amazing views from the summits, or what pleasures (and discomforts) I will encounter on the way.

References

[1] "R870: UNIX System Administration—A Survival Course": https://www.washington.edu/R870/cover-page.html.

[2] D. Dittrich, "Basic Steps in Forensic Analysis of UNIX Systems": https://staff.washington.edu/dittrich/misc/forensics/.

[3] M. Schiffman, *Hacker's Challenge* (McGraw-Hill, 2002).

[4] D. Farmer, W. Venema, *Forensic Discovery* (Addison-Wesley Professional, 2005).

[5] L. Spitzner, "HOSUS (Honeypot Surveillance System)," *;login:*, vol. 27, no. 6 (USENIX, December 2002): https://www.usenix.org/system/files/login/articles/1252-spitzner.pdf.

[6] S. Stover, D. Dittrich, J. Hernandez, and S. Dietrich, "Analysis of the Storm and Nugache Trojans: P2P Is Here," *;login:,* vol. 32, no. 6 (USENIX, December 2007): https://www.usenix.org/system/files/login/articles/526-stover.pdf.

[7] J. Brodkin, "The Legal Risks of Ethical Hacking," *Network World*, April 24, 2009: https://www.networkworld.com/article/2268198/the-legal-risks-of-ethical-hacking.html.

[8] D. Dittrich, M. Bailey, S. Dietrich, "Towards Community Standards for Ethical Behavior in Computer Security Research," Technical Report CS 2009-01, Stevens Institute of Technology, April 2009: https://staff.washington.edu/dittrich/papers/dbd2009tr1/dbd2009tr1.pdf.

[9] M. Bailey, D. Dittrich, E. Kenneally, and D. Maughan, "The Menlo Report," *Security and Privacy*, vol. 10, no. 2 (IEEE, March/April 2012), pp. 71–75: http://www.caida.org/publications/papers/2012/menlo_report/menlo_report.pdf.

[10] A. Brooks, "Your Professional Decline Is Coming (Much) Sooner Than You Think: Here's How to Make the Most of It," *The Atlantic*, July 2019: https://www.theatlantic.com/magazine/archive/2019/07/work-peak-professional-decline/590650/.

Save the Dates!

29TH USENIX SECURITY SYMPOSIUM

August 12–14, 2020 • Boston, MA, USA

The 29th USENIX Security Symposium brings together researchers, practitioners, system administrators, system programmers, and others to share and explore the latest advances in the security and privacy of computer systems and networks.

The Symposium will span three days, with a technical program including refereed papers, invited talks, posters, panel discussions, and Birds-of-a-Feather sessions. Co-located workshops will precede the Symposium on August 10 and 11.

Paper submission deadlines:
Fall Quarter: Friday, November 15, 2019
Winter Quarter: Saturday, February 15, 2020

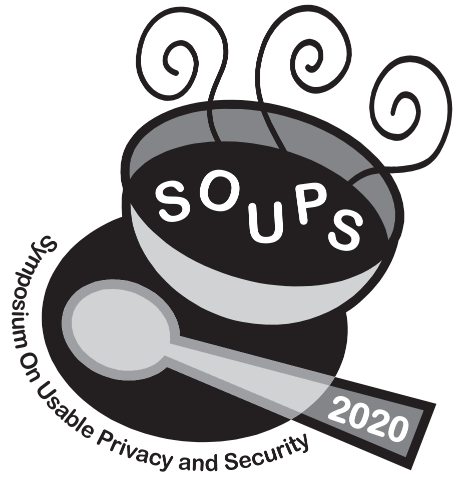
Invited talk and panel proposals deadline:
Friday, February 14, 2020

Program Co-Chairs

Srdjan Capkun
ETH Zurich

Franziska Roesner
University of Washington

www.usenix.org/sec20



Sixteenth Symposium on Usable Privacy and Security

Co-located with USENIX Security ’20
August 9–11, 2020 • Boston, MA, USA

The Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020) will bring together an interdisciplinary group of researchers and practitioners in human computer interaction, security, and privacy. The program will feature technical papers, including replication papers and systematization of knowledge papers, workshops and tutorials, a poster session, and lightning talks.

Symposium Organizers

General Chair
Heather Richter Lipford,
University of North Carolina at Charlotte

Technical Papers Co-Chairs
Michelle Mazurek, *University of Maryland*
Joe Calandrino, *Federal Trade Commission*

www.usenix.org/soups2020

Challenges in Storing Docker Images

ALI ANWAR, LUKAS RUPPRECHT, DIMITRIS SKOURTIS, AND VASILY TARASOV

Ali Anwar is a research staff member at IBM Research–Almaden. He received his PhD in computer science from Virginia Tech. In his earlier years he worked as a tools developer (GNU GDB) at Mentor Graphics. Ali's research interests are in distributed computing systems, cloud storage management, file and storage systems, AI platforms, and the intersection of systems and machine learning. Ali.Anwar2@ibm.com

Lukas Rupprecht is a researcher in the Storage Systems Group at IBM Research–Almaden. His research interests are broadly related to distributed systems for data management, including scalability, performance, fault tolerance, and manageability aspects. He received his PhD from Imperial College London and holds MSc and BSc degrees from Technical University Munich. Lukas.Rupprecht@ibm.com

Dimitris Skourtis is a Researcher at IBM Research–Almaden. Prior to that he worked on resource management and scheduling for ESXi at VMware. He has a PhD in computer science from UC Santa Cruz and a masters in mathematics from the University of St Andrews. His interests include distributed systems, data management, and QoS for modern storage devices. Dimitrios.Skourtis@ibm.com

In this article, we describe the structure of Docker images, how they are managed by Docker clients, and how they are stored at Docker registries. We then present several weaknesses in the current design that can cause Docker images to consume excessive storage capacity, degrade container performance, and create contention on the network and the underlying storage infrastructure. We suggest several improvements to alleviate these problems.

At times it seems surprising that hardware virtualization, established virtual machines (VMs), rather than software containers took precedence in the technology evolution. Indeed, in so many practical use cases, one simply wants to run multiple isolated applications on top of a single kernel instead of emulating an entire operating system. This lightweight approach allows containers to start in a fraction of a second and, compared to VMs, consume much less memory and storage, save CPU cycles, and require only a single OS license.

A number of OS-level virtualization technologies appeared in the early 2000s (e.g., Solaris Zones, Linux-VServer, Virtuozzo), but it was only in 2013, with the advent of Docker, that containerization started its conquest of datacenters, clouds, and human minds. By 2013, the Linux kernel components required for containerization—cgroups and namespaces—were already sufficiently mature to provide reliable resource control and boundary separation. However, what was missing was a user-friendly, practical, and yet flexible way to create, deploy, and manage containers. Docker provided this technology. At its heart are Docker images, which form the basic abstraction for users to operate containers.

Container Images and Their Storage

Docker storage can be roughly split into two main parts: client-side storage of images and image distribution via a central registry. In the following, we describe both of these aspects.

Docker Images and Client-Side Storage

In the majority of today's systems, a running application expects its binaries, libraries, and configuration and data files to be stored and accessed through a file system. Hence, the file system tree is an integral part of an application runtime environment. A container *image*, at its core, can be viewed as a file system tree containing all files required by an application to operate. In a simple image implementation, one could copy the required file system tree to a directory and run a containerized application on top of it. However, when a new instance of the same application needs to be started, a new copy of the entire tree has to be created in order to keep any file changes local to each application instance. This slows down container startups significantly.

As a solution to this problem, Docker employs a copy-on-write (CoW) approach to speed up file system creation for containers. Specifically, similar to the "gold images" concept in VMs, Docker defines images as immutable entities. To create a fully functional—and, in particular, writable—root file system for a container, Docker makes use of technologies such as OverlayFS [2]. OverlayFS can create a logical file system on top of two different directories, also known as *layers*, one of which is designated as *writable* while the other one is *read-only*. When Docker creates a new container, the writable layer is initially empty while the read-only layer contains the file system tree of an immutable image.

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

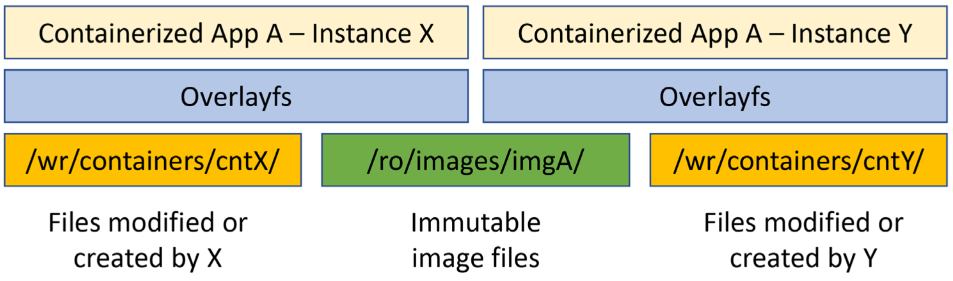

Containerized App A – Instance X
Containerized App A – Instance Y
Overlayfs
Overlayfs
/wr/containers/cntX/
/ro/images/imgA/
/wr/containers/cntY/
Files modified or created by X
Immutable image files
Files modified or created by Y

Figure 1: Two containers X and Y running the application A from the same image

A file in OverlayFS serves as a proxy to either a file in the image (read-only layer) or in the writable layer. For example, reading of a file in OverlayFS is initially redirected to the corresponding file in the read-only layer. However, when an application tries to update a file, OverlayFS seamlessly copies it to the writable layer and updates the file there. After that, all I/O operations to the file go to the copy in the writable layer. In such a design, starting a container is a breeze, as it only requires the creation of an empty writable layer and mounting the OverlayFS. Data copying is performed later and only on demand (copy-on-write). Figure 1 schematically illustrates this setup.

So far, we assumed that immutable container images already exist. But how are they created initially? The capability to easily build images is an important property that makes Docker so attractive. It relies on the ability to convert writable layers to read-only layers and assemble an immutable image from a collection of read-only layers. Figure 2 depicts this organization. In Docker, an image is treated as a stack of read-only layers, where each layer contains the changes, at file granularity, compared to the lower layer. The lowest layer in a stack contains the changes compared to an empty file system. Therefore, every layer can be thought of as a collection of files and directories, and layers belonging to the same image comprise its entire file system tree. OverlayFS is capable of assembling a collection of read-only layers and one writable layer into a single logical file system for a running container. Besides OverlayFS, there are other approaches, which can support the above described storage model of Docker images, e.g., AUFS, device-mapper, or Btrfs. Support for each of these storage back ends is implemented through a *graph driver*.

To create a container image, one can start from an empty container, copy files to its writable layer, and then use the *docker commit* command to convert the writable layer to a read-only layer. As this is tedious, Docker provides the concept of a *Dockerfile* and the *docker build* command for convenience. In this case, Docker creates a temporary build container, updates its root file system using the instructions in the Dockerfile, and commits the writable layer (i.e., converts it to read-only) after every instruction. Images can also be created from previously built images (e.g., an OS distribution). This results in different images sharing layers (see Figure 2).

Registry-Side Storage

For ease of distribution, Docker images are kept in an online store called a *registry*. A registry, such as Docker Hub [1], acts as a storage and content delivery system, holding named Docker images, available in different tagged versions. Figure 3 shows the basic structure of a typical registry and how users interact with it. Users create *repositories*, holding images for a particular application (e.g., Redis or WordPress) or a basic operating system (e.g., Ubuntu or CentOS). Images in a repository can have different versions, identified by *tags*. The combination of a repository name (which typically also includes a user name) and a tag uniquely defines the name of an image.

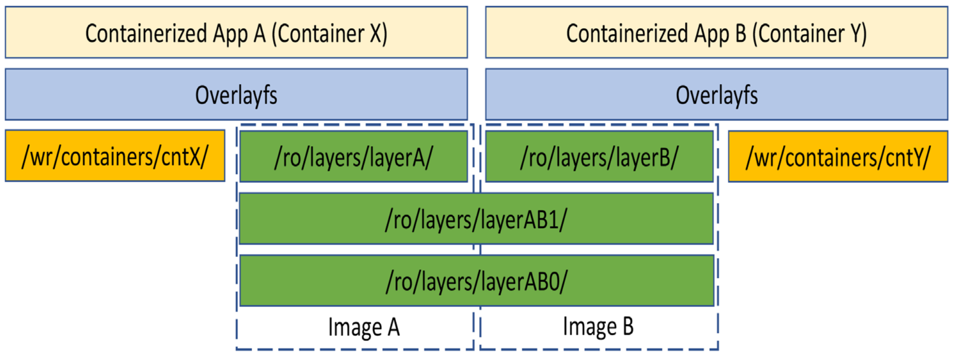


Figure 2: Two applications A and B running in two containers X and Y from two images that share two layers AB0 and AB1 between them

Users can add new images or update existing ones by *pushing* to the registry and retrieve images by *pulling* from the registry. The information about which layers constitute a particular image is kept in a metadata file called a *manifest*. The manifest includes additional image settings such as target hardware architecture, executable to start in a container, and environment variables.

Each layer is stored as a compressed tarball in the registry and has a content-addressable identifier called a *digest*, which uniquely identifies a layer. The digest is a collision-resistant hash of the layer's data (SHA-256 by default). The identifier allows the user to efficiently check whether two layers are identical, share identical layers across different images, and transfer only the missing layers of an image between registries and clients.

