The True Cost of Containing: A gVisor Case Study

Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
University of Wisconsin, Madison

Abstract

We analyze many facets of the performance of gVisor, a new security-oriented container engine that integrates with Docker and backs Google's serverless platform. We explore the effect gVisor's in-Sentry network stack has on network throughput as well as the overheads of performing file opens via gVisor's Gofer service. We further analyze gVisor startup performance, memory efficiency, and system-call overheads. Our findings have implications for the future design of similar hypervisor-based container engines.

1 Introduction

OS-level virtualization, called "containerization" by some, has emerged as a popular alternative to virtual machines as new virtualization mechanisms have become available in Linux (*e.g.*, namespaces and cgroups). Tools such as Docker [18] have made it easy for novices to deploy their applications using these new mechanisms.

Whereas each application running in a virtual machine runs on its own guest operating system, OS-level virtualization allows multiple tenants to efficiently share a common OS. Although efficient, such sharing is also a security concern; as Daniel Walsh quipped, "containers do not contain" [24]. If the OS itself does not run in the container, OS resources with incomplete virtualization are vulnerable [24] and kernel bugs can be exploited through a large attack surface (over 300 system calls) [4].

The growing popularity of OS-level virtualization has spurred new efforts to make hypervisors as fast and as convenient as Docker containers [2, 15, 13, 16]; for example, Kata Containers (a hypervisor-based approach) are advertised as providing the "the speed of containers, the security of VMs." Other hypervisor-based sandboxes are being rebranded as containers as well [11, 23]. In the past, VMs and containers were easy to differentiate; in 2014, for example, Merkel wrote that "to put it simply, containers virtualize at the operating system level, whereas hypervisor-based solutions virtualize at the hardware level" [18]. "Extreme paravirtualization" [15] has blurred this distinction, and the term "container" is now frequently being used to describe any virtualization platform (hypervisor-based or otherwise) that is faster or more convenient than a traditional VM.

These new hypervisor-based container platforms may indeed offer the "security of VMs" [13], but unless they also provide "the speed of containers", or something close to it, few practitioners are likely to abandon traditional containers. Hence, an important question to ask is *what is the true performance cost of using these new security-oriented container platforms?* And *how might those costs be reduced in future implementations without sacrificing security?* In order to focus these broad questions, we embark on an analysis case study exploring the performance of gVisor, a new hypervisor-based container engine that integrates with Docker and backs Google's serverless platform [17].

gVisor is designed so that an attacker may not gain host access by only compromising a single subsystem. Of course, defense-in-depth entails layering, with new potential overheads, which we explore. gVisor also attempts to minimize the attack surface (*i.e.*, unfiltered system calls) between gVisor and the host. Removing commonly-exploited system calls such as `socket` and `open` from this interface requires the primary gVisor OS (the Sentry) to depend on a separate helper process (the Gofer) for opening files; we measure how this strategic splitting of the container kernel affects I/O performance.

Our findings shed light on many facets of gVisor performance, with implications for the future design of security-oriented containers. For example, invoking simple system calls is at least 2.2× slower on gVisor compared to traditional containers; worse, opening and closing files on an external tmpfs is 216× slower. I/O is also costly; reading small files is 11× slower, and downloading large files is 2.8× slower. These resource overheads significantly affect high-level operations (*e.g.*, Python module imports), but we show that strategically using the Sentry's built-in OS subsystems (*e.g.*, the in-Sentry tmpfs) halves the import latency of many modules.

The rest of this paper is organized as follows: we provide a gVisor background (§2), analyze its performance (§3), describe related work (§4), and conclude (§5).

