An Ounce of Prevention is Worth a Pound of Cure: Ahead-of-time Preparation for Safe High-level Container Interfaces

Ricardo Koller Dan Williams
IBM T. J. Watson Research Center

Abstract

Containers continue to gain traction in the cloud as lightweight alternatives to virtual machines (VMs). This is partially due to their use of host filesystem abstractions, which play a role in startup times, memory utilization, crash consistency, file sharing, host introspection, and image management. However, the filesystem interface is high-level and wide, presenting a large attack surface to the host. Emerging secure container efforts focus on lowering the level of abstraction of the interface to the host through deprivileged functionality recreation (e.g., VMs, userspace kernels). However, the filesystem abstraction is so important that some have resorted to directly exposing it from the host instead of suffering the resulting semantic gap. In this paper, we suggest that through careful *ahead-of-time metadata preparation*, secure containers can maintain a small attack surface while simultaneously alleviating the semantic gap.

1 Introduction

Filesystem abstractions like files and directories are universal and ubiquitous. They are fundamental to what containers are and how they are managed. Container workloads are stored as files (executables, configuration files, etc), and distributed as image layers which are simply file archives. Data sharing between containers is also defined at the file granularity: operators routinely specify how host files and directories should be shared between multiple containers.

Today's containers adopt an approach where the host implements the filesystem and exports it to container processes. This has many benefits, most of them due to having a single host filesystem and cache shared by multiple containers: (i) fast startup times due to efficient use of overlay filesystems (ii) efficient use of memory due to a single page cache, (iii) trivial container crash consistency as there is no filesystem state maintained in container memory, (iv) trivial file sharing and easy host introspection due to a single arbitrator for file accesses, and finally (v) straightforward image layer creation as the container filesystem is already layered at the same file-based granularity as images.

However, relying on the host to implement the filesystem abstraction is not without cost: the filesystem abstraction is high-level, wide, and complex, providing an ample attack surface to the host. Bugs in the filesystem implementation can be exploited by a user to crash or otherwise get control of the host, resulting in (at least) a complete isolation failure in a multi-tenant environment such as a container-based cloud. A search on the CVE database [13] for kernel filesystem vulnerabilities provides a sobering reminder of the width, complexity, and ultimately danger of filesystem interfaces.

Efforts to improve the security of containers use low-level interfaces to reduce the attack surface [42]. In general, the abstraction level of the interface between a container process and the host can be lowered by recreating host functionality in a *deprivileged* mode. For example, Kata containers [7] use virtualization to recreate host functionality in a (deprivileged) guest. Google's gVisor [6] implements host functionality as a (deprivileged) user space kernel process. Library OS approaches [38, 30, 22], including unikernels [31], have also been applied to processes [41] and containers [10], to recreate host functionality in a library linked to the (deprivileged) process.

Unfortunately, recreating the filesystem functionality in a deprivileged mode creates a *semantic gap* [19]: the host only sees underlying block accesses and any information about files is obscured. Almost all of the previously mentioned benefits from using the host filesystem are hampered by the semantic gap. These benefits are so core to the container ecosystem that, as a result, some secure container approaches have decided to forgo the deprivileged functionality recreation approach for the filesystem and adopt a *passthrough* approach instead [8]. Unfortunately this does nothing to prevent the exploitation of filesystem bugs.

In this paper, we make an observation that, through careful *ahead-of-time preparation* of filesystem metadata, high-level abstractions from the host can be safely utilized by the container even as the interface to host storage is restricted to a

low-level block interface. With such an approach, containers enjoy a small attack surface and encounter fewer issues due to the semantic gap. We sketch a potential implementation where the host prepares structures by directly exposing file content in read-only mode through a series of memory maps. It then creates filesystem metadata over the memory maps as a copy-on-write filesystem [14], such as a log-structured filesystem [37] (LFS). All metadata operations and all writes to the host filesystem are handled in a deprivileged filesystem persisted through the safer block interface.