Clients communicate with the registry using a RESTful HTTP API. To pull an image from the registry, a Docker client first fetches the image manifest by issuing a `GET` request. Then the client uses the manifest to identify individual layers unavailable in local storage. Finally, the client `GET`s and extracts the missing layers. Pushing works in reverse order compared to pulling.

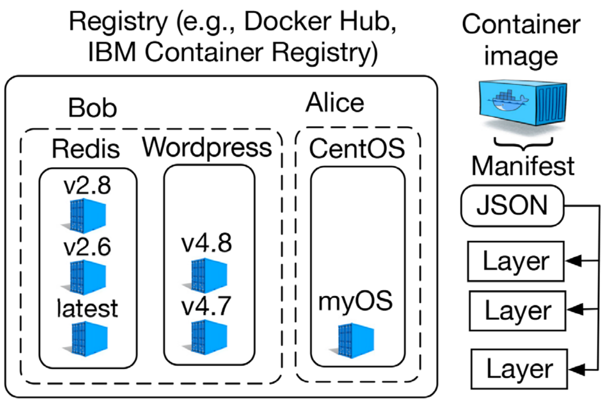


Figure 3: On the left: relationship between registry, users (Bob and Alice), repositories (Redis, WordPress, CentOS), and tagged images (v2.8, latest, v4.8, myOS, etc.). On the right: Docker image structure.

After creating the manifest locally, the client first `PUT`s all the new layers that are not yet stored in the registry, and then `PUT`s the manifest itself.

The existing Docker registry server is a single-node application. To concurrently serve a high-request load, production deployments typically use a load balancer in front of several independent registry instances. All instances store and retrieve images from a shared backend storage. Currently, the Docker registry supports multiple storage back ends such as in-memory for reference and testing purposes, file system for storing layers in a local directory tree, and object storage for storing layers as objects in popular object stores such as Amazon S3.

Challenges of Scale

The increasing popularity of containers and the shift in application development towards cloud-native applications pose several challenges for Docker storage on the client and registry sides.

High Redundancy

As of March 2019, Docker Hub contains more than 2 million public images. Grossly underestimating, we found that those images would utilize more than 1 PB of storage in raw format. The utilization is likely several times higher as we have not considered all images, e.g., we omitted the ones stored in private repositories. Additionally, every day more than 1,500 new images are added. This puts pressure on the storage infrastructure, and it is important to understand the challenges in storing Docker images in order to keep registries and client-side storage scalable.

As described above, Docker employs two mechanisms to reduce image storage utilization: layering of images and compression. However, even with these space optimizations, the storage utilization is still significant. Looking at the individual contributions of each mechanism on the 10,000 most popular images in Docker Hub, we found that layering provides a reduction of 1.48×, and compression decreases the data set by an additional 2.38×. Combined, this results in a total reduction of 3.54×. While this would reduce the estimated 1 PB to approximately 290 TB, storing all images still requires a significant infrastructure budget. Using AWS S3 standard storage, the resulting annual cost for storage alone would be between $75,000 and $130,000 (depending on the specific AWS region) plus any additional networking costs. For companies that provide registries as a service, e.g., Docker Hub, Jfrog, Artifactory, or Quay, this is particularly problematic. However, even companies maintaining their own registries are sensitive to the high cost of the required storage infrastructure.

To reduce storage utilization of Docker images, the primary goal is to remove any existing redundancy in the stored data, as is intended by the layering of images. However, we found that this is ineffective in its current form [9]. In our sample data set of the 10,000 most popular Docker Hub images, 67,047 unique layers still contain almost 80% duplicate files.

We believe that this is due to two main reasons. First, Docker images must be self-contained, contrary to earlier approaches for software packaging (e.g., RPM or DEB). As a result, completely unrelated images may rely on common components like binaries or shared libraries. In our 10,000-image data set, we found that libraries such as libslang, libstdc++, or libc are present in over 1,000 images. Second, developers create their images independently without exhaustively considering existing layers. This leads to many “almost equal” layers, i.e., layers that share a large number of, but not all, files with existing layers and as a result are not identical and so must store separate copies. That is not to blame developers; examining existing layers is not a task to be performed manually, and further, one needs to have the required incentives to even consider doing so.

On top of the registry storage redundancy, network traffic and client-side storage are also affected. Suboptimal layering means that duplicate data is unnecessarily transferred over the network, potentially increasing expensive outbound network traffic in a typical public cloud offering. Additional network traffic can increase startup times, whereas “almost equal“ layers can increase storage space utilization on a single client unnecessarily.

We proposed one approach for solving the redundancy problem through layer restructuring that considers both storage and network utilization [6]. The approach takes the existing layers in a registry and constructs new layers out of the set of all files, such that storage space and network redundancy are minimized. Preliminary results on a small, 100-image data set show that we can achieve storage space savings of up to 2.3×. In the same paper, we discussed the redundancy problem in more detail and explain why file-level deduplication on the registry-side is insufficient.

Low Performance

While containers are, in most cases, much more performant in terms of startup times compared to virtual machines, new use cases such as serverless computing are demanding even lower latencies. Those requirements put pressure on the storage infrastructure, both at the registry and the client side.

As previous work has found, pulling can contribute as much as 76% to the overall container startup time [4]. Hence, the registry is a critical component in the container infrastructure and needs to be designed to minimize pull latencies and serve images as fast as possible. One direction for improving registry performance is to exploit workload characteristics and integrate workload-aware optimizations in a registry’s design or configuration. We performed an in-depth analysis of production traces from the IBM Cloud Container Registry to study common registry workloads and drive potential optimizations [3, 5]. The analysis revealed several important characteristics. First, there are often hotspot layers, which are accessed more frequently than others, leading to a skewed workload. For example, at one of the registry sites, 59% of requests only went to 1% of the layers. Second, most layers are small, with 65% being smaller than 1 MB while 80% are smaller than 10 MB. Third, requests are correlated, i.e., if a client requests an image manifest from a repository and the repository has recently seen new layers being pushed, then these new layers are likely to be pulled.

These observations encourage the use of layer caching and prefetching optimizations to reduce registry load and pull latencies. Using these lessons, we proposed a new registry design [3]. The design employs a two-tier registry cache and exploits the correlation of push and manifest pull requests to preload layers that are likely to be pulled into the cache. Each time a client requests a manifest for an image in a repository that has seen an update in the recent past (defined by a threshold parameter), the layers from the manifest are prefetched into the cache. Our evaluation revealed that having such an optimized backend storage system for the registry can reduce the latency from 100 ms to 10 ms for layers smaller than 1 MB.

Besides the registry, client-side storage can also affect container startup and runtime performance. This is particularly problematic in large-scale setups, where either many containers run on a single host or the same image needs to be pulled by a large number of nodes to run a parallel workload.

In the first case of many containers being started simultaneously on one host, the choice of storage driver can significantly impact how fast containers start and complete [7]. The most important property is the granularity at which the driver performs copy-on-write, i.e., at file- or block-granularity. For example, we found that for the OverlayFS driver, startup latencies can reach hundreds of seconds for write-heavy workloads, which trigger large copies of data due to copy-on-write. As a result, the completion of those containers is also delayed significantly. In contrast, drivers, which perform copy-on-write at block granularity (e.g., Btrfs or ZFS), did not significantly affect startup latencies.

However, other workloads draw a different picture. For example, when running an Ubuntu `dist-upgrade` in 10 containers in parallel, file-based drivers (both OverlayFS and AUFS) outperformed block-based drivers significantly. This could be due to the fact that block-based drivers are often based on native file systems and, hence, benefit less from the Linux page cache, which could slow down containers with mixed read/write workloads. However, we do not know the exact reason at this point.

In the case of large-scale parallel workloads, which require users to pull the same image on many different nodes, additional problems arise. Most importantly, pulling the same image several times (potentially hundreds or thousands of times depending on the scale of the workload) wastes network bandwidth during the pull and storage capacity on the individual Docker clients. Therefore, it is desirable to enable individual clients to collaborate when pulling an image, i.e., let different clients pull different layers of the image and only store a single copy of the image on shared storage such as an NFS file system. In environments where no local storage is available, such as an HPC cluster, sharing images is a necessity to enable containerized workloads.

To enable collaborative pulling and sharing of images, Docker clients need to be synchronized. With Wharf, we have built such a system [8]. As we assume the existence of a shared storage system for the container images, we can use this shared storage to store the global state for all clients, e.g., which images have been pulled already, which images are currently pulled, and who is pulling which layer. Wharf uses additional optimizations such as minimizing lock contention by exploiting the layered structure of Docker images and writing image changes to local storage, if available, to reduce overhead during pulling and running an image. For large images pulled in parallel to an NFS share, Wharf can improve pull latencies by up to 12× compared to a naïve solution, in which each client pulls its images to a separate location on the NFS share.

Conclusion

Containers are expected to form the backbone of prospective computing platforms. However, even though individual containers are lightweight, providing and operating infrastructure for millions of containers is a hard challenge. In this article, we described how Docker stores container images and presented the challenges that we discovered when operating large-scale container deployments: high data redundancy across images, inefficiencies in graph drivers, low-performing registries, the inability to effectively use images on shared storage, and others. We referenced some of the possible solutions and hope that this article will nourish the discussion on this important topic.

Acknowledgments

We would like to thank our collaborators and co-authors from academia, IBM Research, and other organizations: Ali Butt, Yue Chang, Hannan Fayyaz, Zeshan Fayyaz, Dean Hildebrand, Wenji Li, Michael Littley, Heiko Ludwig, Nagapramod Mandagere, Nimrod Megiddo, Mohamed Mohamed, Raju Rangaswami, Douglas Thain, Amit Warke, Ming Zhao, Nannan Zhao, and Chao Zheng.

References

[1] Docker Hub: https://hub.docker.com/.

[2] OverlayFS: https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt.

[3] A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rupprecht, Y. Cheng, N. Zhao, D. Skourtis, A. S. Warke, H. Ludwig, D. Hilde-brand, and A. R. Butt, "Improving Docker Registry Design Based on Production Workload Analysis," in *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST '18)*, pp. 265–278.

[4] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast Distribution with Lazy Docker Containers," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, pp. 181–195.

[5] M. Littley, A. Anwar, H. Fayyaz, Z. Fayyaz, V. Tarasov, L. Rupprecht, D. Skourtis, M. Mohamed, H. Ludwig, Y. Cheng, and A. R. Butt, "Bolt: Towards a Scalable Docker Registry via Hyperconvergence," in *IEEE International Conference on Cloud Computing (IEEE CLOUD 2019)*.

[6] D. Skourtis, L. Rupprecht, V. Tarasov, and N. Megiddo, "Carving Perfect Layers Out of Docker Images," in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '19)*, USENIX Association, 2019.

[7] V. Tarasov, L. Rupprecht, D. Skourtis, W. Li, R. Rangaswami, and M. Zhao, "Evaluating Docker Storage Performance: From Workloads to Graph Drivers," *Cluster Computing*, Online First, 2019.

[8] C. Zheng, L. Rupprecht, V. Tarasov, D. Thain, M. Mohamed, D. Skourtis, A. S Warke, and D. Hildebrand, "Wharf: Sharing Docker Images in a Distributed File System," in *Proceedings of the 9th ACM Symposium on Cloud Computing (SoCC '18)*, pp. 174–185.

[9] N. Zhao, V. Tarasov, H. Albahar, A. Anwar, L. Rupprecht, D. Skourtis, A. S. Warke, M. Mohamed, and A. R. Butt, "Large-Scale Analysis of the Docker Hub Dataset," in *IEEE International Conference on Cluster Computing (IEEE Cluster 2019)*.

SAVE THE DATES!

SRE CON® EUROPE MIDDLE EAST AFRICA

OCTOBER 2–4, 2019 • DUBLIN, IRELAND
www.usenix.org/srecon19emea

SRE CON® AMERICAS WEST

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

SRE CON® ASIA PACIFIC

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

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

COLUMNS

Reliable by Design

The Importance of Design Review in SRE

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

Every organization has regrets about software that doesn't scale, that's difficult to run, or hard to use, or where we just wish we'd done something differently early on, when it would have been easier and cheaper. Sometimes we need to execute fast and accrue technical debt, but often the right thing would have been as easy and fast as the wrong thing—and those are our failures as a profession.

In many organizations (especially larger ones), when a new system is being built or a major change is planned for an existing system, a design (also often known as an RFC, or Request for Comment) is written and reviewed by peer engineers. This is a document that describes the planned change, including the reasons for making it, and alternatives to the proposed design that were considered and rejected. The ideal level of detail is just enough that any competent software engineer could implement the system from the design—in other words, it should be significantly higher-level than code, while clearly describing requirements, system architecture, dependencies, and tradeoffs.

Of course, not every change needs a design document, and people often aren't sure where to draw the line. My heuristic is that any project that is going to lead to the creation of new monitoring or runbooks, or large revisions to existing ones, merits a written design. This does not mean that failure to create required monitoring or runbooks excuses the need to produce a design document.

I'm going to nail my colors to the mast here and say that if you're not producing designs and participating in design reviews with partner teams, then you're not doing SRE but some other flavor of operations. SRE is predicated on having agency and on teams having a voice in decisions that affect the systems they are responsible for. Without designs and a review process, teams don't have the insight they need into the changes that others are planning in the production environment, so having that voice in significant decisions becomes impossible.