2 Background: gVisor Containers

The gVisor container is designed to be a building block in multiple environments. In addition to serving as the isolation mechanism for GCF (Google Cloud Functions) [17], gVisor implements the OCI (Open Container

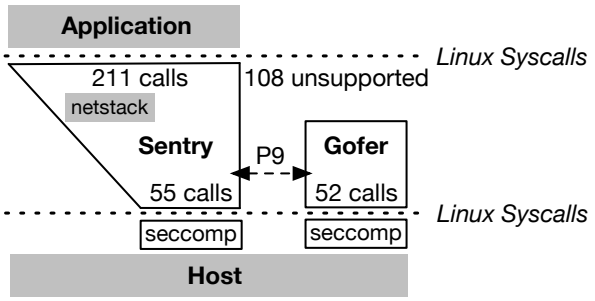

Figure 1: **gVisor Architecture.**

Initiative) [20] standard, used by Docker [18]. Thus, Docker users can readily switch between the default engine, *runc* (based on Linux namespaces and cgroups), and the gVisor runtime engine, named *runsc*.

Architecture: Figure 1 illustrates the gVisor architecture. The application runs on the Sentry, which both implements Linux and itself runs on Linux. This Linux-on-Linux architecture provides defense in depth. A malicious application that exploits a bug in the Sentry only gains access to a restricted user-space process; in contrast, a process escaping a Docker runc container might gain root access. The Sentry has two modes for handling system calls. In ptrace mode, the Sentry intercepts system calls by acting much like a debugger attached to the application. In KVM mode (more common), the Sentry executes as a guest OS in a virtual machine.

System Calls: The Sentry exposes 211 of 319 Linux system calls to the application. The Sentry uses just 55 unique host calls to implement those 211 calls. This narrowing provides a security advantage: `seccomp` filters prevent a compromised Sentry from using system calls beyond the 55 it is expected to use. The gVisor engineers identified `open` and `socket` as common attack vectors [4], so these calls are not among the whitelisted 55; gVisor's storage and networking stacks were carefully designed to avoid needing these calls.

Storage: There are three main patterns for how the Sentry handles file-related system calls. First, gVisor implements several file systems internally (*e.g.*, a tmpfs, procfs, and overlayfs); the Sentry can generally serve I/O to these internal file systems without calling out to the host or other helpers. Second, the Sentry services `open` calls to external files (as permitted) with the help of a separate process called the Gofer; the Gofer is able to open files on the Sentry's behalf and pass them back via a 9P (Plan 9) channel. Third, after the Sentry has a handle to an external file, it can serve read and write system calls from the application by issuing similar system calls to the host. If gVisor is running in KVM mode, the Sentry must switch from GR0 (Guest Ring 0) to HR3 (Host Ring 3) to process such requests, as system calls to the host cannot be invoked directly from GR0.

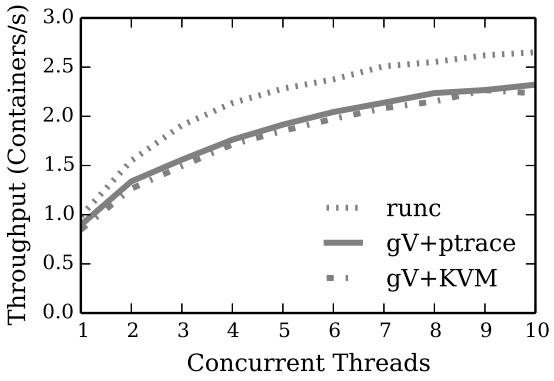

Figure 2: **Container Init/Teardown Performance.** *Varying numbers of threads (x-axis) create and destroy containers concurrently. The total system throughput is reported as containers-per-second on the y-axis.*

Networking: the network stack (called netstack) is implemented in the Sentry in Go; netstack directly communicates via a raw socket with a virtual network device, typically initialized by Docker [7, 9].

Memory: A traditional OS running on a virtual machine assumes a view of "physical" memory, from which it allocates memory for processes. In contrast, the Sentry is built to run on Linux and implements process-level `mmap` calls by itself invoking `mmap` on the host [10].

