Blending Containers and Virtual Machines: A Study of Firecracker and gVisor

Anjali
University of Wisconsin-Madison
anjali@wisc.edu

Tyler Caraza-Harter
University of Wisconsin-Madison
tharter@wisc.edu

Michael M. Swift
University of Wisconsin-Madison
swift@cs.wisc.edu

Abstract

With serverless computing, providers deploy application code and manage resource allocation dynamically, eliminating infrastructure management from application development.

Serverless providers have a variety of virtualization platforms to choose from for isolating functions, ranging from native Linux processes to Linux containers to lightweight isolation platforms, such as Google gVisor [7] and AWS Firecracker [5]. These platforms form a spectrum as they move functionality out of the host kernel and into an isolated guest environment. For example, gVisor handles many system calls in a user-mode Sentry process while Firecracker runs a full guest operating system in each microVM. A common theme across these platforms are the twin goals of strong isolation and high performance.

In this paper, we perform a comparative study of Linux containers (LXC), gVisor secure containers, and Firecracker microVMs to understand how they use Linux kernel services differently: *how much does their use of host kernel functionality vary*? We also evaluate the performance costs of the designs with a series of microbenchmarks targeting different kernel subsystems.

Our results show that despite moving much functionality out of the kernel, both Firecracker and gVisor execute substantially more kernel code than native Linux. gVisor and Linux containers execute substantially the same code, although with different frequency.

CCS Concepts • **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → **Operating systems**; **Cloud computing**; • **Security and privacy** → **Virtualization and security**.

VEE '20, March 17, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7554-2/20/03…$15.00
https://doi.org/10.1145/3381052.3381315

Keywords serverless computing, virtualization, operating systems, code coverage, benchmarking, firecracker, gvisor

ACM Reference Format:
Anjali, Tyler Caraza-Harter, and Michael M. Swift. 2020. Blending Containers and Virtual Machines: A Study of Firecracker and gVisor. In *16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20), March 17, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3381052.3381315

1 Introduction

Serverless computing is a new execution model in which tenants provide lightweight *functions* to a platform provider, which is responsible for finding a host and launching them on that host. As public clouds are multi-tenant, serverless computing naturally seeks to run functions from multiple tenants on the same physical machine, which requires strong isolation between tenants. For example, AWS ran serverless functions in Linux containers inside virtual machines, with each virtual machine dedicated to functions from a single tenant [36]. In contrast, Azure multiplexed functions from multiple tenants on a single OS kernel in separate containers [36]. As a result, a kernel bug could compromise inter-tenant security on Azure but not AWS.

Recently, lightweight isolation platforms have been introduced as a bridge between containers and full system virtualization. Amazon Web Services (AWS) Firecracker [4] and Google's gVisor [7] both seek to provide additional defense and isolation in their respective serverless environments. Both are based on the principles of least privilege and privilege separation. Firecracker takes advantage of the security and workload isolation provided by traditional VMs but with much less overhead via lightweight I/O services and a stripped-down guest operating system. gVisor, in contrast, implements a new user-space kernel in Go to provide an additional layer of isolation between containerized applications and the host operating system.

In order to make sense of these new platforms and how they compare with containers and heavyweight system virtual machines, we can organize the platforms on a spectrum, illustrated in Figure 1. The position illustrates how much of their functionality resides in the host operating system. At the left, the host operating system provides all functionality. With Linux containers (LXC), a user-mode daemon plays an important role in provisioning and configuring containers.

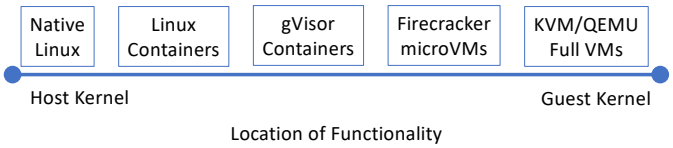


Figure 1. Spectrum of OS functionality location in isolation platforms.

gVisor sits in the middle, as it moves execution of most system calls out of the kernel into a user-mode *Sentry* process. Firecracker runs a lightweight virtual machine (*microVM*) with a complete guest kernel, but relies on the host operating system to securely isolate the Firecracker hosting process. Full virtualization, such as QEMU/KVM, moves most functionality out of the host kernel into either the guest operating system or the host QEMU process.

This paper evaluates the architectures of lightweight isolation platforms based on how they use functionality in the host kernel. We compare the Linux kernel code footprint and performance of three isolation platforms: Linux containers, AWS Firecracker, and Google gVisor. We perform the first fine-grained code coverage analysis of these isolation platforms to determine how kernel services are used differently by each isolation platform: we compare *which* kernel code executes for different workloads as well as the *frequency* of invocation across platforms. For example, we investigate the impact of gVisor's user-mode networking stack on its usage of kernel networking code.

We also perform microbenchmarking of the three platforms compared to native Linux to measure the performance impact of the architectural choices. For CPU, networking, memory, and file system workloads, we measure the achievable performance on all three platforms.

Some highlights of our findings are:

- The code coverage analysis shows that despite moving much operating system functionality out of the kernel, both gVisor and Firecracker execute substantially more Linux kernel code than native Linux alone, and that much of the code executed is not in different functions, but rather conditional code within the same functions executed by native Linux.
- The frequency of invocation varies widely: gVisor, for example, handles most memory mapping operations internally, and hence invokes the kernel much less frequency than Linux containers.
- Our performance results show that neither gVisor nor Firecracker are best of all workloads; Firecracker has high network latency while gVisor is slower for memory management and network streaming.

The rest of the paper is organized as follows. First, we describe how various isolation mechanisms work (§2). We then evaluate the system calls available from each platform and the amount of kernel code executed on each platform (§3). Next, we separately look at the code coverage and performance of CPU (§4), networking (§5), memory management (§6), and file access (§7). Finally, we summarize our findings and discuss the limitations of our work (§8), describe related work (§9), and conclude (§10).

2 Isolation Platforms

In this section, we describe the four isolation platforms that we will analyze (§4-7). First, we introduce LXC-based containers (§2.1), as implemented by Docker's default container engine, runc. Although LXC is sometimes used to refer specifically to the liblxc library [18] that creates and manages containers, we use LXC to more broadly refer to the "capabilities of the Linux kernel (specifically namespaces and control groups) which allow sandboxing" [15]; we explore these features as used by runc. Second, we describe KVM/QEMU (§2.2), a popular virtual machine platform. Finally, we give an overview of gVisor (§2.3) and Firecracker (§2.4), two new platforms that both use a mix of LXC and KVM/QEMU components. gVisor's OCI-compliant [20] runsc engine and ongoing efforts to implement a similar engine for Firecracker [16] suggest it will soon be trivial to choose and switch between LXC, gVisor, and Firecracker when deploying with tools such as Docker and Kubernetes.

2.1 Linux Containers (LXC)

Linux Containers (LXC) [1, 19] are an OS-level virtualization method for running multiple isolated applications sharing an underlying Linux kernel. A *container* consists of one or more processes (generally running with reduced privileges) having shared visibility into kernel objects and a common share of host resources.

Shared visibility into kernel objects is governed by *namespaces*, which prevent processes in one container from interacting with kernel objects, such as files or processes, in another container. Resource allocation is governed by *cgroups* (control groups), provided by the kernel to limit and prioritize resource usage. An LXC container is a set of processes sharing the same collection of namespaces and cgroups.