Written designs have many advantages over informal discussion or presentations. As an author, the written form pushes you to think through details that you might not otherwise spend enough time on. As a reviewer, it gives you time to reflect on the proposed change. It also works better for distributed teams, because feedback can be given and responded to in an asynchronous manner. A long-term advantage of written designs is that they can provide a history of major changes in your organization's systems as well as the reasons behind them and the decisions made. Over time, the reality of your systems will diverge from original designs, but an archive of design documents will still be a valuable resource.

The design review process can be problematic in a few ways on a human level. One problem is time: feedback on designs may drag on for several weeks if there are many interested reviewers. I recommend setting a clear deadline for feedback (in the header of the document itself). Around two or three weeks is ample. If there are unresolved discussions at this point, then schedule meetings to discuss (either one meeting or multiple one-to-one meetings). This will save time overall, and it is easier to resolve technical disagreements face to face.

Another big problem is the use of the design review process to show off, debate matters of taste, or nitpick. This kind of behavior makes people reluctant to write and share designs, and the resulting failure to communicate leads to repeated work, a lack of shared understanding, and failures to catch major problems that colleagues would have noticed. Design review comments should be well intentioned and solely about the important points in the design rather than the color of the proverbial bike shed. Think carefully before commenting. Many large organizations develop norms and guidelines for technical discussions, including pointing out and discouraging this kind of "bike shedding."

I've never seen much guidance on how to perform design reviews as a peer engineer or as a technical lead. People tend to read the document and apply their expertise in an ad hoc way. As someone who has reviewed a fair number of such designs, I've found that it's time-consuming, and I often worry that there's something important I haven't thought about. There's no structured way to approach the problem.

Atul Gawande's book *The Checklist Manifesto* [1] may point towards a solution. Gawande is a surgeon. He noticed that it was very common to make errors in complex surgical procedures. He distinguished between two kinds of errors: errors of ignorance, where not knowing something causes a mistake, and errors of ineptitude, where we don't make proper use of what we know. In the modern world, surgery is such a complex task that forgetting steps, or failing to plan ahead for some eventuality, is almost inevitable. Gawande looked at what other professionals do—in professions like civil engineering and aviation—and it turns out they use checklists to avoid errors of ineptitude.

Checklists may sound like a tedious process—and nobody really likes more process—but bear with me. Surgical checklists are not a substitute for professional expertise. In fact, they absolutely require that expertise to execute them. They are not long manuals that prescribe every detail of every step in a process but instead are prompts, intended to make sure you don't accidentally leave out a key step in a complex task. Surgical checklists are quite short, leaving minutiae to the judgment of those using them; the WHO safe surgery checklist [2] fits on one page, although it does refer to other checklists that may need to be consulted under certain circumstances.

It turns out that well-crafted checklists make a big difference in surgical outcomes—a 2009 study showed that the WHO checklist reduced the incidence of post-surgical complications by a third. In addition to making sure basic (but important) things aren't forgotten, they also encourage and empower all members of a team to point out omissions or problems. They can make teams work better.

I believe checklists can help us improve our system designs too. There is a lot of wisdom in the SRE profession about how to design operable, scalable, reliable, distributed systems. We can add a lot of value at this stage of the process. But there's no checklist to help us do it. What might such a checklist look like? Here's my version [3]:

- **What and why**: do I understand the need for the change, the design itself, and how the proposal relates to other systems?
- **Who**: are there affected teams that haven't been asked to look at this design? If there are privacy or security implications of this system, are there appropriate reviewers?
- **Alternatives considered**: is building a new system the right approach?
- **Stickiness**: what's hard to change about the proposed system?
- **Data**: consider consistency, correctness, encryption, backup, and restore strategies.
- **Complexity**: where is this design overly complex, and can that complexity be reduced?
- **Scale and performance**: how does the design support the scale and performance needed?
- **Operability**: how will the system support (or not) the humans running it?
- **Robustness**: how does the design handle failures, and other issues such as overload?

This high-level checklist is fairly terse, as a usable checklist needs to be—remember, this is here to prompt your expertise, not to replace it. For some designs, some sections of the checklist may not apply—maybe the design in question is a piece of automation that doesn't need to scale, or a stateless service that doesn't need to deal with some of the data considerations. The sections below give more detail for each item on the checklist and, in some cases, further sub-checklists.

The **what and why** questions are first because they are the most important. If you read a design and don't understand it and why it's needed, then the design is missing information or lacking in clarity. If you don't understand it when you're reviewing the design document, you definitely won't understand it when you're trying to respond to a production fire. The best way forward here is to tell the author which parts you're having trouble with and ask them to update the document before proceeding.

Next, who:

- Is there a good reason that you've been asked to review this system? It's good to understand whether the author is looking for some particular expertise or perspective from you, and make sure you've addressed that.
- It's also useful to check who else has been asked to review and that all the affected teams have been asked. Support or operations teams are often left out to the detriment of all involved. Owners of systems that the new system will depend upon should usually be asked to review new designs.
- Many changes should be reviewed specifically for privacy and security.

Alternatives considered is a subject often neglected but important:

- Is there an open-source tool, or a similar proprietary system at this organization, that might work? If so, did the author of the design talk to owners of those similar systems about this use-case? Proliferation of systems is hugely costly. It takes time to build and maintain them, and it complicates an organization's production environment.

Stickiness: give special consideration to thinking about which aspects of a proposed system will be hard to change in the future.

- Imagine you're trying to migrate all the users of the system away from it to its replacement or that you're planning a major change of some sort. What aspects of the design will make that easier or harder? For example, allowing users to extend your code limits what you can do in the future and makes migrating them to replacement systems much more difficult, and so does tight coupling with other systems.
- What assumptions are baked into the architecture or the data model that might change in the future?

Data:

- What is the flow of data through the system?
- What are the data consistency requirements, and how does the design support them?
- Which data can be recomputed from other sources and which cannot?
- Is there a data loss Service Level Objective (SLO)?
- How long does data need to be retained, and why?
- Does it need to be encrypted at rest? in transit?
- Are there multiple replicas of the data?
- How do we detect and deal with loss or corruption of data?
- How is data sharded, and how do we deal with growth and resharding?
- How should data be backed up and restored?
- What are the access control and authentication strategies?
- Have relevant regulations such as GDPR and any data residency requirements been addressed?

Complexity:

- Does each component of the system have a clearly defined role and a crisp interface?
- Can the number of moving parts be reduced?
- Is the design similar to existing systems at this organization? Is it built using standard building blocks (K/V stores, queues, caches, etc.) that engineers at this organization already understand? Does it use the same kinds of plumbing such as RPC mechanisms, logging, monitoring, and so on?
- Does the proposal introduce new dependencies (e.g., uses a different type of message queue than other systems in the same organization) and if so, is that really necessary?

Scale and performance:

- What are the bottlenecks in this system that will limit its scale and throughput (not forgetting the impact of writes and locking)?
- What's the critical path of each type of request, and how do requests fan out into multiple sub-requests?
- What is the expected peak load, and how does the system support it?
- What is the required latency SLO, and how does the system support it?
- How will we capacity plan and load test?
- What systems are we depending on, and what are their performance limits and their documented SLOs?
- What will it cost to run financially?

Operability:

- How does the design support monitoring and observability? For instance, systems involving queues may require extra care in monitoring.
- Do all third-party system components provide appropriate observability features?
- What tools will be available to operators to understand and control the system's behavior during production incidents? How will these tools make clear to the operator what specific actions they should take to avoid surprises?
- What routine work is going to be needed for this system? Which team is expected to be responsible for it? How much of it can and should be automated, and will that automation reduce the operating team's understanding of the system?
- How do we detect abusive users or requests, and what action can we take in response?
- If the design involves relying on third parties (such as a cloud provider, hardware or software vendor, or even an open-source community), how responsive will vendors be to your feature requests or problems?
  - Are all configurations stored in source control?

Robustness:

- How is the system designed to deal with failure in the various physical failure domains (device, rack, cluster/AZ, datacenter)?
- How will it deal with a network partition or increased latency anywhere in the system?
- Are there manual operations that will be required to recover from common kinds of failure?
- How could an operator accidentally (or deliberately) break the system?
- Is there isolation between users of the system?
- What are the smallest divisible units of work and data, and will we likely see hotspotting or large shards?
- What are the hard dependencies of this system, and can we degrade gracefully? How to ensure soft dependencies don't become hard dependencies?

- How can we restart this system from scratch, and how long will that take? Do we depend on anything that might depend on this system? Don't forget DNS and monitoring.
- How will this system deal with a large spike of load?
- Does the system use caching, and if so, will it be able to serve at increased latency without the cache?
- Is the control plane fully separate from the data plane?
- Can I canary this design effectively (e.g., leader-elected designs are hard to canary)?
- Can this system break its back ends by making excessive requests?
- Can this system autonomously drain capacity, and how have risks been managed, in particular with respect to human operators' ability to understand and control the system?
- Can this system autonomously initiate resource-intensive processes like large data-flows (perhaps for recovery purposes), and how are those risks managed?
- Can this system create self-reinforcing phenomena (i.e., vicious cycles)?

These are the things I think about when reviewing a design. No two systems are the same, so not all of these questions make sense for every type of system. As with the WHO surgical safety checklist, local variations are very much encouraged. This is a starting point [3].

All systems involve risk, and all systems make tradeoffs. Better system design won't eliminate all problems. We just can't anticipate everything—errors of ignorance are inevitable. But errors of ineptitude are avoidable, and part of maturing as a profession is getting more systematic about reducing errors of ineptitude.

A good design helps us to understand tradeoffs and risks more thoroughly and make reasoned, deliberate choices that make the most sense for our organizations. Taking the time now to write a design for your team's next big project and get it reviewed by your peers might be the most impactful work you can do.

References

[1] A. Gawande, *The Checklist Manifesto: How to Get Things Right* (Metropolitan Books, 2009).

[2] WHO Safe Surgery Checklist: https://www.who.int/patientsafety/safesurgery/checklist/en/.

[3] L. Nolan, SRE Reliable by Design checklist: https://www.usenix.org/sites/default/files/fall19_sre_checklist.pdf.

Python News

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 this column, I'm covering a bit of Python news, with some info about type checking in more depth.

Time for Python 3

For years now it's been made very clear that Python 2 is coming to the end of support—this was put in writing in PEP 373 [1] . The original date was pushed back to 2020 to give everyone some more time to move to Python 3. Most projects that are still actively maintained have put in the effort to support Python 3, and Python 3 hasn't been standing still. It's adding features like async support and syntax for supporting static type checking (more on this in a moment) that makes it a more modern language than Python 2.

To put the cherry on top, now that it's almost 2020, developers of some prominent Python projects have announced that they're going to discontinue support for Python 2 in future release of their project. In case your projects could be affected by this, go take a look at the projects listed at the Python 3 Statement website (https://python3statement.org/). Many fundamental projects have decided that after performing the work to be compatible with both Python 2 and Python 3 for some time (years and years in some cases), they want to reduce their workload by just supporting Python 3. This seems only fair. Python 2 has had an extraordinary lifetime, and now it's time to retire it with grace. I encourage you to take a look at python3statement.org and to understand if the projects you rely on directly or indirectly will impact your work, and to plan accordingly.

Type Hints in Python 3

I'm in a situation shared by many of my peers where we're still planning our transition to Python 3 for most of our infrastructure code. As part of getting our stories together for upgrading, I'm thinking about the fun stuff that has been created as Python 2 has gone stale.

So I'd like to take a look at one of these cool features I'm anxious to put to good use: static type checking. Static type checking in Python makes it possible for a process that reads code to check that all types passed into a function, and all return values from the function, are appropriate, and alert the developer to deviations that would cause bugs. By using static type checking, you can eliminate a lot of bugs without ever having to run the program. In the way Python implements this, the static checker is an external process—it's not Python itself that checks it before running. So whether or not you use this feature, your code will still run.

Some languages have incorporated static type checking from the outset, but this is not how Python was developed. Historically, Python is among the languages that is dynamically runtime type-checked, which means that it's common to have crashes when there is a severe enough type mismatch. Because of the way that static type checking is being added to Python late in its development, its power to catch problems has been limited by speed of development of the type checking tools, and the rate of adoption and use of those tools. The tools are actually being developed at a really fantastic pace, but many libraries and other code bases can only adopt type checking as they move to recent Python 3 versions.

We'll look at the basic idea of static type checking in Python and at a cool feature that could be added. Unfortunately, this column is not going to be able to cover Python type-checking features in depth. For that there is a lot of excellent documentation written on how static type checking can be used in recent versions of Python when you're ready to use it—the official documentation is thorough and very deep. So that's not what I'm going to write about here.

The basic observation that makes static type checking attractive is that as a Python programmer you know that the following code will run:

```
def badlen(container):
    return len(container)
```

but you also know that the built-in `len()` is only useful on certain types. You probably also know that objects of those types have the *dunder* (double underscore) method `__len__()` to provide their length. And you also know that invoking the `len()` built-in function on an inappropriate type causes a runtime TypeError:

```
>>> badlen(7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in badlen
TypeError: object of type 'int' has no len()
```

People who come from static (type-checking-wise) languages often look at Python and its peers and ask why this is acceptable, when clearly other languages can catch this sort of error before they are ever run. The obvious answer is that part of what Python provides is a very dynamic programming environment where a lot of the knowledge required for static checking is not possible. The success of Python makes it clear that static type checking isn't the most important feature to make the language usable and productive.