2 Containers and the Filesystem

Filesystem abstractions comprise the units of abstraction that developers and system administrators work with and form a basis for the entire container ecosystem. For example, today's container images are stored and managed as sets of files and directories. To facilitate development and image management (e.g., to avoid duplication in storage and image distribution), they are made of *layers*, where each layer is itself a collection of files and directories. Layers are combined into a unified directory tree, called the container's *rootfs*, through the use of *union filesystems* [43]. Files and directories also form the basic unit of policy and sharing in today's container ecosystem. An operator can specify files or directories from the host to be shared with a container in one or more *volumes*.

While the concepts of files and directories have persisted over many years, there are design tradeoffs pertaining to which layer in the stack implements them. Today's container platforms adopt an approach in which the host is the one implementing the filesystem abstraction and directly exposing it to containers.[1] In the rest of this section, we discuss the pros and cons of such an approach.

2.1 Benefits

The container storage system derives many benefits from directly exposing the host filesystem to containers:

- **Fast startup.** Preparing an image for execution requires no format conversion or translation from the files and directories specified in the container image. Typically the only work the host filesystem must perform is to add a new writable filesystem layer, which is a trivial operation for a modern union filesystem like `overlayfs` [11]. The resultant fast startup is important for emerging domains like serverless computing [26, 28].

- **Memory sharing.** Filesystems in the host make use of the page cache in the kernel. If the same file is opened in multiple containers, the corresponding memory pages can be shared in copy-on-write mode.

- **Consistency during container crashes.** Filesystems can contain a fair amount of runtime state in caches or data structures. Fortunately, since the host—not the container—maintains filesystem state, a container crash does not result in a loss of this state.

- **File sharing and host introspection.** The host filesystem provides a single arbitrator for file access, managing the locking and other synchronization tasks required to safely enable concurrent filesystem access between multiple principals. Also, host files and directories conveniently match the granularity at which access control policies are typically specified. This is useful not just for sharing between containers, but also host introspection: for example, it is trivial for the host to examine the container filesystem to identify if any software packages in the container exhibit known vulnerabilities [4, 27, 2, 1].

- **Image layer creation.** A layer for a container image is a file-based diff from any initial filesystem state. Computing such a diff is trivial for a host filesystem like `overlayfs` where each layer is stored in a working directory. An image layer is simply an archive of that directory.

2.2 Security Issues

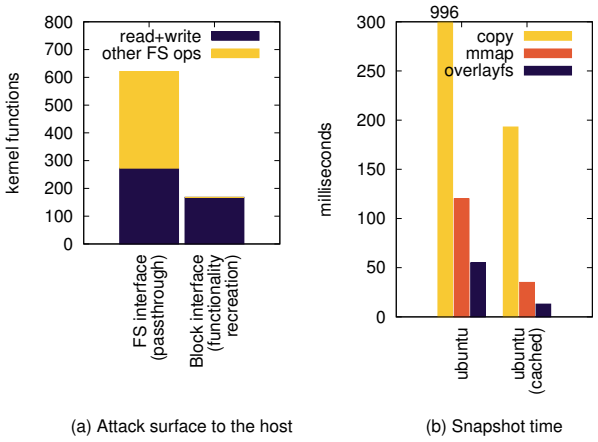
Containers have often been criticized for a perceived lack of security because their wide and high-level interface to the host provides an ample attack surface. The filesystem abstraction is no exception.

For a practical demonstration of the attack surface of the filesystem interface, we reproduced an exploit of a recently discovered bug in the `ext4` filesystem code: CVE-2018-10840. In this case, removing the extended attributes of a file on a crafted `ext4` image leads to a kernel oops due to a buffer overflow. The culprit syscall—part of the filesystem interface—that allows a userspace program to trigger this bug is `removexattr`, due to a missing check.

For a more abstract characterization of attack surface, as in prior work [42, 41], we measure the amount of unique kernel functions accessible through an interface with `ftrace` as a proxy for attack surface. A lower amount of accessible code suggests an implementation with lower complexity, and less code is generally believed to contain fewer bugs [32, 21]. Fewer bugs generally lead to fewer exploitable vulnerabilities, and a safer interface. We note that the density of bugs has been refined based on metrics like code age [34] and code popularity [30], but in general, the less code the better.