3 gVisor Performance Analysis

We compare the performance of runsc (the gVisor Docker runtime) to that of runc and native on a CloudLab 220 M4 machine [3] having two 8-core 2.4 GHz Intel E5-2630 CPUs, 128 GB of RAM, a 10Gb NIC, and 480 GB SSD. We use Ubuntu 16.04 (4.4.0 kernel) and Docker 18.09.3. We explore the following questions: *How quickly can gVisor start and stop containers* (§3.1)? *What are the overheads associated with servicing system calls* (§3.2)? *How do page faults affect an application's malloc performance* (§3.3)? *How does gVisor's user-space, Go-based network stack perform* (§3.4)? *What are the overheads related to file access* (§3.5)? And finally, *how are Python module imports affected by gVisor's system call and I/O overheads* (§3.6)?

3.1 Container Lifecycle

In this experiment, we explore overall container lifecycle performance by creating and destroying containers as fast as possible. We execute long enough to achieve steady state since prior work [5] shows container destruction has significant asynchronous overheads. With no concurrency, the total setup and teardown times are 1.014s for runc, 1.117s for gVisor+KVM, and 1.181s for gVisor+ptrace. Figure 2 shows the result when multiple containers are created and destroyed concurrently by

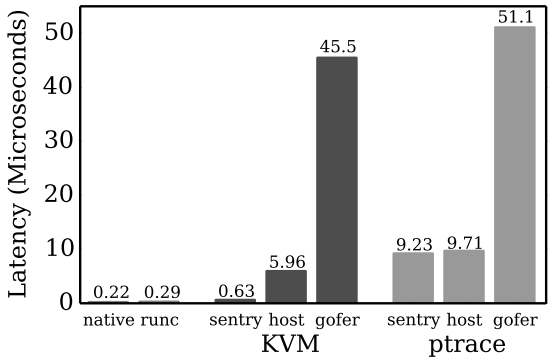

Figure 3: **System Call Overhead.** *The bars show the average latency for* `gettimeofday` *across 100M executions.*

a varying number of threads. All systems scale poorly; runc achieves at most 2.6 containers/second, and runsc achieves 2.3 containers per second.

Implications: Although gVisor is designed for high container density per machine, our experiments show runc will be slightly more performant for high-churn workloads. With respect to startup, gVisor's KVM and ptrace modes are comparable.

3.2 System Calls

Depending on the system call, the Sentry may handle it internally, invoke a call on the host, or get help from the Gofer. In order to compare the overheads across these scenarios, we implemented three versions of the same system call (`gettimeofday`) inside gVisor, and compared the performance to the same call in runc and native.

Figure 3 shows the average latency for native, runc, and gVisor. For gVisor, we show every combination of runtime mode (KVM or ptrace) and implementation pattern (Sentry only, invoke on host, and get help from Gofer). Whereas runc is only 32% slower than native, the fastest gVisor result (Sentry-only on KVM) is 2.8× slower. In KVM mode, calling to the host is 9× slower than operating just within the Sentry, and calling to the Gofer is 72× slower. ptrace latencies are 42-232× slower than native latencies.

Implications: Our results confirm previous observations that workloads heavy on system calls may "have some pain" on gVisor [26]. System calls that cannot be handled solely by the Sentry (i.e. opening or reading a file on the host) may have degraded performance. System-call heavy workloads should avoid ptrace mode.

3.3 Memory

gVisor is designed to have a memory footprint of just 15 MB [4]. Of course, an application that grows its memory footprint beyond the originally allocated physical memory triggers page faults, a process in which the Sentry must sometimes be involved [10].