Docker is an extension of LXC that adds a user-space *Docker daemon* to instantiate and manage containers. The Docker daemon needs root privileges, although a version that drops this requirement is under development. The daemon can be configured to launch containers with different container engines; by default, Docker uses the *runc* engine. It can also be configured to use other container engines such as *runsc*, which wraps gVisor (§2.3). Unless otherwise stated, any evaluation of "Docker" in this paper refers to an evaluation of "runc". The runc engine enables isolation for a running container by creating a set of namespaces and cgroups. Kubernetes functions like Docker in managing and launching containers.

Docker can also initialize storage and networking for the container engine to then use. The storage driver provides a union file system, using the Linux kernel, which allows sharing of read-only files. Files created inside a Docker container are stored on a writable layer private to the container provided through a storage driver. Each container has its own virtual network interface. By default, all containers on a given Docker host communicate through bridge interfaces, which prevents a container from having privileged access to the sockets or interfaces of another container. Interactions between containers are possible through overlay networks or through containers' public ports.

2.2 KVM/QEMU

KVM (Kernel-based Virtual Machine) [10] is a type-2 hypervisor built into the Linux kernel that allows a host machine to run multiple, isolated virtual machines. QEMU [13] is a full-system emulation platform that uses KVM to execute guest code natively using hardware-assisted virtualization and provides memory management, device emulation, and I/O for guest virtual machines. KVM/QEMU provides each guest with private virtualized hardware, such as a network card and disk. On this platform, most operating system functionality resides in the guest OS running inside a virtual machine and in the QEMU process that performs memory management and I/O.

While commonly used for full-system virtualization and in infrastructure-as-a-service clouds, KVM/QEMU virtualization is heavyweight, due to both the large and full-featured QEMU process and to full OS installations running inside virtual machines. As a result, it is too costly to run individual functions in a serverless environment.

2.3 gVisor

Google gVisor [7, 23] is a sandboxed container runtime that uses paravirtualization to isolate containerized applications from the host system without the heavy-weight resource allocation that comes with full virtual machines. It implements a user space kernel, *Sentry*, that is written in the Go Language and runs in a restricted seccomp container. Figure 2 shows gVisor's architecture. All syscalls made by the application are redirected into the Sentry, which implements most system call functionality itself for the 237 syscalls it supports. Sentry makes calls to 53 host syscalls to support its operations. This prevents the application from having any direct interaction with the host through syscalls. gVisor supports two methods of redirecting syscalls: *ptrace-mode* uses ptrace in the Linux kernel to forward syscalls to the sentry and *KVM-mode* uses KVM to trap syscalls before they hit the Linux kernel so they can be forwarded to the sentry. As KVM-mode performs better than ptrace for many workloads [38] and has several benefits over the ptrace platform according to the gVisor documentation [17], we perform all experiments with it enabled.

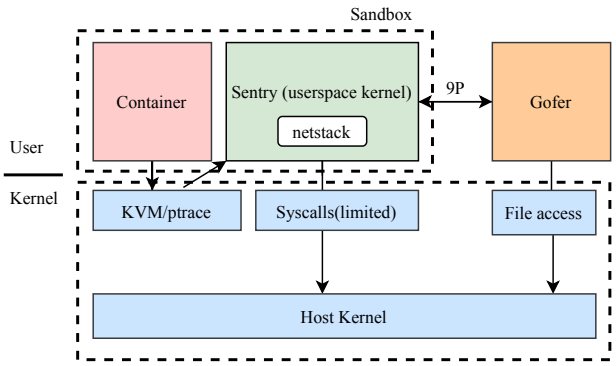


Figure 2. gVisor architecture

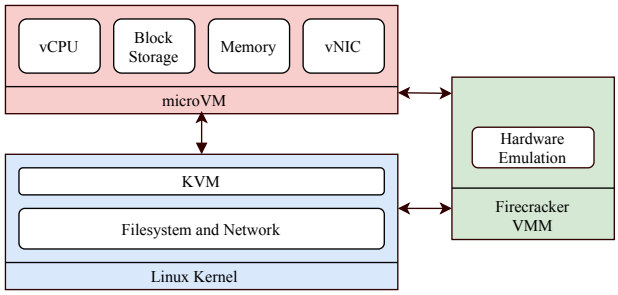


Figure 3. Firecracker architecture

gVisor starts a Gofer process with each container that provides the Sentry with access to file system resources. Thus, a compromised Sentry cannot directly read or write any files. A writable tmpfs can be overlaid on the entire file system to provide complete isolation from the host file system. To enable sharing between the running containers and with the host, a shared file access mode may be used.

gVisor has its own user-space networking stack written in Go called *netstack*. The Sentry uses netstack to handle almost all networking, including TCP connection state, control messages, and packet assembly, rather than relying on kernel code that shares much more state across containers. gVisor also provides an option to use host networking for higher performance.

2.4 Firecracker

AWS Firecracker [4, 26, 32] is a new virtualization technology that uses KVM to create and run virtual machines. Like KVM/QEMU, it uses the KVM hypervisor to launch VM instances on a Linux host and with Linux guest operating system. It has a minimalist design and provides lightweight virtual machines termed *microVMs*. Firecracker excludes unnecessary devices and guest-facing functionality to reduce the memory footprint and attack surface of each microVM. Multiple microVMs can be launched by running separate instances of Firecracker, each running one VM. Compared

to KVM/QEMU, Firecracker has a minimal device model to support the guest operating system and stripped-down functionality. Both of these allow it to make fewer syscalls and execute with fewer privileges than KVM/QEMU.

As shown in Figure 3, each microVM runs as a process within the host OS, which is associated with a dedicated socket and API endpoint. Firecracker provides VirtIO/block and VirtIO/net emulated devices which are backed by files and TAP devices on the host respectively. To ensure fair usage of resource, it enforces rate limiters on each volume and network interface, which can be created and configured by administrators.

Each Firecracker process is started by a *jailer* process. The jailer sets up system resources that require elevated permissions (e.g., cgroups, namespaces, *etc.*), drops privileges, and then launches the Firecracker binary with `exec`, which then runs as an unprivileged process. To further limit the syscalls, Firecracker uses the kernel's seccomp filters to restrict the set of available operations.

Both Firecracker and gVisor rely on host kernel functionality. Firecracker provides a narrower interface to the kernel by starting guest VMs and providing full virtualization, whereas gVisor has a wider interface by being paravirtualized. Although Firecracker uses a minimal device model, its design can be viewed as more heavy-weight than gVisor. They both have low overhead in terms of memory footprint. Firecracker and gVisor are written in Rust and Golang respectively, both being type safe languages adding to their security models.

3 Isolation Platform Comparison

A major distinction between isolation platforms is the location of operating system functionality: is it all handled by the host kernel, or is it moved out to user-space processes or guest kernels within a VM? We evaluate isolation platforms by comparing how they utilize Linux kernel functionality. We measure the *code coverage* of common operations: how much Linux kernel code executes in response to different workloads on each platform? This assists us in understanding how functionality is split between system components.