Figure 1(a) shows the number of unique kernel functions accessed when directly using the filesystem interface with

[1] The host hides some paths via mount namespacing [12].



(a) Attack surface to the host (b) Snapshot time

Figure 1: (a) The high-level filesystem interface exposes a significantly larger attack surface than the low-level block interface, evidenced by the number of kernel functions accessed when stressing the filesystem. (b) On-the-fly conversion of filesystem layers to a disk image suitable for low-level access increases startup time.

a filesystem stress test from the Linux Test Project [29]. The filesystem interface is wide: the test accessed over 600 unique kernel functions!

3 Secure Containers and the Semantic Gap

In response to security concerns, recent industry developments towards more secure containers have tried to lower the level of host complexity (and thus the number of bugs and vulnerabilities) beneath the interface to the host by *recreating* OS abstractions in a less privileged layer. Figure 2 shows three recreation approaches: a) virtualization, used by Kata containers [7], in which the guest kernel recreates OS abstractions; b) a userspace kernel process that implements kernel functionality, embodied today by Google's gVisor project; and c) a library OS approach [22, 35, 40], which has recently been applied in the context of containers as IBM's Nabla containers [10].

If a filesystem is recreated in the container, which we call a *deprivileged filesystem*, then the lower-level abstraction in the storage stack, namely the block layer, becomes the container/host interface. Such an interface consists of reads and writes of block sectors. As shown in Figure 1(a), the block interface presents a significantly smaller attack surface to the host, reducing the number of accessed kernel functions under the filesystem stress test from over 600 to about 150. However, any information about the file content or metadata the blocks contain is obscured. This phenomenon is known as the *semantic gap* [19].

In the container ecosystem, secure container approaches

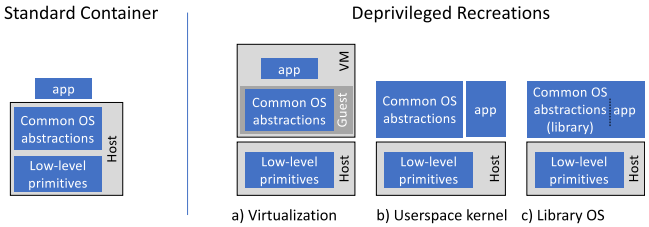


Figure 2: Secure containers employ deprivileged recreations of OS abstractions through (a) virtualization (e.g., Kata containers [7]), (b) a userspace kernel process (e.g., gVisor [6]), or (c) a library OS (e.g., Nabla containers [10]).

must adhere to the developer workflow popularized by containers, described in Section 2, as much as possible. In other words, container images and volumes are based on files and directories, yet for security reasons, deprivileged filesystems encourage the use of block-level interfaces instead. Current secure containers take one of two approaches: 1) *conversion* to a low-level interface suitable for a deprivileged filesystem and suffering the semantic gap, or 2) *passthrough*, potentially with guarding.

3.1 Conversion for a Deprivileged Filesystem

Some secure container approaches utilize layered filesystem images for containers, but convert them to a disk image suitable for a block interface (and deprivileged filesystem) immediately before executing the container. For example, the Nabla containers runtime [10] explicitly creates a formatted disk image file and exports it to the container.[2] However, the semantic gap erodes many of the benefits of using a host-implemented filesystem:

- **Fast startup.** Converting the high-level layered filesystem image to a low-level disk image incurs overhead akin to a file copy. For example, conversion could entail creation of a file (e.g., `dd`), metadata creation (e.g., `mkfs`), then copying filesystem contents to the image. Figure 1(b) shows the time for such a copy (labeled *copy*) for the files comprising the `ubuntu:16.04` container image with cold and warm host caches. Even in the warm case, *copy* takes over 193ms, compared to using the host's union filesystem (labeled *overlayfs*) which takes 13ms. Ironically, this cost is paid in order to *create* the semantic gap.