Since compile-time type checking promises to reduce or eliminate this class of error, it would make Python better to have it, so a lot of work went into discussing what would be necessary to add it without causing any extra work for people who won't use it while bringing benefits to people who do want it.

The guiding principle behind Python development has long been that, to the extent possible, Python should try to advance with backwards compatibility in mind. To this end, a bit of syntax was created and specified which allowed function annotations in PEP 3107 [2]. With that in place, PEP 484 was hashed out; it introduced a standard for type "hints" using the PEP 3107 annotations. Although allowing us as developers to communicate what the type is, the interpreter in effect completely ignores all of this information at runtime. Instead, its purpose is to allow tooling to be built to verify that annotated functions comply with the types that are described. With these checkers in place, even better tests can be created.

So with the introduction of PEP 484 and a common standard for type hinting, tools can be built ensuring that functions are using type hints to get the right input types and therefore are returning the right output types.

This might seem like a small refinement of a very popular language; after all, Python isn't the only language that has succeeded by growing its user base without static type checking. However, statically checkable, and therefore avoidable, type errors are a very common source of bugs, so in the long term, opting into this is likely to be a huge benefit to those who use it.

So what do type hints look like? They can change the declaration of a variable in a function call, for example, from `variable_name` to `variable_name: type`, like this:

```
def betterlen(container: list):
    return len(container)
```

That tells the type checker that the function takes a list. Usually lists are of a particular type, though, and we can ask the type checker to check for an appropriate type of list, or we can make it clear that we're not concerned about the type of list. This is normal Python behavior. To make this possible, there is the `typing` module, which provides definitions of objects that the type checker can use to allow you to declare how thoroughly you want to check your lists, dictionaries, or other container types.

To enable this, you include the `typing` module in your code and import type specifications, which provide the specificity for the structure and types of the things they contain. In this case, we're going to start with a specification of `Any`, which explicitly says "accept that this list can contain elements of anything, it's fine." But this could also be used to be more specific about only particular built-in types or user-defined types. It looks like this:

```
from typing import Any, List

def goodlen(container: List[Any]):
    return len(container)
```

By invoking a static analyzer (`mypy` in this case) on a chunk of code with a type mismatch, like this:

```
goodlen(7)
```

it can describe the problem it sees without actually running the code:

```
$ mypy simply_doesnt_work.py
simply_doesnt_work.py:8: error: Argument 1 to "goodlen" has
incompatible type "int"; expected "List[Any]"
```

Whee! That was pretty easy. For a basic introduction, the next step is to go one more step and specify the return value, which we haven't done yet. We want to specify the return type, too, because the current state of the `goodlen()` function creates a dead-end for the type checker. Because the return type isn't declared, the type checker graph bottoms out and can't do further checking at this point.

So to help the checker, you can add a return type simply by adding a `->` *type*. For a length, we'll always be returning an integer; a simple case looks like this:

```
def betterlen(container: List[Any]) -> int:
    return len(container)
```

The more annotations that are added to a code base, the more automatically simple but critical mistakes can be avoided before your code is ever run.

By itself, this has benefits for unit and integration tests. You can just start adding harmless annotations, and start to check whether your libraries, dependencies, etc. are doing the right thing.

But there's another very interesting thing that is possible, which, hopefully, Python will adopt in the future. It's presently available in Rust, so let's use that as the example.

You may have heard of Rust, the language, since it's received a lot of attention since it hit 1.0 in 2015. In case you haven't had a chance to look into it, I think it's fair to say that its goal is to be a language that can achieve the performance and control of C or C++, while providing the memory safety of a garbage-collected language like Java, Python, or Go. In addition, Rust also eliminates other risks present in most other mainstream programming languages.

As part of providing this attractive sounding set of goals, Rust includes strong compile-time type checking as a fundamental feature. Rust also incorporates a very interesting idea: exhaustive checking of all possibilities in a match (as I understand it, this originated in the ML languages). This is needed because a lot of bugs are created when a series of conditional statements—e.g., in Python an `if… elif… else`—is produced that due to oversight, or changes in the set of possible choices, ends up not covering all of the possibilities.

To make this work, Rust uses a clever trick. The implementation of this clever trick is the `match` expression, which is like a `case` or a `switch` in other languages. But instead of being just another way of writing if…else if…else if…else, it makes sure that when a match is invoked, it can identify that all possible matches have been covered. So if the type being matched is an unsigned 32-bit integer, then the compiler knows that if you haven't specified either all numbers from 0 to 2^{32}-1 or used a default match (Rust does this with the underscore target in a match—this is the equivalent of an `else` in Python), then you have left possible values which haven't been accounted for, and it will refuse to let that code compile or run.

Another clever extension is combining this with `enums`, or an enumerated set of possible values that are declared up-front. With an `enum`, the compiler knows whether or not all possible arms of the possible matches with enum values have been checked, because the enum can only have a fixed number of possibilities. A quick example of what this could look like in Rust is:

```
enum BreadSpreads {
    Butter,
    Margarine,
    CreamCheese,
    Nutella
}

fn breakfast_bread(spread: BreadSpreads) {
    println!("Breakfast bread with {}",
        match spread {
            BreadSpreads::Butter => "butter",
            BreadSpreads::Margarine => "margarine",
            BreadSpreads::CreamCheese => "cream cheese",
            BreadSpreads::Nutella => "nutella"
        }
    )
}

fn main() {
    let butter_spread = BreadSpreads::Butter;
    let margarine = BreadSpreads::Margarine;
    breakfast_bread(butter_spread);
    breakfast_bread(margarine);
}
```

This is very straightforward and not particularly noteworthy when it is working. What is more interesting is that if you change the `breakfast_bread` function by removing any of the arms of the match (let's use `Margerine` for this example), the compiler will refuse to compile it. It will tell you that the code is broken and save you from having to discover the problem in production:

```
$ cargo build
   Compiling breadspread v0.1.0 (/home/spacey/dvcs/pcn/login/
2019-6/breadspread)
error[E0004]: non-exhaustive patterns: 'Margerine' not covered
  --> src/main.rs:10:15
   |
1  | / enum BreadSpreads {
2  | |     Butter,
3  | |     Margerine,
   | |     --------- not covered
```

```
4  | |    CreamCheese,
5  | |    Nutella
6  | | }
   | |_- 'BreadSpreads' defined here
...
10 |         match spread {
   |              ^^^^^^ pattern 'Margerine' not covered
   |
   = help: ensure that all possible cases are being handled,
possibly by adding wildcards or more match arms

error: aborting due to previous error

For more information about this error, try 'rustc --explain
E0004'.
error: Could not compile 'breadspread'.
```

This feature of the Rust compiler works because the set of possible enums can't change once they've been declared. Of course, being able to change that after runtime would break guarantees that Rust provides with this little trick. So generally, the compiler looks at the match to make sure that you have accommodated each possible variation that the enum could take, because as an enum those possibilities are, well, enumerated in the code. In addition, as with most case/switch/if...then...else constructs, you have the equivalent of an else clause, so this need for an exhaustive match doesn't require you to write out a match for every possible case individually. It just requires that you don't leave off the equivalent of the else clause and leave cases uncovered. It doesn't protect the programmer from every mistake, but it prevents cases from being missed.

So it's interesting to ask, would this be possible in Python and how much would it help? And what would it look like if it was being used? Until recently the nearest available data types to structures and enums are dictionaries or sets (or possibly classes built on these), however these are not static enough, so they can't be used for this kind of type checking. There is no mechanism for the type checker to exhaustively test all of the possible variations with a dictionary, for instance, since the possibilities are unknowable at check time.

So since there are are other motivations to want an enumeration type, PEP 435 [4] was written and proposed, and an enumeration type was added in Python 3.4. Since this piece is in place, it seems likely that there will be a way in the near future to ask Python type checkers to exhaustively check enums and to alert to this common type of bug.

I expect that the static type checking features of Python 3 will improve and provide better safety in the future. I think it's interesting to think about how the type checkers could influence future programming practices in Python. It could become more common for Python to develop recommended idioms that will help to restrict the breadth of possible mistakes we make, similar to being able to check all branches of if/elif/else statements to provide better information for a static type checker to feed on. It will be interesting to see whether or not some of the ideas of what's Pythonic will change based on what's best for modern type checking.

References

[1] PEP 373: https://www.Python.org/dev/peps/pep-0373/.

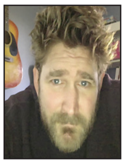
[2] PEP 3107: https://www.Python.org/dev/peps/pep-3107/.

[3] PEP 484: https://www.Python.org/dev/peps/pep-0484/.

[4] PEP 435: https://www.Python.org/dev/peps/pep-0435/.

iVoyeur
Prometheus (Part Two)

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

I'm writing to you from the warm afterglow of Monitorama PDX 2019, a conference where we are invariably treated to at least one talk concerning itself with the ever-noble cause of statistics for anomaly detection.

If that sounds a bit sarcastic, it probably is, but only a little bit. If giving talks with titles like "statistics for sysadmin" is a crime, it's one I myself am guilty of many times over, and I fully admit that, like elk grazing their way across a hillside, no matter how many times I see it, it never fails to fascinate.

The talk invariably begins the same way, with a baseline introduction of the normal distribution and an accompanying graphic depicting our old and steadfast friend the bell curve, along with a rundown of some of our very favorite actors, like standard deviation from the mean, z-score, and the like.

But the speaker has a secret that you can probably guess if you are a regular reader of this column, which is this: system metrics are rarely normally distributed. So, really, there are two paths this talk can walk.

In the first, the speaker has gotten lucky and found a use-case for which the input signal happens to be normally distributed, and has therefore been able to apply straightforward statistical analysis to achieve a successful predictive model. The speaker will subsequently encounter a litany of follow-on problems that are sure to entertain us, including unexpected seasonality like cyber-Monday and unpredictable aperiodic events such as labor-union strikes and the like.

If the speaker's problem is not easily represented by a normally distributed metrics signal, things get interesting pretty quickly. This path descends into the land of custom models, advanced math, and data science, which is always fun. But even if the speaker is ultimately successful, the results are rarely directly applicable to our own peculiar set of problems, or are nontrivial to implement if they are.

Well, that's not exactly fair. It's true that complex anomaly detection models are difficult to implement, but that's also true of simple techniques that work on normally distributed signals. Aside from some commercial offerings like Circonus and SignalFx, and a handful of rapidly aging, very basic tools like the Holt-Winters predictive analysis features built into RRDTool, there haven't really been any tools in the monitoring world you can pick up and use to experiment with anomaly detection on time series.

That's why I was delighted to see a pair of talks this year whose content could succinctly be described as: "My Prometheus Queries: Let Me Show You Them!" One is a lightning talk by Jack Neely called "Five Neat Prometheus Tricks" [1], and the other, a full-length talk by Andrew Newdigate entitled "Practical Anomaly Detection Using Prometheus" [2].

As promised, Jack shows us some neat tricks, including overriding the `avg()` function to express things that aren't averages (like ratios), and he helpfully explains how to use operators creatively to craft up/down alerts that don't fire if the host has only been up for five minutes, and measure metrics like memory usage as a function of things like OS or Go version.

Andrew meanwhile dives into the nitty-gritty of anomaly detection, showing us how to compute z-scores on moving averages of Prometheus vectors. Both talks are well worth seeing, but what has me excited is the more general pattern of using PromQL to express an answer, or set of answers, to common monitoring problems. This is especially true in the context of anomaly detection, where we have seen so many talks on general principles without being able to lay our hands on anything like a functional language driving a visualization engine capable of expressing anomaly detection primitives like the z-score.

In my last article, I detailed Prometheus's data model and commented that I was enamored of the tool's ability to pull together different types of engineers by providing a system-agnostic monitoring signal that everyone could "get behind." The simplicity and ubiquity of Prometheus's data model is a huge success, which I believe likely to outgrow the tool itself.

Prometheus Query Language

Prometheus's query language, PromQL, is another great success, as evidenced by the fact that engineers are using it in conference talks as if it were a specification language to communicate techniques and general solutions to common monitoring problems. While the language is certainly more tightly coupled to Prometheus itself than the data model, and has its limitations, I think it was designed sufficiently well that it's already doing a pretty great job of scaling beyond the imagination of its creators.

The simplest Prometheus query is the literal name of a metric. One metric that will probably be available in every Prometheus install is "up." The query syntax is very simple:

```
up
```

The `up` metric is built into every off-the-shelf Prometheus exporter and displays a "1" if the exporter could be contacted by the poller or "0" if it could not. The `job` label shows the name of the exporter that generated each particular `up` metric.

We can filter the output of this query by label, by adding the label name in braces. Node_exporter [3] is the de facto Prometheus system agent, so its `up` metric is a pretty solid metric for host availability in general. We could filter for only the `up` metrics exported by node_exporter like so:

```
up{job="node_exporter"}
```

Internally, every query is actually implemented in this way, with a comma-separated list of label-name equality-operator and value surrounded by braces. Our first query was actually a shortcut for:

```
{__name__="up"}
```

and our second query:

```
{__name__="up",job="node_exporter"}
```

Our equality operator doesn't have to be =. In fact, PromQL supports the following range of equality operators:

- =: Select labels that are exactly equal to the provided string
- !=: Select labels that are not equal to the provided string
- =~: Select labels that regex-match the provided string
- !~: Select labels that do not regex-match the provided string

Regex in PromQL is RE2 [4] syntax, and generally every query that uses a regex must either specify a name or at least one label matcher that does not match the empty string. You can also match the same label multiple times, so an admittedly convoluted way to match every `up` metric from `node_`, except those from `node_blarg` could be:

```
up{job=~"node_.*", job!="node_blarg"}
```

What if, instead of the output of the latest poll, we wanted to see the last five minutes of samples from the poller?

```
up{job="node_exporter"}[5m]
```

By adding a range duration in square brackets to our query, we express to Prometheus that we want to see every sample within the duration for every returned result. Prometheus refers to this output (confusingly) as a "range vector," as opposed to a single-sample response or "instant vector." In the example above, we've expressed our desired duration in minutes, but you can use seconds, minutes, hours, days, weeks, or years instead.