Furthermore, isolation can directly reduce performance: additional safety checks, layers of indirection such as namespaces, overlay file systems, and virtual networking all add extra code to system call and I/O paths. For multiple workloads, we evaluate the performance of isolation platforms in order to determine the runtime cost of isolation. As a baseline, we compare against native Linux processes with no special isolation. We have not evaluated KVM/QEMU in this paper.

3.1 Methodology

We run all our experiments on Cloudlab [2] xl170 machine, with a ten-core Intel E5-2640v4 running at 2.4 GHz, 64GB ECC Memory (4x 16 GB DDR4-2400 DIMMs), Intel DC S3520 480 GB 6G SATA SSD and 10Gbps NIC. We run on Ubuntu 18.04 (kernel v5.4.13). We performed all the experiments on gVisor release-20200127.0 version and Firecracker v0.19.1. Since these systems are in continuous development, results might vary with the older and future versions.

We run all experiments across four configurations: host Linux with no special isolation, Linux containers (labeled LXC), Firecracker, and gVisor. For gVisor we use KVM-mode to trap system calls from a container. The official performance guide of gVisor [21] uses ptrace mode in almost all their experiments, so many of our performance results are different. For starting a Firecracker microVM, we set the memory to 8GB, storage to 4GB and vCPU to 1. For gVisor and LXC memory was set to 8GB by passing the –memory flag with *docker run*. Each platform's default file systems are used for the measurements.

Platform	**Total allowed syscalls to the host kernel**
LXC	all except 44
Firecracker	36
gVisor w/o host networking	53
gVisor w/ host networking	68

Table 1. Total number of system calls allowed out of 350

3.2 Security Policies

The Linux kernel provides over 350 syscalls as entry points to operating system functionality. Each syscall is a possible vector for attack against the kernel, as a flaw in the kernel could allow user-mode code to subvert OS protection and security mechanisms. However, most programs use only a subset of the available system calls. The remaining system calls should never occur. If they do occur, it is perhaps because of a compromised program.

Linux provides a secure computing mode—*seccomp* [12]—to restrict the system calls a running process may make. We focus on use of *seccomp-bpf*, which allows a user space-created policy to define (a) the permitted syscalls, (b) allowed arguments for those syscalls, and (c) the action to be taken when an illegal syscall is made.

All three systems we study (LXC, gVisor, and Firecracker) use seccomp filters to reduce application-host kernel interaction, thereby increasing isolation and reducing the attack surface. A summary of the allowed syscalls in each system is shown in Table 1.

LXC can use seccomp [3] to limit actions available to the process running inside the container. By default, Docker blocks 44 system calls that are obsolete, can only be called with root privilege, or are not protected by kernel namespaces.

Firecracker has a whitelist of 36 syscalls [6] that it requires to function. It supports two levels of seccomp filtering. With *simple* filtering, only the 36 calls are permitted and the remainder denied; with *advanced* filtering, Firecracker adds

	Host	Firecracker	LXC	gVisor
Lines	63,163	77,392	90,595	91,161
Coverage	7.83%	9.59%	11.23%	11.31%

Table 2. Union of line coverage across all workloads out of 806,318 total lines in the Linux kernel.

constraints on the values allowed as arguments to system calls as well. A syscall is blocked if the argument values do not match the filtering rule.

gVisor limits the syscalls to the host by implementing most of them in the Sentry process [34]. The Sentry itself is only allowed to make a limited set of syscalls to the host. As of November 2019, Sentry supports 237 syscalls out of 350 within the 5.3.11 version of the Linux kernel. It uses a seccomp whitelist of 68 syscalls to the host with host networking enabled and 53 without networking.

Overall, both gVisor and Firecracker provide much stronger system call limits than Linux containers, with Firecracker being a bit stricter due to its use of fewer system calls and tighter controls over the arguments to those system calls. In contrast, Linux containers allow isolated applications to make most system calls.

3.3 Total Code Footprint

We measure the code coverage for four different microbenchmarks, described in later sections: CPU, network, memory, and file write on the four systems (host, LXC, Firecracker and gVisor).

Each workload runs for ten minutes. We run the lcov [11] code coverage tool, version 1.14 on kernel v5.4.13, which reports which lines of source code were executed and what fraction of functions, lines, and branches of total kernel code were executed. It also reports how many times each line of code was executed. Using these results, we can compare the kernel code coverage for the same workload across different isolation platforms to see first, how much code they execute, and second, whether a platform uses the same code as another platform or does something different.

Table 2 shows the union of line coverage across the workloads. The lines row shows the number of lines of code executed at least once out of the total 806,318 lines of code in the linux kernel. Overall, we find that native Linux executes the least code and that both Firecracker and gVisor, despite having a separate guest kernel and a user-space kernel respectively, execute substantially more Linux kernel code.

While this gives us a broad picture, it does not tell us anything about whether the platforms are largely executing the same code and more, or if there are large non-overlapping bodies of code. In the following sections, we take a detailed look at the code executed for different kernel subsystems when running microbenchmarks exercising that subsystem. Figures 4, 6, 13, and 18 show the intersection of code executed by each system, represented as a set of Venn diagrams.

When determining whether to consider a line of code as executed by a platform, we filter out invocations not related to the application running in the isolation platform. To do this, we measure the code coverage of an idle host Linux system for 10 minutes and calculate the *hit rate* of every line of code—how often it is called. When determining which code was executed by a platform, we compare the hit rates of the code with the platform against the idle system and only include the code if it was executed more frequently with the isolation platform present. With this mechanism, we minimize the noise from the background processes, though we cannot eliminate it completely in our measurements. Stripping idle code removes approximately 10k-15k lines of code, depending on platform.

4 CPU Workload

4.1 Coverage Analysis

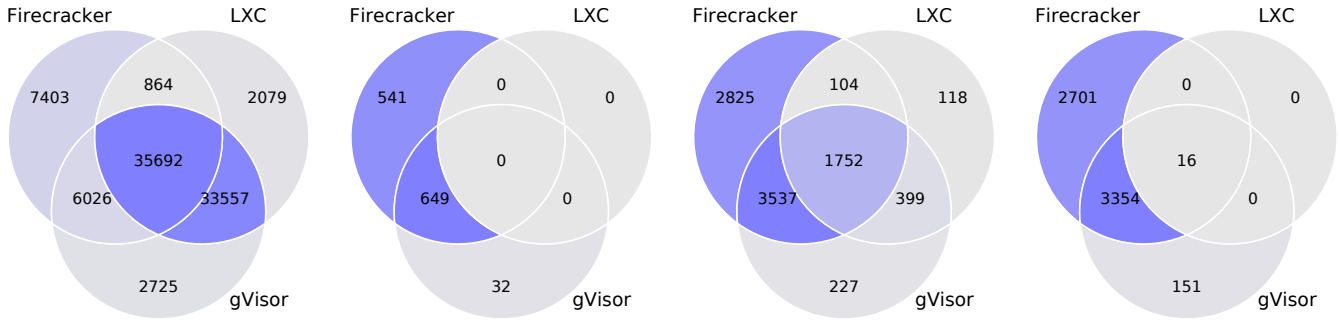
We run the sysbench CPU benchmark [14] on each virtualization platform and identify the footprint in case. Figure 4a shows the lines of code in each footprint across the entire Linux kernel, and how the footprints overlap: darker portions of the diagrams corresponding to larger line counts.