- **Memory sharing.** The host's page cache can no longer effectively share memory pages when identical files are in use by two containers on the system. Block accesses do not provide information about files because

[2] We also consider block-based layering in the host, like *devicemapper*, used by AWS Firecracker [3], to be a form of conversion, and such an approach encounters similar semantic gap issues.

of the semantic gap, so the system must resort to CPU-intensive content-analysis based approaches to memory sharing [9, 24].

- **Consistency during container crashes.** A container crash implies a loss of runtime state in a deprivileged filesystem. The deprivileged filesystem must be careful to write data in the right order to the underlying block layer to be able to reconstruct all state for consistency (e.g., a log or journal [37, 14]). Maintaining consistency in the face of filesystem crashes is complex, subtle and continues to be an active area of research [14, 33, 18].

- **File sharing and host introspection.** Due to the semantic gap, the host does not know how to arbitrate access to files and directories. Thus the filesystem structures inside the opaque disk image can only be safely mounted by one filesystem implementation (by either a deprivileged filesystem or the host) at a time.[3] Furthermore, file-based policies for access control, including sharing are obscured in an opaque disk image.

- **Image layer creation.** Only modified blocks, not files, are visible to the host. Creating a file-based diff is expensive: it entails traversing both the filesystem contained on the new set of blocks and the original filesystem and comparing the contents of all of the files.

3.2 Passthrough

In an attempt to avoid the semantic gap, several of the secure container approaches that otherwise lower their interface to the host through deprivileged recreation have chosen to directly expose the host filesystem. For example, Kata containers use the virtio 9P [8] protocol to pass filesystem system calls directly to the host. Unfortunately, this exposes the host to attack: we have successfully exploited CVE-2018-10840 from within a Kata container (on a host with a vulnerable kernel).

To mitigate the issue of a large attack surface, other approaches, like gVisor, also expose the host filesystem, but through a trusted guard. gVisor uses gofer as a proxy to the filesystem [6], which blocks the use of certain filesystem operations, like following symbolic links outside of the directory tree belonging to the container (e.g., to fix CVE-2017-1002101).

4 Ahead-of-time Preparation

We now describe an approach that falls between the two extremes of passthrough and deprivileged recreation. Our proposal, shown in Figure 3, is paradoxically both 1) high-level, largely avoiding the semantic gap, and 2) low-level, maintaining a low attack surface. In particular, the only direct access permitted to the host filesystem are reads of file content. All other filesytem operations are handled by the deprivileged filesystem, backed by a block device in the host.

[3]The exceptions are symmetric clustered based filesystems, like RGFS or PVFS2 [39].

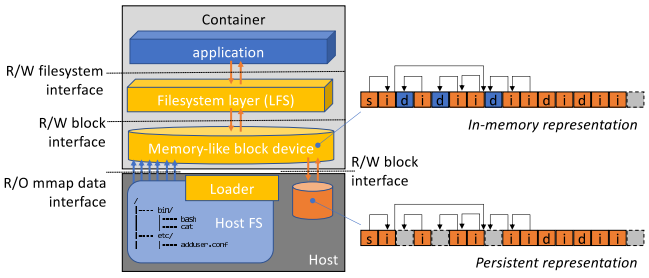


Figure 3: By preparing memory-mapped structures ahead of time, the storage interface can be both high-level enough to avoid the semantic gap and low level enough for a small attack surface.

The key insight is that prior to starting the container, in a safe, controlled fashion, the host can traverse the filesystem and prepare structures without adding to the runtime attack surface. We refer to this technique as *ahead-of-time* preparation, performed by a *loader* component (as shown in Figure 3). In the rest of this section, we sketch the basic design, revisit the filesystem properties discussed in the previous two sections to see where this design alleviates the semantic gap, then describe how it maintains a low attack surface.