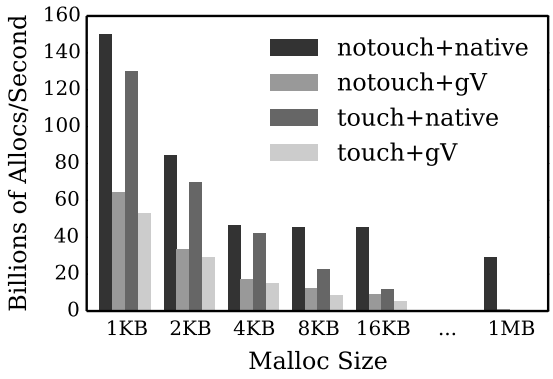

Figure 4: **Malloc Performance.** *Results are shown for both native and gVisor experiments, with and without touching the allocated memory.*

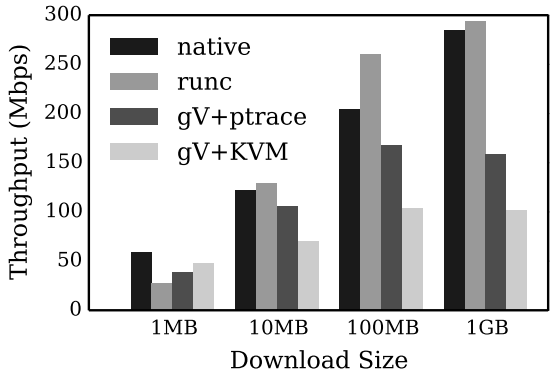

Figure 5: **Network Throughput.** *Throughput is measured by downloading files of varying sizes (x-axis) with* `wget`.

We stress the memory system with a benchmark that performs 100K allocations of varying sizes using `malloc`. We experiment with both accessing all the pages of allocated memory or not accessing them. The benchmark never frees any allocations. Figure 4 shows gVisor achieves 40% the allocation rates of the native experiment. For large allocations (*e.g.*, 1 MB), the native experiment achieves 29.3 billion allocations per second as long as the allocated memory is not touched. In contrast, gVisor's rate approaches zero, suggesting gVisor's allocation does work more eagerly than the native setup.

Implications: Memory-heavy applications can be expected to experience significant slowdowns on gVisor. It is worth investigating whether allocating and mapping memory in larger batches could decrease the Sentry's involvement and provide performance closer to native.

3.4 Network

The `socket` system call was identified as a commonly exploited attack vector by the gVisor team [4]; gVisor guards against such attacks by relying on a user-space networking stack, called netstack [19].

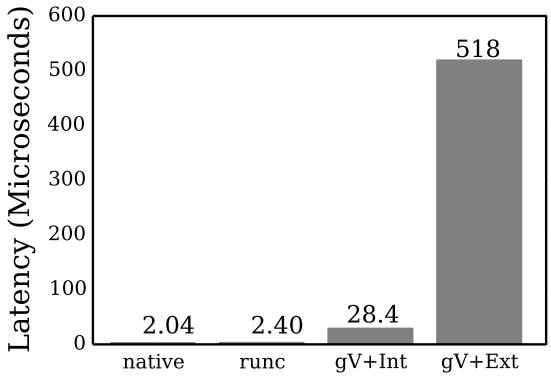

Figure 6: **Open and Close Latency.** *Results are an average over 100K consecutive accesses to the same file on native, runc, internal tmpfs (gV+Int), and external tmpfs (gV+Ext).*

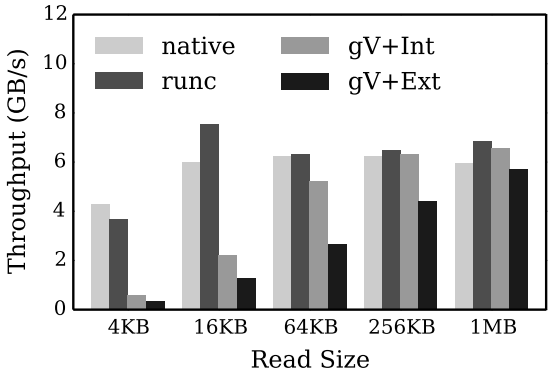

Figure 7: **Read Throughput (tmpfs).** *The x-axis shows the read size. The right two bars in each group are for gVisor.*

We evaluate the network performance by downloading files of varying sizes (from 1 MB to 1 GB) from a speedtest site using `wget`; Figure 5 shows the result: for the 1 GB file, gVisor's KVM platform achieves about 34% of the throughput achieved by the other systems. gVisor's ptrace platform performs slightly better (it is unclear why), yet is still only 54% as fast as the other systems. For small downloads (*e.g.*, 1 MB), gVisor throughput is comparable to other systems.