We see a majority of the lines (35,692) are shared by all three platforms; Firecracker has the most unique lines (7,403) not shared by any other platform, and LXC and gVisor share a significant footprint not exercised by Firecracker (33,557 lines). We note that although gVisor implements many system calls in the Sentry, it exercises more host code than LXC; although surprising, the same was found in prior analysis [24]). Overall, gVisor has the largest total footprint (78k LOC) and Firecracker has the smallest (49k LOC).

We illustrate the source of the differences between platforms by analyzing code in specific Linux source directories. The code in */virt* directory shown in Figure 4b is executed only in Firecracker and gVisor, because they virtualize with KVM. Firecracker has the highest footprint here of 1,190 LOC (541+649) as it uses VirtIO emulated Network and Block devices, while gVisor relies on system calls for I/O, resulting in a footprint that is 43% smaller than and nearly a subset of Firecracker's footprint.

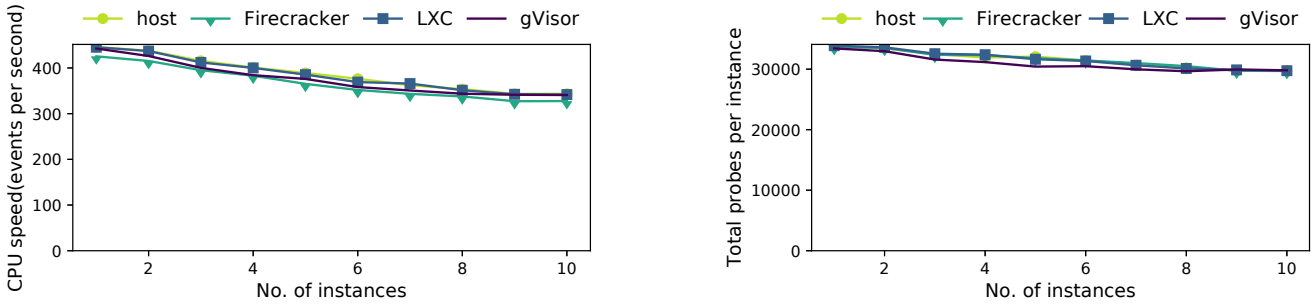
Firecracker and gVisor have a much larger */arch* footprint than LXC (Figure 4c); most of the additional code is under */arch/x86/kvm*, as shown in Figure 4d. Here, we see gVisor's footprint is 42% smaller than and nearly a subset of Firecracker's exercised lines of code. KVM code is used by Firecracker for microVMs and by gVisor to redirect system calls to the Sentry; however, Firecracker has 2,550 more lines of code than gVisor. This difference is mostly from the emulation code that Firecracker executes but gVisor does not.

Despite these substantial differences in functionality, much of the new code executed by gVisor and Firecracker is not in new functions. Rather, it is conditional code in the same functions exercised by containers. For instance most of the



(a) CPU workload overall **(b)** /virt **(c)** /arch **(d)** /arch/x86/kvm

Figure 4. CPU workload coverage



(a) CPU Speed **(b)** LLC Probe

Figure 5. CPU workload

code in the file *block/blk-cgroup.c* is executed just by gVisor as an extra check. We will see more such examples in the following sections.

4.2 Performance

We conduct two experiments to measure how choice of isolation platform affects CPU performance. First, we run the sysbench CPU benchmark [14] for ten seconds and observe the number of events it executes. An event is the loop that finds prime numbers up to a limit. This workload is CPU-bound, and hence measures the processing overhead of the platforms. We observe in Figure 5a that all the platforms perform similarly. As we increase the number of instances to ten, performance per instance drops 23%.

In the second experiment, we execute LLCProbe [35], which sequentially probes 4B from every 64B of data within an LLC-sized buffer, using cache coloring to balance access across cache sets. To measure the performance, we measure the number of probes completed in 10 seconds. Figure 5b reports the average probes observed per instance. There is a slight drop in performance as the number of instances increases across all the environment, due to overhead of the platform. All platforms achieve approximately 33k probes per instance.

4.3 Insights

Despite the user-mode Sentry, *gVisor has the largest footprint and Firecracker the smallest.* But, all three platforms exercise a significant body of code not exercised by the other two. While not all lines of code are equally likely to be vulnerable to an exploit, more code indicates potentially higher risk.

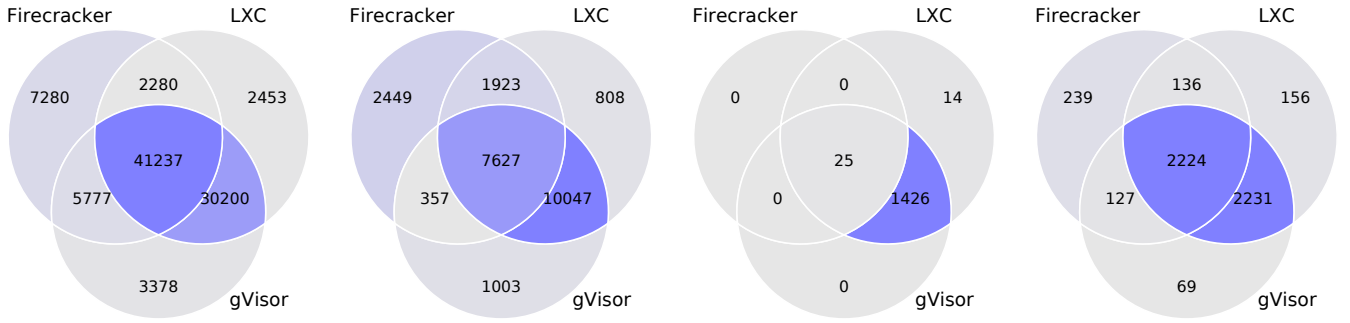
Firecracker and gVisor require more architecture-specific code due to their dependence on hardware virtualization, which may make portability to other processor architectures more difficult. Firecracker depends on KVM to achieve the lightweight virtualization, adding to the lines of code it executes in the kernel. gVisor is heavily paravirtualized and also uses a smaller fraction of this functionality when running with the KVM platform. LXC does not use this functionality as it lies on the leftmost side of the isolation spectrum as seen in Figure 1.

The CPU performance corresponding to these architectural differences does not vary significantly for both the workloads, as overall there is fairly little kernel involvement.

5 Networking

5.1 Coverage Analysis

We measure code coverage of network operations for each platform using iperf3 [9], which streams data over TCP/IP.



(a) Network workload overall **(b)** /net **(c)** /net/bridge **(d)** /net/core

Figure 6. Network Workload Coverage

Figure 6a shows the overall kernel footprints and Figure 6b shows the footprints in the */net* directory alone. Despite having an in-Sentry networking stack (including TCP, UDP, IP4, IP6 and ICMP), gVisor has the highest overall coverage and surprisingly exercises much of the same code as LXC under */net*. Some parsing of packets still happens in the host kernel as a sanity check before routing them. Similarly, when Generic Segmentation Offload (GSO) is enabled in gVisor, the host kernel does more processing of packets. The */net/bridge* and */net/core* directories also show a high overlap among them with 1,426 and 2,231 lines shared by LXC and gVisor as presented in Figures 6c and 6d respectively.

Firecracker runs the fewest lines of networking code. This reflects that most networking happens in the guest OS.