4.1 Basic Idea

Figure 3 shows an overview of our proposal, which provides a filesystem abstraction on top of two underlying storage mechanisms: read-only data files from the host filesystem and a read-write block device. To integrate well with the existing container ecosystem, the deprivileged filesystem can be constructed on top of any (mounted) file system in the host. Internally, due to the multiplicity of underlying data sources, it is useful to refer to two related representations of the filesystem. The underlying block device contains the on-disk, *persistent* representation. The persistent representation contains filesystem metadata in its entirety, but is incomplete: holes exist for some data pages. The loader combines the incomplete persistent representation from the block device with memory-mapped files directly from the host filesystem to construct a complete, *in-memory* representation. The two representations are identical in layout and content, except the persistent representation contains holes where the in-memory representation contains data. Before executing the container, the loader exposes the in-memory representation to the deprivileged recreation as a direct-access [5] block device. When the deprivileged recre-

ation finally runs, it will use a standard filesystem implementation to interact with the data.

Importantly, the deprivileged filesystem should not attempt to write to the regions corresponding to the file data backed by the memory-mapped files. We believe the container can construct such a deprivileged filesystem if it takes inspiration from union filesystems [43] and filesystems with snapshot support [25, 16, 36]: basically any filesystem that provides the ability to have a read-only base layer below a writable layer. One such filesystem could be a modified log-structured filesystem (such as LFS [37]), where the first part of the log is used for read-only data and metadata, and the writable append-only part of the log handles all updates, persisted via the host through the block interface. The filesystem must not attempt to reuse the exposed (read-only) host pages for any purpose; for example, garbage collection in LFS must be modified to avoid collecting the read-only pages.

4.2 Alleviating the Semantic Gap

This design mitigates the semantic gap as follows:

- **Fast startup.** Unlike the conversion approaches described in Section 3.1, ahead-of-time preparation only requires reading filesystem metadata, not copying content. Figure 1(b) shows that it only takes about 35ms warm and 120ms cold to perform the ahead-of-time preparation (labeled *mmap*), a significant improvement over a full copy.
- **Memory sharing.** The host's use of memory maps effectively eliminates the semantic gap for memory sharing: multiple containers using the same base images and accessing the same files will share the same pages. It is important, however, that these pages do not get cached again in the deprivileged filesystem to avoid double caching inefficiencies [23]. For example, if the deprivileged filesystem is part of Linux (e.g., in a VM), its page cache can be avoided using filesystems that support direct access (DAX) [5], memory-like devices.
- **Container crash consistency.** The deprivileged filesystem will lose state on a container crash, even though the read-only base layer will be kept consistent by the host. Log-structured filesystems guarantee crash consistency as no data is written in place. However, most log-structured filesystems only guarantee consistency for sector-addressable storage. Depending on how the in-memory representation is constructed, especially to take advantage of direct-access memory-like block devices, the block device may present byte-addressable storage when forming the in-memory representation. Fortunately, filesystems built for non-volatile memory [20, 44] extend log-structured concepts to byte-addressable environments.
- **File sharing and host introspection.** Sharing a container volume requires simultaneously recreating the in-memory representation on multiple containers and/or the host. In our proposal, accessing unmodified files in a read-only manner from an initial filesystem state is trivial due to the direct access for reads to the host filesystem. Many common container patterns, such as sidecars or adapters [17], share volumes with a single writer and multiple readers. In this case, each reader must interpret the persistent representation of the volume, which contains the writes in a filesystem log. As described above with respect to crash consistency, a log-structured filesystem guarantees that any such reader will read a consistent view of the filesystem log state. However, in order to ensure up-to-date contents, once again, no caching should take place in the deprivileged filesystem (e.g., via DAX [5]). If two or more containers are concurrently writing into the same log, more heavyweight synchronization mechanisms (i.e., like locking in clustered file systems) are required.
- **Layer creation.** Diffs between layers contain the full content of all the updated files and directories, but in the case of LFS, the semantic gap remains because the log is based on blocks. In order to create the list of modified files, the LFS log must be replayed, mounted, and then walked from the root comparing against the base layer.