Durations and range-vector results give us the opportunity to begin measuring aggregations of samples over time. For example, to find hosts that have been unavailable any time in the last hour, we can use a function to retrieve the minimum value of the `up` metric over a duration of samples from the last hour (this will return `0` for hosts who have been down any time in the duration):

```
min_over_time(up{job="node_exporter"}[1h])
```

PromQL supports the aggregations you'd expect as well as a few you might not have predicted:

- avg_over_time(range-vector): the average value of all points in the specified interval
- min_over_time(range-vector): the minimum value of all points in the specified interval
- max_over_time(range-vector): the maximum value of all points in the specified interval
- sum_over_time(range-vector): the sum of all values in the specified interval
- count_over_time(range-vector): the count of all values in the specified interval
- quantile_over_time(scalar, range-vector): the φ-quantile ($0 \le \varphi \le 1$) of the values in the specified interval

- stddev_over_time(range-vector): the population standard deviation of the values in the specified interval
- stdvar_over_time(range-vector): the population standard variance of the values in the specified interval

PromQL also has first-class support for "offsets," meaning it's easy to express, for a given query, that you want to see the samples from last week or two hours ago instead of the current samples.

```
up{job="node_exporter"} offset 1w
```

This would give you the instant-vector value of the node_exporter's up metric from exactly one week ago. The syntax works the same for range-vector outputs like so:

```
up{job="node_exporter"}[5m] offset 1w
```

And for function invocations across range vectors:

```
min_over_time(up{job="node_exporter"}[1h] offset 1w)
```

Finally, myriad operators [5] are supported. These allow you to perform mathematical operations and/or filter the results by the return values themselves and enable a lot of other more advanced functionality I won't have space to get into here. If, for example, we *just* wanted to see the hosts that had been down in the last hour, rather than a complete list of hosts with 0s and 1s to indicate their respective status, we could use a binary comparison operator to filter out the "OK" hosts like so:

```
min_over_time(up{job="node_exporter"}[1h]) < 1
```

That should get you started exploring Prometheus metrics with PromQL, but there's a lot more to learn. The aforementioned talks are a great way to sample some of PromQL's outer limits, and of course the docs [6] are well written and expansive.

Take it easy.

References
[1] https://vimeo.com/341145117#t=24m17s.

[2] https://vimeo.com/341141334.

[3] Node_exporter: https://github.com/prometheus/node_exporter.

[4] Syntax: https://github.com/google/re2/wiki/Syntax.

[5] Operators: https://prometheus.io/docs/prometheus/latest/querying/operators/.

[6] Basics: https://prometheus.io/docs/prometheus/latest/querying/basics/.

Using SQL in Go Applications

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

Many applications work on some set of local data. Even some command line applications need to keep data across invocations. Flat files in a columnar or JSON format work for many cases. However, it can get to the point where a more structured approach can make life easier. SQL databases are the typical next stopping point for a structured approach to data.

Go has a generic interface around SQL with `database/sql` in the standard library. The interface supports drivers which provide the backing to common database technologies. A list of common drivers is available on the Go wiki: https://github.com/golang/go/wiki/SQLDrivers. While most of these are dependent on an external data service, one, SQLite, is not.

SQLite is a self-contained SQL database engine. It stores its data in a file, which makes it easy to embed in local applications. The underlying implementation is in C and has many common language bindings, including several for Go. In Go, this does require cgo support which should, in general, work. However, be aware that it may require additional C compiler binaries to be installed, and cross compilation will require even more.

In this article, we're going to work with the Go SQL interface, specifically the github.com/mattn/go-sqlite3 driver.

The code for these examples can be found at https://github.com/cmceniry/login in the `sql` directory. This code is using dep for dependency management, but this should work with Go modules as well. After downloading the code, you can run each example directly from the main package's directory (`login/sql`) with `go run EXAMPLE/main.go`. The examples use the same example database which will get created in the main package's directory. If you change directories out of that, it may get confused.

Note: As mentioned, you may also need to install SQLite development packages in your environment to complete these examples.

The SQL Interface

The SQL interface provides a simple way to perform the most common SQL methods: open and close a database, execute a Data Definition Language (DDL) or Data Manipulation Language (DML) statement, and perform a query. SQL abstracts away much of the overhead such as connecting to the database, handling connection pooling, and performing connection cleanup.

Since it is an interface, the expectation is to interact with **all** databases the same way, regardless of back-end driver.

Import

The database/sql driver mechanism relies on the blank identifier, _, import. All of the examples use this import format.

```
import (
    "database/sql"
    _ "github.com/mattn/go-sqlite3"
)
```

As normal, the blank identifier indicates to ignore an item. In this case, it's ignoring all of the exported identifiers from the go-sqlite3 package. Our code will not be using any of the possible functions or variables from go-sqlite3 directly.

However, the normal import actions still happen. This includes the variable definitions and initialization mechanisms. Inside the go-sqlite module is an init function. On the first import of a package, it runs this init function. In this case, it registers itself with the database/sql drivers available and makes it available as a back end.

github.com/mattn/go-sqlite/sqlite3.go.

```
func init() {
  sql.Register("sqlite3", &SQLiteDriver{})
  }
```

Other libraries enhance the Go runtime using the blank identifier. The standard HTTP profiling library, net/http/pprof, is another example of a library that you do not call directly. This is a practice that you can use for your code, but use it with caution.

Note: There is a common order to how the init functions (and package-level variables) are run: imported packages and then alphabetical by package file within a package. However, it is still very easy to put yourself in a situation where you are attempting to use them in a different order.

Creating a DB

In our first example, we will create a simple database. The database will be defined with a simple schema:

create/main.go: schema.

```
var schema = CREATE TABLE sample (
  i INTEGER,
  s TEXT,
  t DATETIME DEFAULT CURRENT_TIMESTAMP
  )
```

With this schema in hand, we can start initializing our database. We begin our main function with a call to open the database. The arguments to Open tell the SQL interface which driver to use with which options. The options are specific to the driver—in this case, "read," "write," and "create." We then rely on Go's defer mechanism to ensure that we close the database when we're done.

create/main.go: opencreate,close.

```
func main() {
    db, err := sql.Open("sqlite3", "file:testdb?mode=rwc")
    ...
    defer db.Close()
```

With the open database, we now create our schema in it. We can call the Exec function on the database and pass in the schema string as the argument. Exec returns two values—a result and an error. The result is meaningless for DDL statements, so the main concern here is to receive the error. For the example, handling the error is a simple panic.

create/main.go: exec.

```
    _, err = db.Exec(schema)
    if err != nil {
        panic(err)
    }
```

We will see this same Exec function in the next example and will examine the result.

Insert

Once the database is initialized, we can start feeding data into it. Since this is a new process, we need to reopen the database. In this case, we don't want to create it, so we will leave off the "create" option to open.

insert/main.go: open,close.

```
func main() {
    db, err := sql.Open("sqlite3", "file:testdb?mode=rw")
    ...
    defer db.Close()
```

With the open database, we can add data to it like any other SQL data addition—INSERT. As with the previous example, we use the Exec function to perform the insert. The first argument to Exec is the SQL statement to execute—in this case, a simple insert into the sample table of an integer and a string. While SQLite uses dynamic typing, we're still using parameterized bind variables, ?, instead of combining our values directly with our SQL statement. This provides two large benefits: First, we do not have to handle the type conversion into the statement. (This type handling will show up again in the next example, query.) Second, this form is much less susceptible to SQL injection attacks. The remaining arguments to Exec are bound to the respective positional ?. Exec is variadic in that the number of arguments is dependent on the SQL statement.

insert/main.go: query.

```
    res, err := db.Exec(
        "INSERT INTO sample (i, s) VALUES (?, ?)",
        2,
        "2",
    )
```

If there is a syntax or back-end issue, an error will be returned. After checking the error, we also want to confirm how many rows were inserted. For inserts, this may not matter as much, but in other cases (SET) it can indicate an issue in data or logic. We obtain the numbers of rows inserted with the RowsAffected method of our result.

insert/main.go: rows.

```
	affected, err := res.RowsAffected()
```

With the value in hand, we can print it out and visually inspect it.

insert/main.go: print.

```
	fmt.Printf("%d row(s) inserted\n", affected)
```

The output of this should be fairly simple:

```
$ go run insert/main.go
1 row(s) inserted
```

RowsAffected is really the only indicator of the impact of your SQL statement, and may or may not be interesting depending on your situation. If you alter the insert statement to include additional VALUES pairs, it will increase accordingly. It can also be more than one for SET statements which affect multiple lines. It can even be zero in the cases where no rows match, indicating a logic or data error.

Query

In our final example, we're going to pull previously inserted data back out of the database. As in the previous insert example, we will see inferred type conversion.

As before, we start the main function by opening the database.

query/main.go: open,close.

```
func main() {
	db, err := sql.Open("sqlite3", "file:testdb?mode=rw")
	...
	defer db.Close()
```

Next we use the Query function to submit our SQL statement. Query behaves very similarly to Exec. It is variadic. The first argument is our SQL query statement, which may contain bind variables, ?, in the WHERE clause. Any additional arguments are bound to their positionally respective bind variables. Yes, the DATE(t) <= DATE(?) is a bit superfluous but is included for demonstrative purposes.

query/main.go: query.

```
	rows, err := db.Query(
		SELECT i, s, t FROM sample WHERE DATE(t)
<= DATE(?), time.NOW(),
		)
```

If the query is successful, a result set is returned. Behind the scenes, the SQLite package creates a cursor which holds the location of the data—relative to both the query result processing and its location in the database file. To avoid consistency issues, this also locks the database until this query is complete. The indicator that the query is complete is with a Close on the result set. For this simple example, we can release the statement when we finish the function, so we use Go's defer mechanism.

query/main.go: stmtclose.

```
	defer rows.Close()
```

Now we can process the returned rows by iterating through the rows. To move through the cursor, we call the Next function. The Next function updates the underlying cursor information for the next unprocessed row. The Query does not do this initially, so a first call to Next is required to even begin to access data. This also allows us to wrap it all in a for loop.

query/main.go: next.

```
	for rows.Next() {
```

With the cursor properly in place for our next row, we can pull all of the values out of the row. We need a place to store the data local to our code, so we start by defining some variables. We then pass pointers for those variables into the Scan function, which will set them as appropriate. In addition to providing a place for the data, using pointers to our variables allows for Scan to cast the row values into the appropriate type. Scan is also variadic, and the position of arguments to it are the respective positions for the fields in the SELECT statement.

query/main.go: scan.

```
		var i int64
		var s string
		var t time.Time
		err := rows.Scan(&i, &s, &t)
```

Now we can print the results out.

query/main.go: printout.

```
		fmt.Printf("%s: %d %s\n", t, i, s)
```

An example output of this looks like:

```
2019-06-15 13:21:06 +0000 UTC: 1 1
2019-06-15 13:21:11 +0000 UTC: 1 1
2019-06-16 18:03:38 +0000 UTC: 1 1
2019-06-17 04:44:27 +0000 UTC: 1 1
```

Conclusion

In these examples, we've explored the `database/sql` package and an accompanying driver for it, the `github.com/mattn/go-sqlite3` for SQLite. In addition to what has been demonstrated here, the `database/sql` package and the various back ends provide other features—interrogating the columns and arbitrary results, handling timeouts with `Context`, direct creation of prepared SQL statements, and many more. You can dig into the Go SQL interface at http://go-database-sql.org.

Sometimes data gets complex enough that writing flat file parsers becomes tedious. Sometimes you have to interact with an existing application database. Go's SQL interface provides a simple way to interact with many different types of SQL databases. I hope this has given you a good basis for using SQL when needed. Good luck, and Happy Going.

USENIX Supporters

USENIX Patrons
Bloomberg • Facebook • Google • Microsoft • NetApp

USENIX Benefactors
Amazon • Oracle • Two Sigma • VMware

USENIX Partners
Cisco Meraki • ProPrivacy • Restore Privacy • Teradactyl • TheBestVPN.com

Open Access Publishing Partner
PeerJ

For Good Measure

Is the Cloud Less Secure than On-Prem?