5.2 Code Differences

In comparison to the host, all three isolation platforms execute more setup code for their network interfaces and more extra checks across all networking code. *dev_get_by_index()*, defined in */net/core/dev.c*, searches for a network device; it only gets executed in LXC and gVisor but not on host and Firecracker which suggests that gVisor and LXC have the same type of network interface, i.e., a bridge network.

There are some functions that execute only on some of the platforms, such as *tcp_sum_lost* in */net/ipv4/tcp_input.c*, which checks the sum of the packets lost on the wire. This gets executed 27,719 times in LXC, 3 times in Firecracker and not executed in gVisor. This shows functionality that gVisor handles in Sentry; hence not using the kernel network stack for this processing.

Figure 7a shows the code snippet and Figure 7b shows the hits (number of times a line executes) for some of the lines. LXC has the highest hits, more than 100 million, followed by gVisor, which calls this function 1,805,020[12] times showing

```
bool is_skb_forwardable(const struct net_device *dev,          1823
        const struct sk_buff *skb){
    unsigned int len;                                          1824
    if (!(dev->flags & IFF_UP))                                1825
        return false;                                          1826
    len = dev->mtu + dev->hard_header_len +                    1827
        VLAN_HLEN;
    if (skb->len <= len)                                       1828
        return true;                                           1829
    if (skb_is_gso(skb))                                       1830
        return true;                                           1831
    return false;                                              1832
}                                                              1833
```

(a) net/core/dev.c

	Hits		
Lines	**LXC**	**gVisor**	**Firecracker**
1823	1.012e+8	1.805e+6	0
1825	2.111e+8	5.099e+6	0
1827	2.111e+8	5.099e+6	0
1828	2.111e+8	5.099e+6	0
1830	8.270e+6	9.974e+5	0

(b) Number of hits in net/core/dev.c

Figure 7. Hits in net/core/dev.c

that it handles part of this inside netstack. Firecracker does not execute this code.

5.3 Performance

We evaluate network performance by measuring bandwidth and latency. The network bandwidth is measured by running iperf3 between two Cloudlab machines with a 10 Gbps network link and the latency by measuring the round-trip time with ping.

Host and Firecracker achieve the highest bandwidth; LXC is slightly slower while gVisor is the slowest with 0.805 Gbps[3], (805 Mbps) but the aggregate increases to 3.294 Gbps

[1]https://stackoverflow.com/questions/46447674/function-coverage-is-lesser-even-with-100-code-coverage-when-objects-created-in
[2]https://stackoverflow.com/questions/2780950/gcov-line-count-is-different-from-no-of-lines-in-source-code

[3]We get similar result on Google Cloud Platform (GCP).

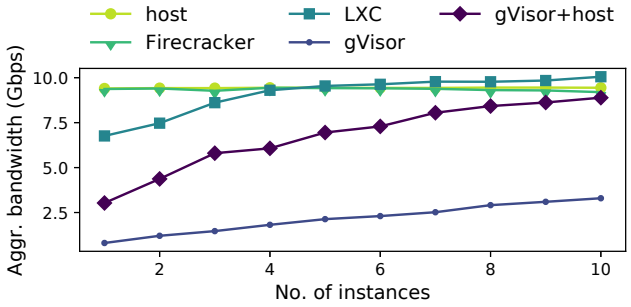


Figure 8. Aggregate Network Bandwidth

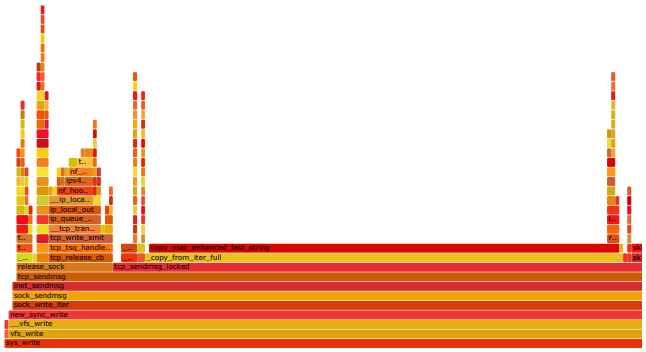


Figure 9. Host network profile

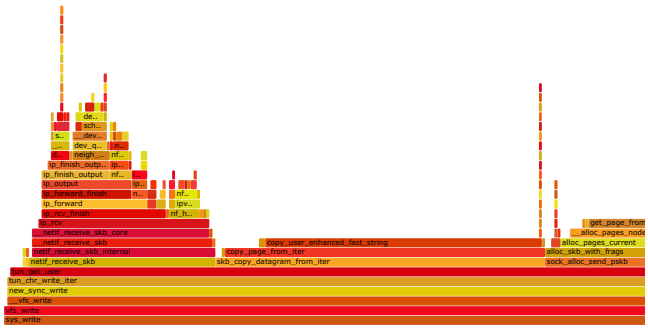


Figure 10. Firecracker network profile

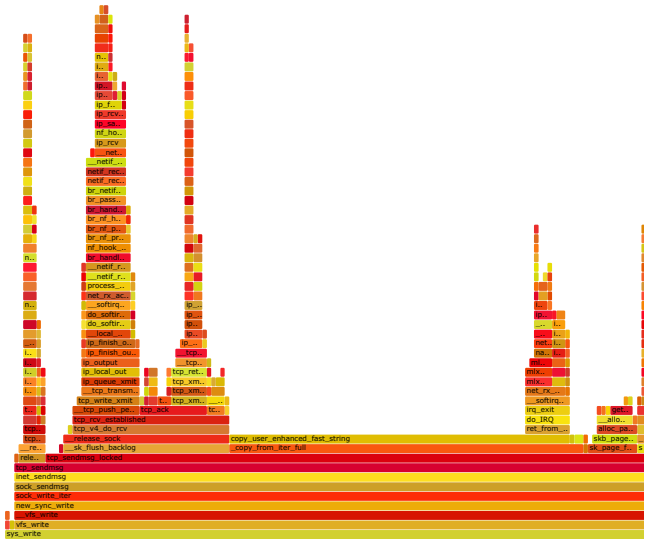


Figure 11. LXC network profile

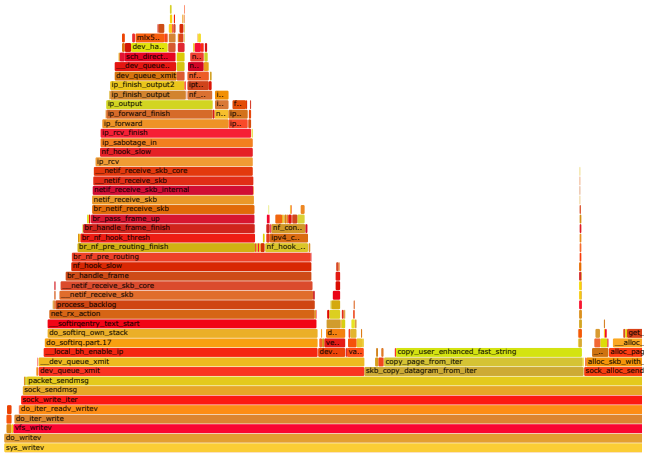


Figure 12. gVisor network profile

at 10 instances. This is likely due to its user space network stack, which is not as optimized as the Linux network stack. When using the host networking stack, gVisor is still slowest, but achieves 3.03 Gbps. The performance of ptrace platform as stated in the gVisor official guide [21] is faster. The gVisor network performance for the KVM platform has increased substantially from its release-20190304 version, nearly by 800%.