Implications: gVisor's netstack handles small downloads well, but scales poorly. However, the slowdowns we observed were not as severe as those previously reported [26], agreeing with the assessment in gVisor's documentation that the "network stack is improving quickly." [8]. It will be worthwhile to revisit these measurements as the implementation matures.

3.5 Storage

We now measure gVisor's file performance, both on the Sentry's internal tmpfs, and on an external tmpfs on the host, accessed via the Gofer. To start, we measure the latency of opening and closing a file without performing any I/O requests. As Figure 6 shows, opening and closing a file on an external tmpfs is 216× slower than making the same accesses from a runc container, suggesting routing through the Gofer is a major bottleneck. Accessing a file in the Sentry's internal tmpfs is faster than the external accesses, but is still 12× slower than runc.

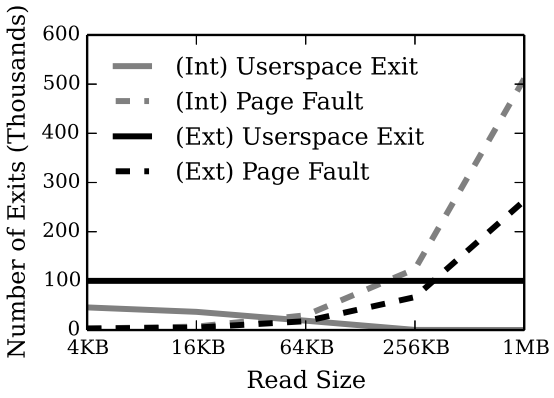

Figure 8: **KVM Exits for Read Workload.** *The x-axis shows the read size. The solid line represents the number of user-space exits performed by the KVM. The dotted line represents the number of exits by the KVM due to page faults.*

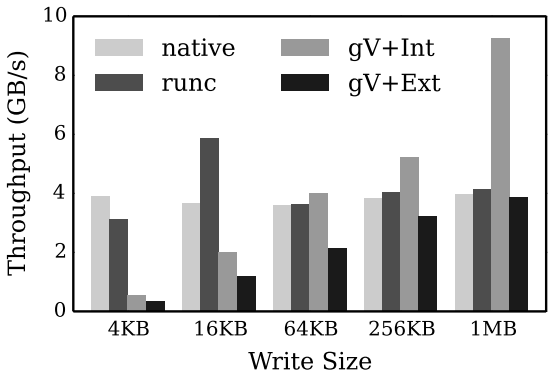

Figure 9: **Write Throughput (tmpfs).** *The x-axis shows the write size.*

We measure read throughput by performing 100K sequential reads (of varying size) to a large in-memory file. Figure 7 shows that gVisor performs poorly for small requests, but access to the internal tmpfs becomes competitive for request sizes around 64 KB and throughput to an external tmpfs becomes competitive around 1 MB.

For the two gVisor bars shown in Figure 7 (gV+Int and gV+Ext), we investigate further by tracing KVM exit events. We observed two common types of KVM exits: user-space exits (which occur when the Sentry switches rings to invoke host system calls) and page faults. Figure 8 shows the frequency of these two events for the two workloads. User-space exits approach zero for internal accesses as the request size increases; user-space exits are constant for the external workload, though more bytes are read per exit at larger sizes. Page faults increase