DAN GEER AND WADE BAKER

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

Dr. Wade Baker is an Associate Professor in Virginia Tech's College of Business, teaching courses for the MBA and MS of IT programs. He's also a Co-Founder of the Cyentia Institute, which focuses on improving cybersecurity knowledge and practice through data-driven research. Prior to this, Wade held positions as the VP of Strategy at ThreatConnect and was the CTO of Security Solutions at Verizon, where he had the great privilege of leading Verizon's annual Data Breach Investigations Report (DBIR) for eight years. wbaker@vt.edu

So you got to let me know,
Should I stay or should I go?
—The Clash

According to Deloitte's Chief Cloud Strategy Officer, "[2019] is the year when workloads on cloud-based systems surpass 25 percent, and when most enterprises are likely to hit the tipping point in terms of dealing with the resulting complexity" [1]. Given the nature of For Good Measure (this column), it may surprise you that it wasn't the 25 percent statistic that caught our attention in Deloitte's quote; it was reference to a "tipping point" where "dealing with the resulting complexity" in the cloud begins to negatively affect security. So we ask, do we see evidence that this is occurring? Are the rate of security exposures in the cloud higher than on-prem?

Conducting such an analysis requires data on security exposures affecting both on-prem and cloud-based hosts. RiskRecon [2] was kind enough to provide a sanitized data set derived from their efforts to provide visibility into third-party cybersecurity risk. For each organization analyzed, RiskRecon trains machine-learning algorithms to discover Internet-facing systems, domains, and networks. For every asset discovered, RiskRecon analyzes the publicly accessible content, code, and configurations to assess system security and the inherent risk value of the system based on attributes such as observable data types collected and transaction capabilities. The data set supplied by RiskRecon spans 18,000 organizations and over five million hosts yielding 32 million security findings of varying severity. Digging in, what can we determine about what organizations are seeing with respect to security complexities in the cloud vs. on-prem?

Figure 1 offers a bird's-eye view of our leading question. Each dot represents an organization in our data set, with a sufficient number of hosts in both on-prem and cloud environments to support this test. Their position on the grid is the intersection of the percentage of on-prem (horizontal) and cloud-based (vertical) hosts that have high or critical security findings. So, for example, the firm indicated by the arrow has an on-prem exposure rate of approximately 8% compared to a much lower 0.2% in the cloud. Organizations marked by blue dots (below the line) indicate they have comparatively fewer security issues when in the cloud. Green dots (above the line) represent firms that appear to be better off on-prem. Overall, there's a 60/40 split between organizations that operate with fewer issues on-prem (60%) vs. in the cloud (40%).

We infer from these results that the question of security destiny in the cloud is not predetermined. If you go, there may indeed be trouble; if you stay it may or may not be double. And it very well could be half.

Unfortunately, we do not have historical data available to determine whether those numbers are trending toward or away from a 50/50 "tipping point," but we were able to identify some

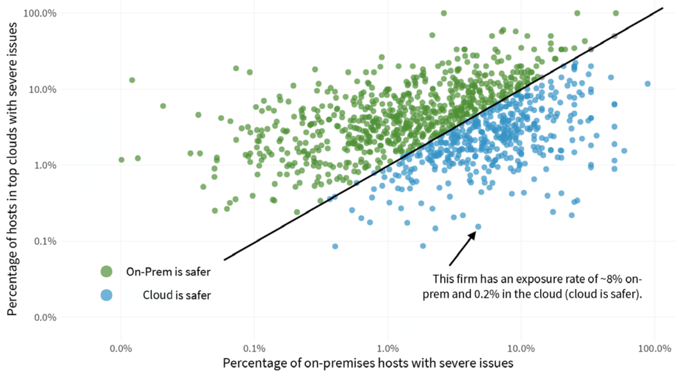


Figure 1: Comparison of hosts with severe findings in on-prem vs. cloud environments. Dots above the line indicate firms that have comparatively fewer security issues when on-premises.

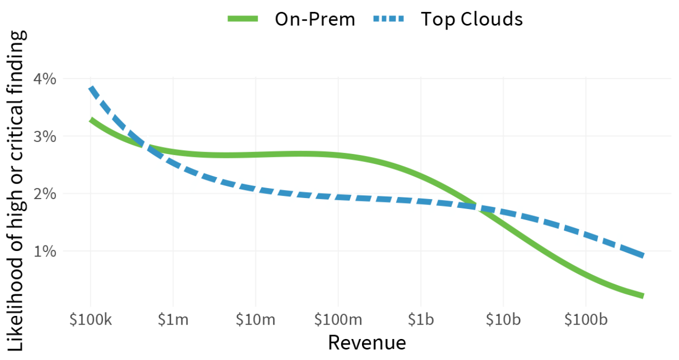


Figure 2: Models comparing exposure rates on-prem vs. cloud by organization size (annual revenue in log scale)

factors that affect a firm's likelihood of landing on one side of that line or the other. We discuss three of these factors below.

The Deloitte quote provides inspiration for the first factor we wanted to investigate. There's an implied statement that higher cloud adoption leads to a tipping point where added complexity affects security. Do we see evidence in the data that such a tipping point exists? To test that, we compared the rate of high and critical security findings in the cloud with the percentage of all hosts in the cloud for each organization. The result was a statistically significant but very low positive correlation (r=0.07) between those two variables. In other words, security exposures do increase as organizations put more and more hosts in the cloud...but not by much and only gradually. Not exactly evidence in favor of a tipping point.

The second factor is organization size as measured by annual revenue. We'd like to more directly measure characteristics like resources, IT complexity, and security capability, but size is the best proxy we have for those things. The question in view here is whether firm size (revenue) increases or decreases the likelihood of severe security exposures in cloud and on-prem hosts. Figure 2 constructs a regression model to test this correlation.

Let's first observe the general trend of decreasing likelihood of exposure as revenues grow for both on and off-prem hosts. This may reflect increased resources and maturity but may simply be an artifact of scale. It's almost inevitable that the likelihood of any single host being exposed declines as total population grows in larger enterprises.

Beyond that general trend, Figure 2 reveals some interesting "tipping points" between security in the cloud and on-prem. According to the model, organizations with annual revenues between $1M and ~$5B operate a little more safely in the cloud. The opposite holds true for firms outside that range—the really small and the really big. Might this imply that fast-growing organizations will want to use the cloud preferentially, but not small organizations and not giant, established players?

The third and final factor looks at the effect of consolidation vs. diversification in the cloud. In other words, is it better from a security perspective to consolidate hosts into one (or a small number of) cloud provider(s) or to spread services across many providers? Figure 3 reflects the data's answer to that question.

The "bars" in Figure 3 are actually made up of "dots" representing the 18,000 firms in our sample. We visualized it this way to emphasize the high degree of variation among organizations, especially toward the left side. But our focus is on the trendline, which turns out to be quite interesting. It suggests that the rate of severe findings is at its highest when cloud diversity is at its lowest. As organizations use more cloud providers, that rate drops steadily...to a certain point. Firms with four clouds exhibit one-quarter the exposure rate of those with just one cloud provider. Having eight clouds drops that rate in half again. Beyond that, security issues level off and even begin to rise among hyper-diversified cloud users. We can't help but see a kind of "tipping point" here: there's a point where consolidation and diversification find balance in the cloud, and that point varies from firm to firm. Echoing Deloitte, is that balance where complexity and the ability to manage it are themselves in balance?

One bit of caution regarding Figure 3: all kinds of factors are at play here that we cannot consider in our analysis. For instance, perhaps many of the firms with only one cloud provider are simply experimenting. This may reflect various stages of cloud maturity from left to right rather than the effects of consolidation vs. diversification. Given what we learned from Figure 2, one may hypothesize that this simply reflects the effects of organization size on exposure rates (the assumption being larger enterprises use more clouds). We included both variables in our analysis, but the number of cloud providers alone was the significant one.

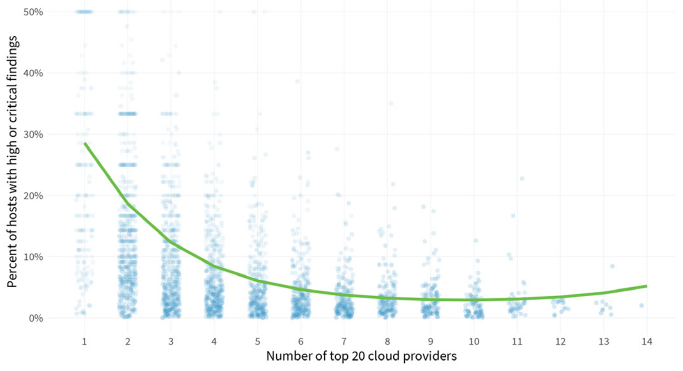


Figure 3: Rate of security exposures among hosts by number of cloud providers

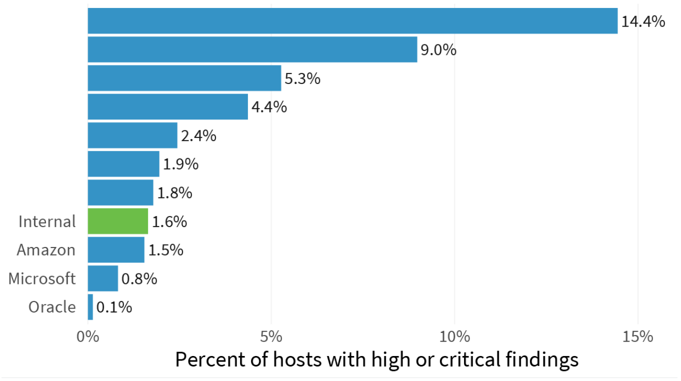


Figure 4: The spread in insecurity across major cloud providers

Of course, not all clouds are the same, either, as illustrated by Figure 4. Here we compare the prevalence of severe security findings among the top cloud providers. "Top" here refers to adoption. The clouds represented in Figure 4 accounted for over 90% of the cloud-based hosts in our data set. We also include the comparable rate for internal (on-prem) hosts. To give some sense of familiarity, only the three clouds with the lowest exposure rates bear labels. The point is not whether Cloud A is "better" than Cloud B, but rather that substantial variation exists among them. We cannot explain why the provider at the top of the list has an exposure rate 144× that of Oracle, but we suspect it has a lot to do with the nature of those clouds and how they're used. Perhaps systems in Oracle's cloud primarily host major enterprise applications that are rigorously maintained by their owners. Perhaps the unnamed cloud on top plays home to a higher share of SMBs and/or test workloads. We simply don't know. But we can safely conclude that scattering your hosts randomly across cloud providers is unlikely to achieve positive outcomes. If you do go, "where?" is the next—and equally important—decision.

None of this discussion deals with common-mode failure among cloud suppliers such as the Meltdown [3] and Spectre [4] issues announced in January 2018. Rather, it asks a fuzzy question: what is the causal relationship here? Is it size? Is it diversity? Is it complexity in some other sense? Can the causal mechanism be identified and sufficiently well understood to drive policy? What more data would help (or would more data help)?

As with other budding romances, "Should I stay or should I go? (Don't you know which ~~clothes~~ clouds even fit me?)"

References

[1] D. Linthicum, "Cloud Complexity Management (CCM): A New Year, a New Problem": https://www2.deloitte.com/us/en/pages/consulting/articles/cloud-complexity-management-a-new-year-a-new-problem.html.

[2] https://www.riskrecon.com/.

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

[4] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution": https://arxiv.org/abs/1801.01203.

/dev/random
Layers

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

Once upon a time, there was only one layer: the operating system. It was your first and best means of exchanging ones and zeroes with the processor, the mystical heart of your computer. You wrote code, compiled it or fed it to an interpreter, and something interesting usually happened. Well, my code *always* made something interesting happen, but my threshold for interesting includes power cycling and printers with a pronounced tendency to spit out page after page of nonsense. I also evinced a preternatural knack for triggering crash dumps that literally caused the machine, not to mention any clued-in onlookers, to shudder.

As I've said (too) many times, my digital heyday came during a previous geological era as reckoned in the accelerated chronology of computing. The code I hammered out looks primitive and ragged now, much like my wardrobe and finances. Those were the days when shareware came on floppies obtained at music-turned-discount-software stores in the mall, before every new computer game release required the latest supercharged video card to run at better than two frames per second. Those were the days when hacking was, at worst, criminal mischief—not supervillainy.

The first incursion of layers was the graphic user interface, intended to make navigating the operating system a little easier for people who hadn't the patience to commit dozens of program options and arguments to memory. This was in no way mandatory: those of us who liked the cryptic nature of the command line beast could still accomplish whatever we needed without getting our fingers GUI. But then, gradually, virtualization and emulation and compartmentalization began to creep into our systems like vampires seeking refuge from a clear summer's afternoon. After a while it was no longer at all apparent what floor of the computational skyscraper you were working on.

I retired some years ago from looking over the shoulders of system administrators to verify that they'd implemented at least the minimal security measures mandated by government standards. Toward the end of that intellect-numbing career, virtualization was already complicating lives. Did certain security settings apply only to the underlying operating system, for example, or did they need to be duplicated for every virtual machine instance? In those situations where one operating system had significantly different mandated security parameters from another, but both were instantiated in virtual machines on the same box, which one's security settings took precedence?

The virtualization rabbit hole now goes much deeper from those comparatively halcyon days. Today the concept of a single operating system directly supporting applications in the enterprise seems as quaint and whimsical as a racoon coat in a rumble seat. Not all of these layers are distinct operating system images; it's true. Some of them, like Docker layers, are just topological metaphors for processes being run as a suite within a lightweight container. I think I just got a charley horse in my frontal lobe from typing "topological metaphors." Ow.

Anyway, this layering mania got me to wondering: just what are these people running from? What is it about the base operating system that makes them so uncomfortable? Are they embarrassed by the belief that people regard them as unsophisticated because they only have a couple of layers going? Or is it just that they were exposed to the OSI model during their formative years and now feel that multiple layers are necessary for things to work?