To better understand how the different platform architectures use the kernel differently, we show flame graphs [28] for the *sys_write* system call in Figures 9–12. Each rectangle represents a function, and the stacks represent call stacks. The width represents the fraction of time that function was on the call stack. Several patterns are visible. Compared to host networking in Figure 9, LXC in Figure 11 has a much taller stack, which represents the cost of traversing two network interfaces to implement bridging. For gVisor, the trace is done for the Sentry process that does a lot of processing on the packet. Firecracker runs TCP in the guest, so it invokes packet forwarding code at the IP layer and has a much flatter profile.

We show the aggregate bandwidth as the number of instances increases in Figure 8. With increasing instances, there is almost no change in aggregate and average bandwidth in most cases. For gVisor with host networking, the aggregate increases with more instances, but peaks below the other platforms at 10 instances.

We measured the round-trip time (RTT) by running ping for ten seconds and taking the average RTT shown in Table 3.

	Host	Firecracker	LXC	gVisor
RTT (μs)	146	371	149	319

Table 3. Round-trip time

Firecracker incurs maximum latency followed by gVisor, LXC and host. In case of Firecracker, a packet traverses through two levels of kernel network stack, that explains its high latency. gVisor does not go through all the layers of host network making its latency slightly better than Firecracker, but still far from host or LXC.

5.4 Insights

For the network workload we observe that *Firecracker has most of the implementation inside the guest and uses less functionality of the host compared to gVisor and LXC.* It also shows high performance. *Even though gVisor has its own networking stack, it touches a lot of kernel code.* LXC uses much of the same code as the host, but adds substantially to it for bridging, which comes at little performance cost.

Although LXC has high performance, it relies completely on the kernel for all the packet processing and sanity checks, which lowers its isolation. gVisor does most of the processing inside netstack, but some checks are still performed at the host, which provides more isolation. Firecracker has almost all the processing inside the guest, which makes its networking stack relatively more isolated.

6 Memory management

6.1 Coverage Analysis

We show the overall kernel coverage in Figure 13a for a workload that mmaps and munmaps regions. The */mm* directory in 13b shows 1,743 lines of common code between LXC and gVisor. Memory management in gVisor [8] has two levels of virtual to physical page mapping, one from application to Sentry and the other from Sentry to host. Sentry requests memory from host in 16MB chunks to minimize the number of mmap() calls. For a 1GB memory request by an application, Sentry will make 64 mmap() calls to the host. Although gVisor reduces the number of such calls, it still executes the same lines of code as LXC for any memory request. Thus, we observe both having similar coverage in this directory. Firecracker has lower coverage here with only 54 unique lines of code. It does not make any mmap() calls to the host after the first initialization needed to run the microVM and has only 54 unique lines of code.

Figure 13c shows the architecture specific memory management code with Firecracker having 211 unique lines of code. LXC and gVisor have similar footprints in this directory. Although gVisor implements memory management in the Sentry, it still depends on the host for this functionality like LXC, whereas Firecracker implements the whole memory stack inside the guest and does not use this after running some initial architecture specific setup code.

6.2 Code Differences

Several kernel functions show significant differences in invocation frequency. The *do_mmap()* function in *mm/mmap.c* for the memory workload is called more than 1 million times by LXC, but only 5,382 times from gVisor, due to the two-level page table implemented in Sentry that reduces the number of syscalls to the host. Firecracker invokes this function only 3,741 times, as it is only used by the VMM for physical page management and not in response to guest mmap() calls.

zap_page_range_single() defined in *mm/memory.c* which removes user pages in a given range is executed 459,984 times in gVisor and 0 times in Firecracker and LXC. This is likely due to the memory management implementation inside the Sentry process.

The *vma_wants_writenotify()* in */mm/mmap.c* checks for write events in pages marked read-only. This function gets called in all the three platforms, but only Firecracker and gVisor execute the conditional code inside it.

6.3 Performance

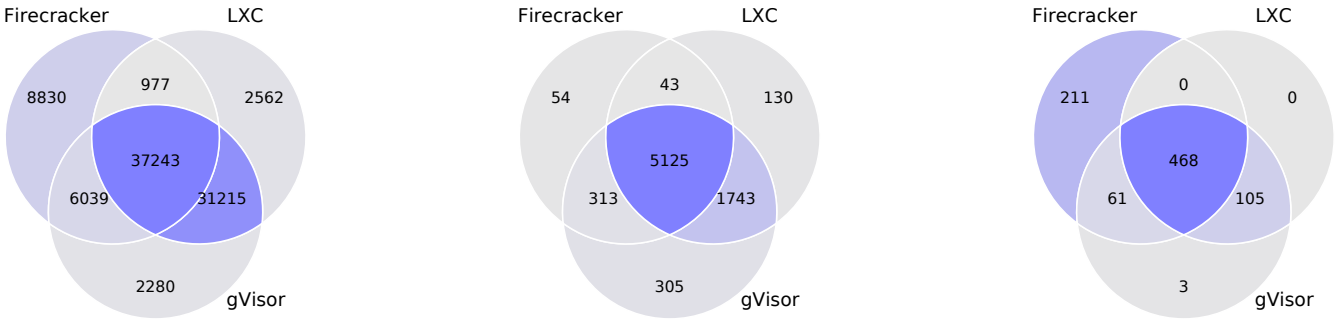
We benchmark memory allocation cost by allocating and freeing memory with mmap() and munmap() system calls. The benchmark allocates 1 GB of memory using mmap() calls with varying granularity from 4KB to 1MB, so with larger requests there are far fewer allocations. After each call, the test program touches one word of each allocated page by writing to it. We also run the same experiment with calling munmap() after touching the pages of each allocation so physical memory and virtual addresses can be recycled.

In Figures 14 and 15, we present the time taken for mmap() and touch respectively. In this case we do not unmap pages after each allocation. In Figure 16, time includes mmap() and munmap(), and the touch time is presented separately in Figure 17.

For both the cases, gVisor is expensive, taking the highest time for allocation, allocation+munmap, and touch. However, we observe that it becomes competitive with the increase in mmap size. We suspect this is due to the two levels of page mapping, from application to Sentry and then to the host which needs management of more code and data structures contributing to this overhead. Firecracker and LXC perform similarly to the host in most of the cases, but becomes more expensive if we unmap, likely due to higher costs of shootdowns on virtualized platforms.