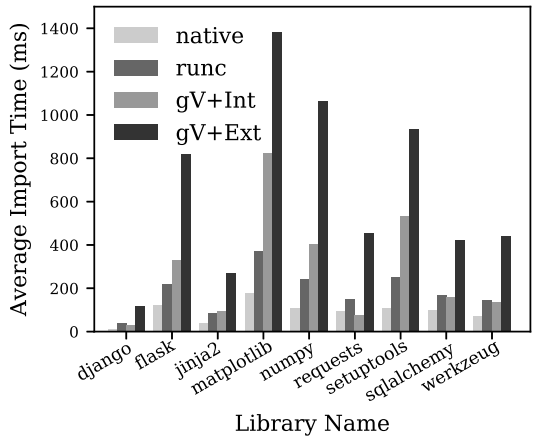

Figure 10: **Python Import Latency.** *We measure the time to import various modules (along x-axis) without containers, with runc, and with gVisor. With containers, we show performance with a host tmpfs volume mount (both runc and gVisor) and Sentry tmpfs mount.*

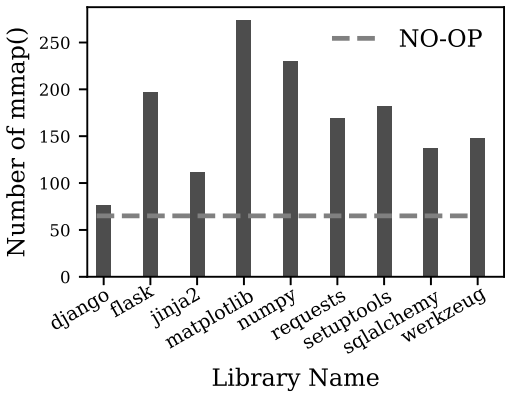

Figure 11: **Number of mmap.** *The dashed line indicates mmap calls during Python startup with no imports (the portion of bars above the line indicates module-specific mmaps).*

in both cases for larger requests.

Figure 9 repeats the Figure 7 experiments, but for writes. For large requests, access to the internal tmpfs actually outperforms runc. KVM exit patterns were similar for the internal write and read experiments (not shown). For external writes, user-space exits were still constant around 100K, though there were almost no page faults.

Implications: gVisor I/O involves significant switching between modes and processes, and the resulting throughput suffers for opens and small I/O requests. Our results show that an internal tmpfs significantly improves performance; for stateless workloads (*e.g.*, Google Cloud Functions), this appears a good option.

3.6 Python

Google's latest Python runtime for its serverless platform is based on gVisor [17], and importing Python modules is a significant cold-start bottleneck for serverless applications [5]. We now explore import costs in gVisor.

We measure module import latency for nine popular Python modules using each platform, showing the results in Figure 10. We observe that imports on gVisor are usually 2-4× slower than for runc when gVisor is using an external tmpfs; when using its internal tmpfs, gVisor sometimes outperforms runc.

Given gVisor's known issues with I/O performance, we measure the number of mmap calls made during each import experiment, showing the results in Figure 11. We observe that the four modules causing the most mmaps (flask, matplotlib, numpy, and setuptools) were also the slowest for importing modules on gVisor+Ext.

Implications: We have shown that importing Python modules from the Sentry's prepopulated internal tmpfs is 40% to 70% faster than fetching modules from an external tmpfs. This suggests that Python workloads will benefit from mechanisms that make application resources available inside a container through some means other than the Gofer and the regular I/O stack.

4 Related Work

Other lightweight hypervisor-based container systems have also been recently implemented using KVM, including Kata Containers [12, 26] and Amazon's Firecracker [2, 27], which backs AWS Lambda. LightVM is a re-design of Xen aimed at booting unikernels inside lightweight VMs quickly [16]. The funnel-shaped architecture of gVisor's Sentry resembles that of Drawbridge [21]. Prior to gVisor's netstack, the GoNet project implemented a user-space network stack in Go [22].

Many prior measurement studies have explored container performance in different scenarios [1, 14, 25]. Felter *et al.* [6] did a recent comparison of containers and virtual machines, and a talk by Wang [26] highlighted some of the performance tradeoffs between gVisor and Kata containers.