Speaking of network models, it seems to me that we'll need to reinforce the TCP/IP stack in order to bear the weight of all these new layers. Maybe stick some rebar in there or something. Come to think of it, perhaps it's also time to establish an entirely new nomenclature that reflects today's puff pastry networking reality. After all, continual change for change's sake is what technical advancement is really all about, right? No novelty, no progress.

We'll start at the bottom, because that's where I'm most at home. The current lowest level is the Physical layer (OSI Layer 1), so-called because it deals with wires and adapters and those little cylindrical doodads on some cables that you don't know what they do—physical objects, in other words. I propose we rename this the *Fiddly Bits* layer, since one out of one columnist surveyed declared this to be a lot more descriptive and accurate.

Next up is the Data link layer (OSI Layer 2). Data link sounds like some rural ISP that set up shop using old satellite dishes and routers they dug out of the dumpster behind Fry's. I think a better name for something that connects data paths is the *Drawbridge* layer. Above the Drawbridge we come to the Network layer (OSI Layer 3). Here the bits really hit the fan, what with packets and frames buzzing around like flies over garbage. For that reason, I think of it as the *Landfill* layer.

The Transport layer (OSI Layer 4) is where those bits get packaged and shipped off to market, so I call it the *Loading Dock* layer. The Session layer (OSI Layer 5) is mostly concerned with keeping lines of communication open, so we'll think of it as the *Switchboard* layer. Layer 6, the OSI Presentation layer, is where one format gets converted to another; I'll call this the *Thesaurus* layer. Finally, there is the Application layer (OSI Layer 7). This is sort of a catchall area for everything else that needs to happen to make software and user care about one another, so to me it is the *Kitchen Drawer* layer.

There you have the layers of the RGF model: Fiddly Bits, Drawbridge, Landfill, Loading Dock, Switchboard, Thesaurus, and Kitchen Drawer. The old "All People Seem to Need Data Processing" mnemonic doesn't work any longer, admittedly, but at least these are layer names that evoke actual mental images, not those sterile engineering labels your brain has to massage into real language before they mean anything to you. I doubt my terminology will make it into an RFC, unless it's an April Fool's submission, but that's not my concern. I'm just the idea guy.

"Layer," incidentally, can also refer to a hen that actively produces eggs. Eggs, like operating systems, have shells which both protect and provide access to the underlying contents. Computer science and animal husbandry: working hand in, um, talon for a better tomorrow.

Cluck().

Book Reviews

MARK LAMOURINE AND RIK FARROW

Continuous Delivery
Jez Humble and David Farley
Pearson Publishing, 2011, 464 pages
ISBN 978-0-321-60191-2

Reviewed by Mark Lamourine

The ideas of continuous integration (CI) and continuous delivery (CD) are fairly common, almost mainstream, today. CI originated in Extreme Programming DevOps in the mid to late 1990s, becoming more formalized over the following decade. CD was for a long time an afterthought. Humble and Farley offer what appears to be the first attempt to present CD as its own discipline.

The authors lay out all of the moving parts of a CD system and they explain why they are there and how they interact. Agile methods were developed as a practical response to the failure of earlier software management methods to account for human psychology and the realities of business and life. A CD system depends on the interactions and feedback from the components. The authors give both the theory and practice for each component so that the reader will understand how each is important to the function of the whole.

In many ways *Continuous Delivery* compares with Limoncelli, Hogan, and Chalup's *The Practice of System and Network Administration*. Humble and Farley treat the entire ecosystem of a CD system, from definition and implementation to maintenance and life-cycle operations.

There are a few ways in which *Continuous Delivery* shows its age. The authors list a number of tools that are no longer the first choice. They discuss CVS and Subversion and explicitly mention the need to disable mandatory locking for CI operations. When *Continuous Delivery* was published in 2011, Git had only existed for five years and GitHub for two, and neither had achieved the acceptance that they have now. The authors still refer to configuration management tools such as Cfengine, and there is no mention of Ansible or Salt. Other than the fact that recent configuration management tools are deemphasizing defining a state model in favor of just reexecuting a set of operations and the advent of software containers that replace long-lived hosts and VMs, the concepts and solutions remain applicable.

Continuous Delivery provides all that a new developer needs in order to understand the goals and motivations for a well-run CD system. For the advanced reader, it fills in the gaps that are the inevitable result of organic learning, providing context and completeness. It does stand the test of time.

Deep Learning and the Game of Go
Max Pumperla, Kevin Ferguson
Manning, 2019, 531 pages
ISBN 978-1-617-29532-4

Reviewed by Mark Lamourine

"Deep learning" is a relatively new term, and it partially supersedes an older term I'm more familiar with: "neural networks." Today, neural network refers to a technology, a well-defined software structure that takes some inputs and produces some outputs. Deep learning is a technique for using neural networks to do a set of tasks that are difficult, using conventional prescriptive programming.

The term "deep learning" is strongly associated in the mind of the general public with AlphaGo, the research project by DeepMind (now part of Alphabet). The game of Go was long thought to be intractable for AI because, when compared with chess, the move-branching factor is orders of magnitude higher. IBM's Deep Blue managed to beat the reigning chess champion, Gary Kasparov, in the late 1990s using primarily brute force branch search and some clever hand-programmed move ranking and pruning algorithms. Humans observed the play and tweaked the search and pruning rules until the system's ability exceeded the best human's.

In *Deep Learning and the Game of Go*, the authors use Go and the model provided by AlphaGo to introduce the reader to deep learning as a concept and a practice. AI research has long used games as well-known problem spaces to explore learning techniques. Games remove the messiness of the real world, and they have well-defined goals, rules, and states. This makes for a nice clean teaching/learning environment. The authors take advantage of this as well.

Deep Learning and the Game of Go provides some foundational context before jumping in but is light on theory and mathematics, saving those for the appendices. The approach is very practical, offering the reader examples and sample code from GitHub to work and play with. The method is hands-on, so the reader will build experience through contact.

This is also a weakness. At the end, the reader has only explored a single deliberately clean problem space. As a beginning, it suits, but readers must realize where they stand at the end of the book. They can choose to stop or to continue into the complexities that real-world deep learning entails. There are other books for that.

Deep Learning with Python
François Chollet
Manning Publications, 2018, 445 pages
ISBN 978-1-617-29443-3

Reviewed by Mark Lamourine

One of the leading pure AI fields is called "deep learning." It has revitalized the use of artificial "neural networks" (poorly named). Neural networks were first created in the late 1990s but languished from insufficient CPU power and imagination. A lot has happened since then, and neural networks have seen a revival. François Chollet wants to tell you all of it.

In *Deep Learning with Python,* Chollet tries to provide a working knowledge and code samples to allow the reader to create and verify a variety of deep learning experiments (as he calls them) using modern AI techniques based on convolutional neural networks.

After the too brief history, basically everything was new to me. In some ways this book feels like a detailed syllabus for a year-long graduate-level course in deep learning techniques and software. The book is structured around the Keras deep learning library. Python has a long history in scientific calculation and numerical systems due to the ability to create compiled libraries. The math and science communities have taken advantage of this to provide high performance libraries of domain-specific functions that can be used by a scripting language. This results in the ability to fast prototype the work logic using established, stable optimized algorithms.

Chollet does offer a bit of theory and context at the beginning, but it becomes clear after the first few chapters that he is assuming significant prior knowledge on the reader's part. Each chapter is more about the set of mathematical tools that the library provides and how to use them than it is about how they work and what the results mean. For someone first approaching deep learning, this might be overwhelming. For a researcher familiar with the math, but who just wants to use the tools to ask questions in their problem space, this is a breezy survey.

I did learn a lot despite being largely in over my head with the jargon and algorithms. The fact that there are flavors of neural networks and even flavors of algorithms for each layer of a network was new. I hadn't considered the implications of simple linear networks, with forward learning and feedback versus more complex network topologies. I don't expect to become an AI researcher, but I now have a better chance of understanding what they've achieved when I see it.

Fall, or Dodge in Hell
Neal Stephenson
Harper Collins, 2019, 800 pages
ISBN 978-0-062-45871-1

Reviewed by Rik Farrow

What might it be like to experience the Singularity, at least the part where your personality lives on beyond your body? Stephenson takes on this challenge of eschatology, giving some characters from *Reamde* a second chance at novel life and death.

Stephenson gets some of the technology right: simulating a single thought process will take enormous amounts of processing power, networking, storage, and just plain power. Doing it for everyone will involve taking over the earth. But the first in are, of course, the billionaires and their friends and family, and having the scions of industry involved affects everything. Their memories of real life may be partial, but the personalities are as overpowering as ever.

Halfway through the book, Stephenson gets a little biblical on us, but don't despair. His Jehovah is more like the one in the Book of Job, and the second coming, leading to the Fall mentioned in the title, is definitely twisted.

Stephenson has a knack for creating interesting, somewhat wacky, but wholly believable characters. Sometimes he subtly does things that I found disturbing without the cause being blatantly obvious.

While I don't expect to awaken in the cloud, Stephenson does a good job of imagining what it might be like and is still thoroughly entertaining—most of the time. Some of the world building does get tedious, but it's definitely a good read overall.

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

2018 Constituent Survey Results

Liz Markel, Community Engagement Manager

Last year we reached out to the many people we serve—our members, our conference attendees, and those who have expressed interest in our activities—and asked you to take several minutes to respond to our community survey—the first of its kind in five years. More than 1,000 of you responded, sharing information about yourselves and your thoughts on a variety of questions related to membership benefits, the communities you participate in, how well we're doing with making our mission a reality, and more topics relevant to our mission.

With your responses we were able to:

- Create baseline measurements for key metrics such as community demographics and USENIX's perceived performance with respect to its mission.
- Gather data to help inform upcoming decisions by USENIX staff and leadership.

We appreciate everyone who took the time to complete the survey! I'd like to share some highlights from the survey results, and also let you know about some changes we're implementing based on those results.

I also want to take this opportunity to remind you that my inbox is always open for conversations about these results, general suggestions, or other topics that you might like to chat about. You can reach me via liz@usenix.org.

Demographics: Who Is USENIX?

Demographic questions served several purposes within the context of this survey. First, it provided a profile of our constituents in aggregate: who they are, where they come from, and a brief but illuminating glimpse into their professional lives.

Second, demographic questions provided an important benchmark for diversity and inclusion initiatives. Our overarching goal is to maximize the accessibility and welcoming, inclusive environment at our conferences and throughout other areas of our organization's work. Anecdotally, we feel we are generally successful in this area, although there is always more work to be done, of course. However we wanted statistics to back up those anecdotes. We also want to track our progress in this area over time. In order to do this, we ask questions about things such as race and gender, and only use that data in aggregate.

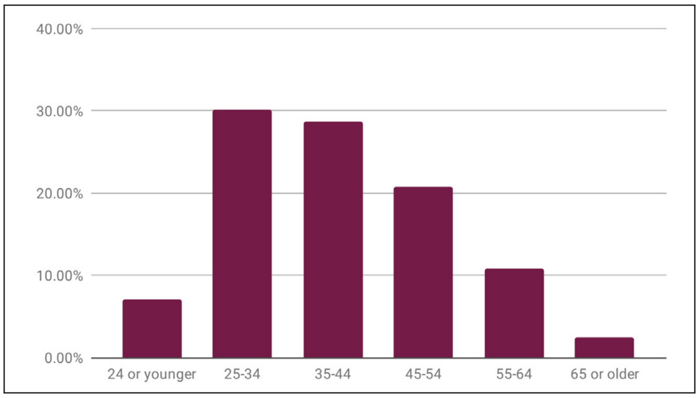
Demographic questions are also a valuable tool for cross-referencing responses to other questions. Where differences exist, we can explore the reasons for those differences, and consider if and how we might address those differences. For example, if there were a significant discrepancy between employers' coverage of professional development costs when comparing responses from self-identified males with those from self-identified females and non-binary gender, we would consider how this impacts our Diversity Grant program. (On that subject, for respondents who said that their employers cover 100% of the costs of conference travel and participation, 52% of those respondents were male, and 44% were female.) In order to conduct this analysis, we must ask questions about race and gender. Results are, again, examined solely in aggregate.

While we value these metrics, we also recognize that specific demographic elements such as gender or race are complex. We are open to dialogue around this topic that supports our goals to track our progress in a meaningful, metric-driven way, while also demonstrating respect for all members of our community. If you have feedback about our approach, including ideas of better ways to gather and assess this data, please let me know.