4.3 Attack surface

The attack surface available to a container at runtime is reduced to the read path, accessible through read-only memory-mapped pages and block-level read/write for the log. To estimate the attack surface, we partition the filesystem access measurement for the number of unique kernel functions accessed with a filesystem stress test into two sets: 1) reads and writes, that we would expect to remain with ahead-of-time preparation, and 2) all other filesystem operations. Figure 1 shows the result: a larger attack surface (273 functions) than a strictly block-level interface (168 functions), but significantly smaller than full filesystem access (621 functions).

5 Summary

The semantic gap indicates that a safe, low-level block storage interface is not sufficient for containers, which have flourished by directly using the useful but insecure filesystem of the host. As we think about how to improve the security of containers, ahead-of-time preparation could help strike a balance between high-level functionality and low-level attack surface reduction.

6 Discussion Topics

Attack surface: runtime vs. pre-/post-runtime. In this paper we focused on *runtime* attacks: for example, exploiting a vulnerability through a carefully crafted system call argument. There are also *pre-runtime* and *post-runtime* attacks: for example, exploiting a vulnerability in setup code through a carefully crafted image or in teardown code when flattening a filesystem back into an image. Does ahead-of-time preparation reduce the runtime attack surface at the cost of increasing the pre- and post-runtime attack surface? Is one more dangerous than the other? We believe, through image provenance and ensuring that the host never interprets untrusted filesystem logs, pre- and post-runtimes attacks may be mitigated.

Interfaces beyond filesystems and block devices. We have been focusing on two interfaces: a high-level filesystem and a low-level block device interface, but there may be other interfaces that achieve a low attack surface with fewer semantic gap issues. For example, a key-value store interface may strike a middle ground. Do we, as a community, have sufficient measurement methodology to evaluate both the attack surface and semantic gap for various interfaces?

The deprivileged filesystem. We proposed leveraging LFS as a layering mechanism atop the ahead-of-time metadata preparation in order to manage filesystem updates. LFS uses copy-on-write (COW) at the block granularity [14], but many other options exist. By doing COW at the block granularity we increase memory sharing by not breaking the sharing whenever a file is updated even by a byte. However, it complicates the creation of image layers which have a different granularity (files instead of blocks). Instead, we could consider using copy-on-write at the file granularity by using a union filesystem [43] as the deprivileged filesystem, or we could use operation logs [15] which could simplify file sharing across containers. Is a log-structured filesystem the best choice?

Container image layer representations. Many of the semantic gap tradeoffs we identified, including startup time and layer creation, as well as other characteristics of the container system, like image size, stem from the fact that the layer representations are based on the file granularity. Using a different granularity of copy-on-write or even operation logs[4] for the image layers would change the size of the semantic gap. Is there another format that is not tied to a filesystem implementation (e.g., a least common denominator format) that could work within the container ecosystem?

[4]Image layering already uses some form of operation logs: files deleted in a layer are marked as deleted by creating an empty file with the same name (slightly modified to differentiate from an actual empty file with the same name).

References

[1] Anchore: Automated Container Security and Compliance for the Enterprise. `https://anchore.com/`.

[2] Aqua microscanner. `https://github.com/aquasecurity/microscanner`.

[3] AWS Firecracker. `https://firecracker-microvm.github.io/`.

[4] Clair: Automatic container vulnerability and security scanning for appc and Docker . `https://coreos.com/clair/docs/latest/`.

[5] DAX - Direct Access for files. `https://www.kernel.org/doc/Documentation/filesystems/dax.txt`. (Accessed on 2019-05-09).

[6] gvisor - Container Runtime Sandbox. `https://github.com/google/gvisor`. (Accessed on 2018-08-28).

[7] Kata Containers. `https://katacontainers.io/`.

[8] Kata Containers Architecture: Storage. `https://github.com/kata-containers/documentation/blob/master/design/architecture.md\#storage`.

[9] Kernel Samepage Merging. `https://www.linux-kvm.org/page/KSM`. (Accessed on 2018-08-28).

[10] Nabla Containers. `https://nabla-containers.github.io/`.

[11] Overlay Filesystem. `https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt`.