5 Conclusion

gVisor is arguably more secure than runc, as a compromised Sentry only gives an attacker access to a user-space process severely limited by seccomp filters, whereas a compromised Linux namespace or cgroup may give an attacker access to the host kernel. Unfortunately, our analysis shows that the true costs of effectively containing are high: system calls are 2.2× slower, memory allocations are 2.5× slower, large downloads are 2.8× slower, and file opens are 216× slower. We believe that bringing attention to these performance and scalability issues is the first step to building future container systems that are both fast and secure.

6 Acknowledgments

Feedback from the anonymous reviewers has significantly improved this work. We also thank the members of the ADSL research group for their helpful suggestions and comments at various stages. This material was supported by NSF/VMware ECDI grant CNS 1838733. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or other institutions.

7 References

[1] Marcelo Amaral, Jorda Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder. Performance Evaluation of Microservices Architectures Using Containers. In *2015 IEEE 14th International Symposium on Network Computing and Applications*, pages 27–34, Boston, MA, 2015.

[2] Jeff Barr. Firecracker Lightweight Virtualization for Serverless Computing. https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/, November 2018.

[3] CloudLab. https://www.cloudlab.us/, February 2018.

[4] Dawn Chen and Zhengyu He. Container Isolation at Scale (Introducing gVisor). https://schd.ws/hosted_files/kccnceu18/47/Container%20Isolation%20at%20Scale.pdf, 2018.

[5] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, 2018. USENIX Association.

[6] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015.

[7] gVisor Documentation - Networking. https://gvisor.dev/docs/user_guide/networking, May 2019.

[8] gVisor Documentation - Performance Guide. https://gvisor.dev/docs/architecture_guide/performance, May 2019.

[9] gVisor: network.go. https://github.com/google/gvisor/blob/master/runsc/sandbox/network.go, May 2019.

[10] Sentry Memory README.md. https://github.com/google/gvisor/tree/master/pkg/sentry/mm, May 2019.

[11] Hyper-V Isolation Documentation. https://docs.microsoft.com/en-us/virtualization/windowscontainers/manage-containers/hyperv-container, September 2018.

[12] Kata Design Document Github. https://github.com/kata-containers/documentation, May 2018.

[13] The Speed of Containers, the Security of VMs. https://katacontainers.io/, March 2019.

[14] Zhanibek Kozhirbayev and Richard O. Sinnott. A Performance Comparison of Container-based Technologies for the Cloud. *Future Generation Computer Systems*, 68, 2017.

[15] Nicolas Lacasse. Open-Sourcing gVisor, a Sandboxed Container Runtime. https://cloud.google.com/blog/products/gcp/open-sourcing-gvisor-a-sandboxed-container-runtime, May 2018.

[16] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than your Container. *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[17] Eyal Manor. Bringing the best of serverless to you. https://cloudplatform.googleblog.com/2018/07/bringing-the-best-of-serverless-to-you.html, July 2018.

[18] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014(239), 2014.

[19] IPv4 and IPv6 Userland Network Stack. https://github.com/google/netstack, May 2019.

[20] Open Container Initiative. https://www.opencontainers.org/, March 2019.

[21] Don E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the Library OS from the Top Down. *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[22] Harshal Sheth and Aashish Welling. An Implementation and Analysis of a Kernel Network Stack in Go with the CSP Style. *CoRR*, abs/1603.05636, 2016.

[23] vSphere Integrated Containers. https://www.vmware.com/products/vsphere/integrated-containers.html, March 2019.

[24] Daniel J. Walsh. Are Docker Containers Really Secure? https://opensource.com/business/14/7/docker-security-selinux, July 2014.

[25] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 132–145, Boston, MA, 2018. USENIX Association.

[26] Xu Wang. Kata Containers and gVisor: a Quantitative Comparison. https://www.openstack.org/summit/berlin-2018/summit-schedule/events/22097/kata-containers-and-gvisor-a-quantitative-comparison, November 2018.

[27] Radu Weiss, Noah Meyerhans, James Turnbull, and Alexandra Iordache. Firecracker Design Document Github. https://github.com/firecracker-microvm/firecracker/blob/master/docs/design.md, May 2019.