6.4 Insights

The large difference in invocation frequency for gVisor occurs because it has moved some of the memory management inside Sentry. This design makes it very expensive for workloads that do allocation in smaller chunks sizes. *Firecracker does not use any of the host functionality after initializing the microVM*, making the memory stack fully isolated with high performance in most cases. But system virtualization



(a) Memory workload overall **(b)** /mm **(c)** /arch/x86/mm

Figure 13. Memory workload coverage

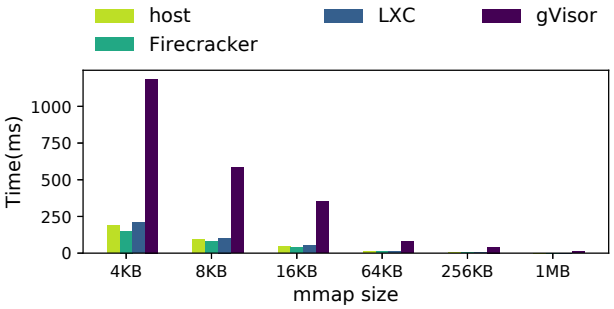


Figure 14. Total allocation time (without munmap) for 1GB

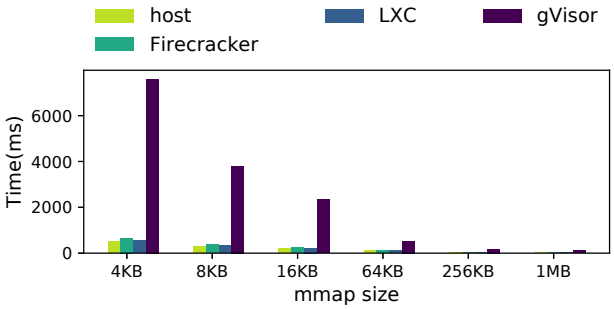


Figure 16. Total allocation+unmap time for 1GB

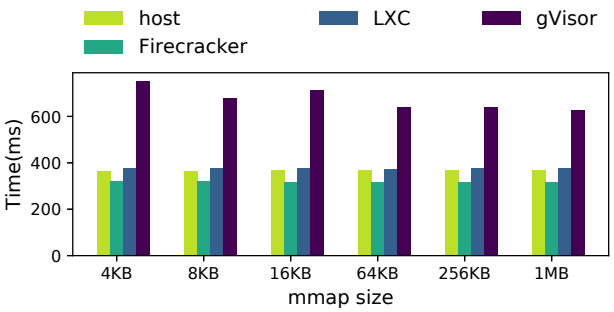


Figure 15. Total touch time (without munmap) for 1GB

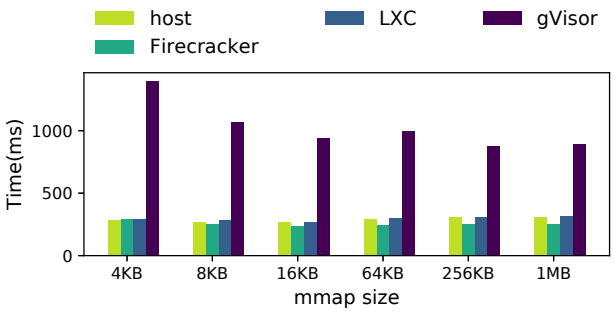


Figure 17. Total touch time (with munmap) for 1GB

becomes slower when we start unmapping and re-mapping, so for workloads that do a lot of mmap() and munmap(), this might come with an overhead. *LXC heavily uses the kernel code with very high hit rates*, which makes it less isolated. But it has high performance, better than Firecracker in some cases.

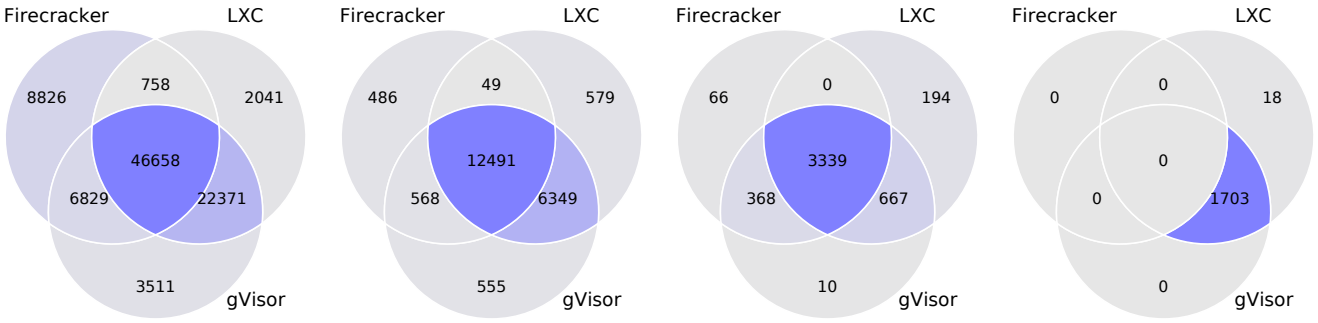
7 File access

7.1 Coverage Analysis

We measure coverage with a workload that opens, reads, and writes to files. Figure 18a shows the overall kernel footprints and Figure 18b shows the footprints in the */fs* directory, which includes VFS code as well as the implementations for various file systems. We observe that most lines (about 18k) are shared between gVisor and LXC; Firecracker uses about half of these lines as well.

In our experiments, all three systems use the ext4 file system. Figure 18c shows that gVisor and LXC exercise more ext4 code than Firecracker. Storage in Firecracker is provided by an emulated block device, which is backed by a file on the host, so there are no directory operations.

gVisor and LXC containers do not directly access ext4 files; rather, they interact with overlayfs, which provides copy-on-write and stitches together a single file-system view from multiple ext4 directories. Figure 18d show that LXC



(a) File write workload overall **(b)** /fs **(c)** /fs/ext4 **(d)** fs/overlayfs

Figure 18. File write workload coverage

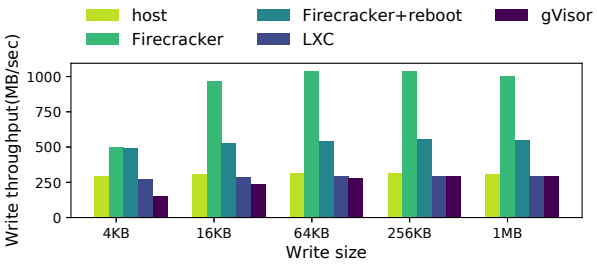


Figure 19. Write Throughput

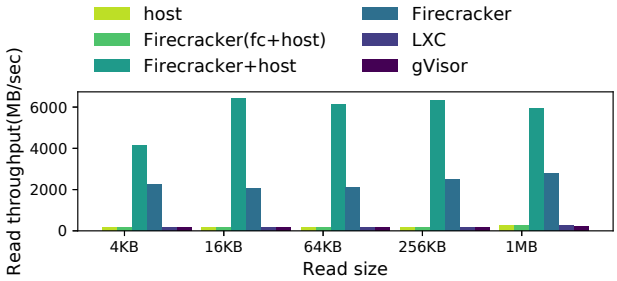


Figure 20. Read Throughput

and gVisor have a similar overlayfs footprint, and that (as expected) Firecracker makes no use of overlayfs.

7.2 Performance

We measure I/O throughput with a microbenchmark that performs reads and writes of various sizes on a 1 GB file. For writes, the test creates the file and then writes sequentially. After each iteration, the file is flushed to disk; the cache is cleared between each read test.

For the write throughput shown in Figure 19, we run two variations for Firecracker. In the first case, we run the test repeatedly on the same microVM. In the second case, we reboot the microVM before each test. This causes Firecracker to close and reopen the backing file for its block device, which may account for the lower performance following a reboot. Nonetheless, in both cases Firecracker is much faster than the other platforms because it makes no effort to write data back to persistent storage. As a result, it operates much closer to the speed of memory. In contrast, gVisor and LXC perform very similarly to native Linux; gVisor is a bit slower for small files where the overhead of calling through the Sentry is not masked by data movement costs.