[12] `namespaces` Linux man page.

[13] Common Vulnerabilities and Exposures (CVE). `https://cve.mitre.org/`, 2015.

[14] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[15] Srivatsa S Bhat, Rasha Eqbal, Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. Scaling a file system to many cores using an operation log. In *Proc. of ACM SOSP*, Shanghai, China, October 2017.

[16] Jeff Bonwick and Bill Moore. Zfs: The last word in file systems, 2007.

[17] Brendan Burns and David Oppenheimer. Design patterns for container-based distributed systems. In *Proc. of USENIX HotCloud*, Denver, CO, June 2016.

[18] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proc. of ACM SOSP*, Shanghai, China, October 2017.

[19] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proc. of USENIX HotOS*, Elmau/Oberbayern, Germany, May 2001.

[20] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proc. of ACM SOSP*, Big Sky, MT, October 2009.

[21] Nigel Edwards and Liqun Chen. An historical examination of open source releases and their vulnerabilities. In *Proc. of ACM CCS*, Raleigh, NC, October 2012.

[22] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. of ACM SOSP*, Copper Mountain, CO, December 1995.

[23] Brendan Gregg. *Systems performance: enterprise and the cloud*. Prentice Hall, 1 edition, 2014.

[24] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *Proc. of USENIX OSDI*, San Diego, CA, December 2008.

[25] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proc. of Winter USENIX Conference*, San Francisco, California, January 1994.

[26] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A Berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, University of California, Berkeley, Feb 2019.

[27] Ricardo Koller, Canturk Isci, Sahil Suneja, and Eyal de Lara. Unified monitoring and analytics in the cloud. In *Proc. of USENIX HotCloud*, Santa Clara, CA, July 2015.

[28] Ricardo Koller and Dan Williams. Will serverless end the dominance of Linux in the cloud? In *Proc. of ACM/SIGOPS HotOS*, Whistler, BC, Canada, May 2017.

[29] Paul Larson. Testing Linux with the Linux test project. In *Proc. of Ottawa Linux Symposium*, Ottawa, Canada, June 2002.

[30] Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. Lock-in-Pop: Securing privileged operating system kernels by keeping on the beaten path. In *Proc. of USENIX Annual Technical Conf.*, Santa Clara, CA, July 2017.

[31] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proc. of ACM ASPLOS*, Houston, TX, March 2013.

[32] Steve McConnell. *Code complete*. Pearson Education, 2004.

[33] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proc. of USENIX OSDI*, Carlsbad, CA, October 2018.

[34] Andy Ozment and Stuart E. Schechter. Milk or wine: Does software security improve with age? In *Proc. of USENIX Security Symposium*, Vancouver, B.C., Canada, July 2006.

[35] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. Rethinking the library OS from the top down. In *Proc. of ACM ASPLOS*, Newport Beach, CA, March 2011.

[36] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.

[37] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[38] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. Ebbrt: a framework for building per-application library operating systems. In *Proc. of USENIX OSDI*, Savannah, GA, November 2016.

[39] T. D. Thanh, S. Mohan, E. Choi, S. Kim, and P. Kim. A taxonomy and survey on distributed file systems. In *2008 Fourth International Conference on Networked*

Computing and Advanced Information Management, volume 1, pages 144–149, Sep. 2008.

[40] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. Cooperation and security isolation of library OSes for multi-process applications. In *Proc. of ACM EuroSys*, Amsterdam, The Netherlands, April 2014.

[41] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. Unikernels as processes. In *Proc. of ACM SoCC*, Carlsbad, CA, October 2018.

[42] Dan Williams, Ricardo Koller, and Brandon Lum. Say goodbye to virtualization for a safer cloud. In *Proc. of USENIX HotCloud*, Boston, MA, July 2018.

[43] Charles P Wright and Erez Zadok. Kernel korner: unionfs: bringing filesystems together. *Linux Journal*, 2004(128):8, 2004.

[44] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proc. of USENIX FAST*, Santa Clara, CA, February 2016.