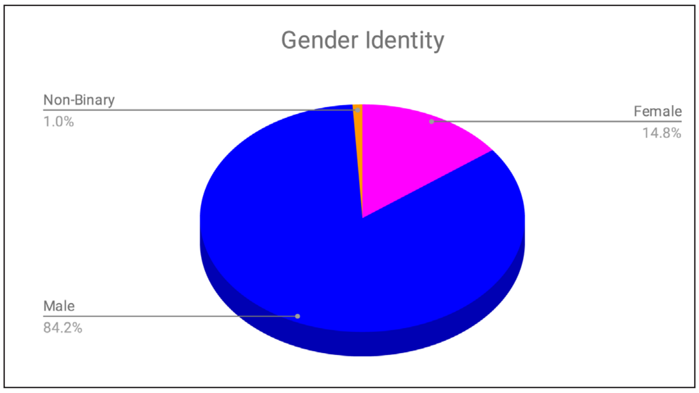
Here's an overview of our demographic results:

Employed	73.12%
Student (Graduate-level Program)	18.42%
Self-Employed/Freelance	3.89%
Student (Undergraduate)	1.23%
Other	1.23%
Unemployed	1.04%
Retired	1.04%

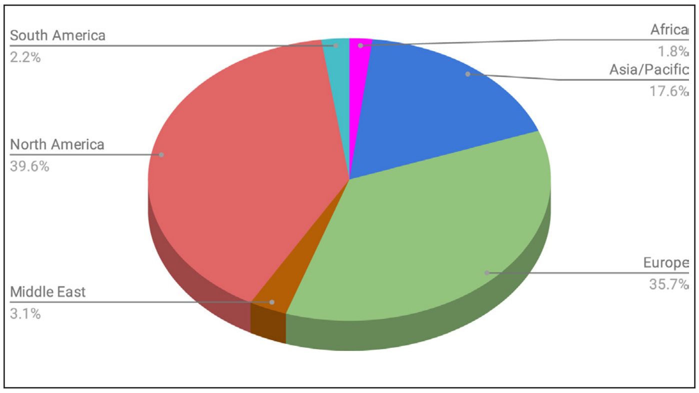
Employment: The majority of respondents (73%) are employed; student respondents comprised just under 20% of all respondents (18% graduate students).



Age: The majority of respondents were between ages 25 and 44.



Gender Identity: Almost 16% of respondents identified as female or non-binary. For the Non-Binary category, respondents could enter their own response.



Geography: Respondents came primarily from North America and Europe.

Mission fulfillment and leadership

At the core of a nonprofit organization is its mission. The survey provided an opportunity to gauge our performance on the four parts of our mission to:

- Foster technical excellence and innovation
- Support and disseminate research with a practical bias
- Provide a neutral forum for discussion of technical issues
- Encourage computing outreach into the community at large

We asked respondents to rate our performance in these areas on a scale of 1 to 4, where 1=needs significant improvement, and 4=we're doing amazing work. The weighted average for each of these was:

	Weighted Average
Foster technical excellence and innovation	3.33
Support and disseminate research with a practical bias	3.3
Provide a neutral forum for discussion of technical issues	3.21
Encourage computing outreach into the community at large	2.99

We were thrilled to see these results, and to have our hard work affirmed by you—the people for whom we're doing the work. Of course, there's still room for improvement here. Those improvements may come from our actual efforts, or they may come from greater emphasis on the work we are currently doing. We'll work on both of these aspects and hope for even higher marks on the next survey. As a reminder, you can always contact me directly with any questions or suggestions.

Communities

When we think about the people who comprise USENIX's broad community of advanced computer systems professionals and their related sub-communities, we tend to think of them in terms of our conferences. However, we know that not everyone involved with USENIX attends our conferences, and that you may think of your professional identities differently than we do.

Consequently, we asked two questions on this survey pertaining to community membership:

1. Respondents were asked to indicate which conferences they had attended; they had the opportunity to indicate their affiliation with the conference community even if they had not attended the conference.
2. On a separate question, respondents were asked to select all of the professional communities to which they felt they belonged, including file and storage systems researchers or practitioners; system administrators or engineers; networked systems researchers or practitioners; systems researchers or practitioners, broadly defined; security, usability, and privacy researchers or practitioners; site reliability engineers; security researchers or practitioners.

A rough analysis of the overlaps that appeared in this second question were surprising to us. For example, of those who selected "security researchers or practitioners" as a community to which they belong, 51% also selected the community of "systems administrators or engineers". These particular overlaps were unexpected, and required further exploration. Did respondents select both of these answer choices because their roles straddle both of these areas, or they are professionally adjacent to each other? Was our grouping of types of roles too broad, such that a sysadmin who is responsible for security as one of many aspects of a job role would thus identify as a security practitioner for that reason? Are these fields more closely related than we anticipated, and are there implications for conference content to better serve people who function across two of these communities? We are also considering that the surprising results may have to do with our survey design: could we have asked the question in a different way?

We need your feedback! What do you think about these overlaps? Do you have anecdotal observations that support these results?

Communications and Connections

We use many tools to broadcast information about USENIX activities. The survey asked respondents to select which ones they use to learn about USENIX events, and to check all that apply. The top responses were emails from USENIX, the USENIX website, and friends/colleagues.

We need your feedback! I have spent time improving the content and aesthetics of the email newsletter over the past year: what you do you think about these improvements? I am also exploring your responses to separate questions about why you visit the website and what you think can be improved, and how we might implement some of those updates.

We need your feedback! Is there something we can do to facilitate sharing information about USENIX news and events between you and your colleagues? I am open to your suggestions about how to make this process easier for you.

Speaking of sharing, we also asked about your preferred method of connecting with your professional colleagues, and to check all that apply. Close to 90% of you said you prefer to connect in person. Online chat and social media were popular choices, but nowhere near as popular as in-person connections. This data backs up our anecdotal evidence that attending our conferences and engaging with others is a worthwhile investment.

We need your feedback! How can we support you and/or your colleagues to make conference attendance possible? We already offer Student Grants and Diversity Grants to cover registration and travel costs, help facilitate room sharing, and shift the locations of our conferences to provide the opportunity for more convenient attendance. I'm looking for outside-the-box ideas beyond these—perhaps something you've seen at other conferences that has been successfully implemented and might align well with our existing processes.

Membership

Of those who responded to the survey, 41% are currently members, with an additional 18% having been members previously. Both members and non-members were asked about USENIX membership benefits and pricing.

The most noteworthy outcome of this portion of the survey was the high value respondents assigned to open access to papers, proceedings, and video content from our conferences. It is important to note that open access has not been (and will not be) connected to membership in any way; our content will continue to remain free and open to the public. However, membership dues provide financial support for the organization as a whole and thus help underwrite the costs of producing and sharing these materials.

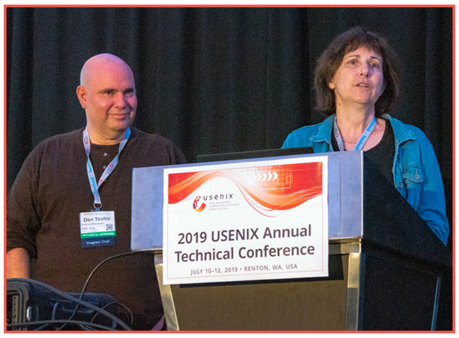
Based on these results, there are exciting changes to USENIX membership in the works that will increase access to membership and increase the value of membership for all levels of contributors. We are working on the behind-the-scenes logistics of these changes, and will announce the details once we are close to a launch date.

A Treasure Trove of Data

There's much more data from the survey—too much to summarize here—but it's already been useful as a resource and guiding light for all types of decisions. We are looking forward to continuing to use this information moving forward, and to make surveys a regular part of your opportunity to provide feedback and tell us how we're doing. If you didn't have an opportunity to complete this survey, I hope you'll take the time to complete the next one! We'll announce it in the USENIX News email when it launches in 2020.

2019 USENIX Annual Technical Conference

2019 USENIX Flame Award winner Margo Seltzer (left) and Awards Committee member Angela Demke Brown.



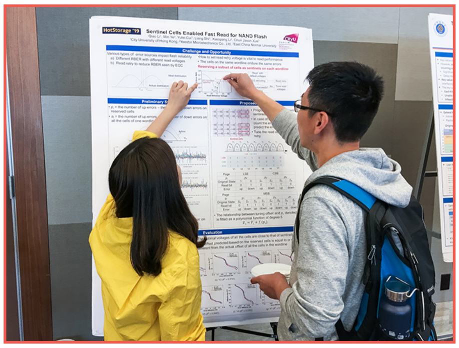
USENIX ATC '19 co-chairs Dahlia Malkhi and Dan Tsafrir deliver their opening remarks.

USENIX ATC '19 attendees take advantage of the conference hotel's outdoor spaces to engage in conversation.

Some of the Student Grant and Diversity Grant recipients who attended USENIX ATC '19.

Remzi Arpaci-Dusseau, University of Wisconsin—Madison, delivers his USENIX ATC '19 keynote address, "Measure, Then Build."

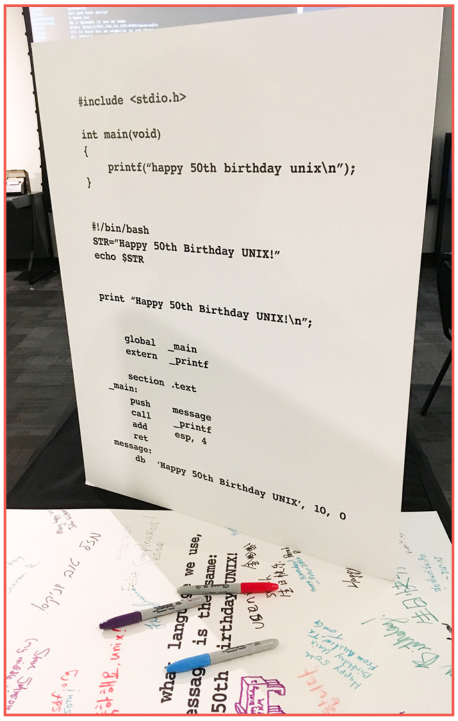
We were fortunate to have slightly overcast skies for the USENIX ATC '19 Luncheon, creating the perfect conditions for eating outdoors.



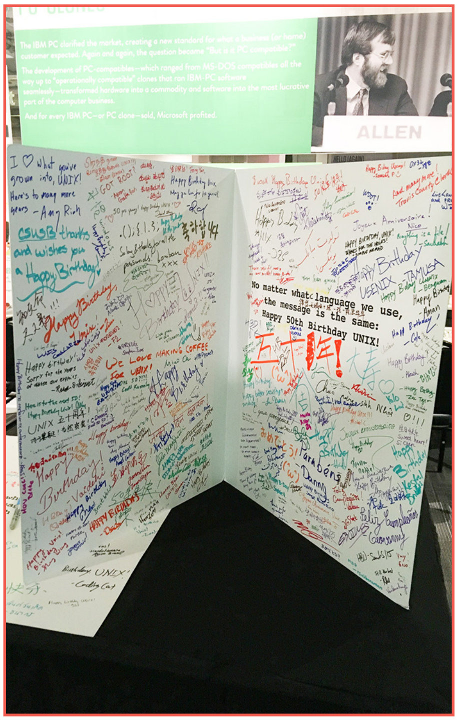
Poster sessions for USENIX ATC '19 and its co-located events were lively opportunities to explore research and engage in conversation with the researchers.

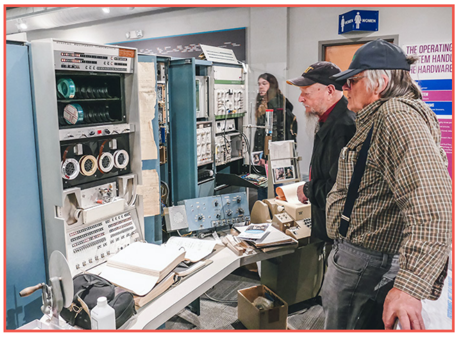
Celebrating UNIX's 50th Anniversary: UNIX Exhibit Preview & Meetup at the Living Computer Museum + Lab

USENIX ATC '19 attendees ventured to downtown Seattle for a sneak peek at the UNIX 50th anniversary exhibit at the Living Computer Museum + Lab. Thanks to LCM+L for hosting—we highly recommend visiting them on your next trip to Seattle—and special thanks to the LCM+L team for the event photos! Thanks to everyone who signed the birthday card for UNIX, too.



```
#include <stdio.h>

int main(void)
{
    printf("happy 50th birthday unix\n");
}

#!/bin/bash
STR="Happy 50th Birthday UNIX!"
echo $STR

print "Happy 50th Birthday UNIX!\n";

    global  _main
    extern  _printf

    section .text
_main:
    push    message
    call    _printf
    add     esp, 4
    ret
message:
    db  'Happy 50th Birthday UNIX', 10, 0
```


ALLEN
No matter what language we use, the message is the same: Happy 50th Birthday UNIX!
Happy Birthday



Save the Dates!

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

USENIX ATC '20

2020 USENIX Annual Technical Conference

JULY 15–17, 2020 • BOSTON, MA, USA
www.usenix.org/atc20

The 2020 USENIX Annual Technical Conference will bring together leading systems researchers for cutting-edge systems research and the opportunity to gain insight into a wealth of must-know topics, including operating systems; runtime systems; parallel and distributed systems; storage; networking; security and privacy; virtualization; software-hardware interactions; performance evaluation and workload characterization; reliability, availability, and scalability; energy/power management; bug-finding, tracing, analyzing, and troubleshooting. Paper submissions are due Wednesday, January 15, 2020.

Program Co-Chairs

Ada Gavrilovska
Georgia Institute of Technology

Erez Zadok
Stony Brook University

www.usenix.org/atc20

OSDI 20

14th USENIX Symposium on Operating Systems Design and Implementation

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

OSDI brings together professionals from academic and industrial backgrounds in what has become 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.

Program Co-Chairs:
Jon Howell, *VMware Research*
Shan Lu, *University of Chicago*

The Call for Papers will be available soon.

www.usenix.org/osdi20

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

LISA19

October 28–30, 2019 | Portland, OR, USA
www.usenix.org/lisa19

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

Register by Monday, October 8, and save!