Read throughout is shown in Figure 20. Host, LXC, gVisor all perform similar across all read sizes, as they all must read from storage. For Firecracker we run three variations: plain *Firecracker* flushes the file cache in the microVM only; *Firecracker+host*, flushes the cache only in host Linux and not the microVM, and *Firecracker(fc+host)* flushes caches in both locations. When all caches are flushed, Firecracker behaves similarly to the other platforms, as it must fetch data from storage. When only the cache in the microVM is flushed, performance is closer to memory speed, but still pays the cost of exiting the VM to copy data from the cache in the host kernel. Finally, when just the host cache is flushed, performance is near memory speed: the data can still be accessed from the guest OS cache. Overall, a warm cache within the microVM results in nearly double the throughput as a warm cache in the host.

7.3 Insights

Firecracker only uses ext4 as a backing for a virtual disk; this results in only a moderate footprint since this use case only involves file-system data, not metadata. In contrast, *gVisor and LXC use overlayfs over ext4; this results in increased footprint for overlayfs, as well as more ext4 code being exercised* (overlay file systems heavily use metadata in the underlying file system).

8 Summary

Outcomes The preceding sections analyzed how LXC, Firecracker, and gVisor make different use of kernel functionality, and how they exercise different kernel code. Our analysis demonstrated several aspects of these isolation platforms.

First, using Linux containers, while highly performant, greatly increases the amount of kernel code exercised. In some cases, this was new functionality that greatly lengthened code paths, such as networking where bridging expanded the amount of code needed for transmitting data. However, in many cases the new code was not an entire module, but instead was conditional code interspersed with code run by native Linux. In such cases, existing functions execute more slowly due to the added code.

Second, Firecracker's microVMs are effective at reducing the frequency of kernel code invocations, but had a much smaller impact on reducing the footprint of kernel code. Indeed, running workloads under Firecracker often expanded the amount of kernel code executed with support for virtualization. In addition, Firecracker's use of more limited functionality, such as memory mapping only at startup, file I/O for data but not directories, and networking at the IP level, show that future kernel releases targeting microVMs could aggressively optimize specific hot code paths at the expense of sacrificing performance of more general workloads needed by a general purpose OS.

Finally, we found that the gVisor design leads to much duplicated functionality: while gVisor handles the majority of system calls in its user-space Sentry, it still depends on the same kernel functionality as Linux containers. Thus, its design is inherently more complicated, as it depends on multiple independent implementations of similar functionality. In contrast, LXC relies on a single implementation of OS functions in the Linux kernel, and Firecracker relies on a much-reduced set of kernel functionality for performance.

We believe our findings serve as an architectural guide for building future systems and also be a motivation for future research concerning the coverage and security of new isolation platforms.

Limitations Our methodology is a middle ground between full-system profiling and foreground-only profiling, and may err by capturing a bit of background activity, even though the goal is capturing all kernel activity induced from the isolation platform. Furthermore, while we focus on code coverage, studying coverage with microbenchmarks only provides a hint as to the security of a system; more investigation of coverage under richer workloads, including exploring the total coverage available from a platform, may be needed to make stronger security claims.

The microbenchmarks we use for the study allow us to stress particular subsystems and study their performance and coverage. While running more complex workloads or test suites would provide more information about total code coverage, their complexity could obscure significant differences in usage of different subsystems. Moreover, running existing containerized benchmarks is a challenge because Firecracker is not yet integrated with Docker.

9 Related Work

In this work, we explore the host kernel footprint and performance of Firecracker [4] and gVisor [7]. These are both KVM-based sandboxes attempting to compete on performance with traditional LXC containers [18], on which Docker is built. New sandboxing systems with similar goals include Kata Containers [29] (which attempt to provide "The speed of containers, the security of VMs"), Nabla containers [24], and LightVM, built by Manco *et al.* [31]. Manco *et al.* argue that traditional containers are insecure because malicious code has access to a large attack surface (all Linux system calls); many of the new systems are structured much like Drawbridge [33], in an attempt to narrow this interface. Narrowed interfaces are then typically protected with stricter seccomp [12] filters. Our methodology is not specific to gVisor and Firecracker; it could be applied to these other isolation platforms as well.

Both gVisor and Firecracker restrict the set of system calls available to improve security and isolation. Narrowing the interface of a platform will not necessarily improve security, unless it reduces the footprint of host code likely to contain vulnerabilities, so we have focused on understanding the kernel footprint underneath each virtualization system. Bottomley [24] takes a similar view and measures the footprint of various sandboxes at function granularity; we explore footprint at line and branch granularity, using lcov [11]. gVisor implements the network stack in the Sentry; while the work of Williams *et al.* [25] shows that moving subsystems out of the host kernel can significantly reduce kernel footprint, our work shows gVisor and LXC have a surprisingly large coverage overlap in `/net`.

In addition to measuring code footprint, we also evaluate performance, building on prior virtualization performance analysis studies [22, 27, 30, 36–38]. Our work differs from past studies in that it focuses on the performance impact of architectural choices and identifies the kernel code differences that play a role in performance.

10 Conclusion

In this paper, we presented a system-wide Linux code-coverage analysis for three isolation platforms. This enables us to study the impact of different isolation architectures that move more and more code out of the kernel. We also ran performance benchmarks to understand the runtime cost of each platform. Our results show that despite adding substantial code outside the kernel for OS functionality, both gVisor and Firecracker sill exercise more kernel code for most workloads.

11 Acknowledgments

We would like to thank our shepherd, Tim Harris, and the reviewers for their feedback. This work was supported by the National Science Foundation grant CNS-1763810.

References

[1] 2019. CHAPTER 1. INTRODUCTION TO LINUX CONTAINERS. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html/overview_of_containers_in_red_hat_systems/introduction_to_linux_containers.

[2] 2019. Cloudlab. https://www.cloudlab.us/.

[3] 2019. Docker seccomp. https://docs.docker.com/engine/security/seccomp/.

[4] 2019. Firecracker documentation. https://github.com/firecracker-microvm/firecracker/tree/master/docs.

[5] 2019. Firecracker Getting started. https://github.com/firecracker-microvm/firecracker/blob/master/docs/getting-started.md.

[6] 2019. Firecracker seccomp. https://github.com/firecracker-microvm/firecracker/tree/master/src/vmm/src/default_syscalls.

[7] 2019. gVisor Documentation. https://gvisor.dev/docs/.

[8] 2019. gVisor mm. https://github.com/google/gvisor/tree/master/pkg/sentry/mm.

[9] 2019. iperf3. https://github.com/esnet/iperf.

[10] 2019. KVM. https://wiki.archlinux.org/index.php/KVM.

[11] 2019. lcov. http://ltp.sourceforge.net/coverage/lcov.php.

[12] 2019. Linux seccomp. http://man7.org/linux/man-pages/man2/seccomp.2.html.

[13] 2019. QEMU. https://wiki.archlinux.org/index.php/QEMU.

[14] 2019. sysbench. https://github.com/akopytov/sysbench.

[15] 2020. Docker frequently asked questions (FAQ). https://docs.docker.com/engine/faq/.

[16] 2020. firecracker-containerd. https://github.com/firecracker-microvm/firecracker-containerd.

[17] 2020. gVisor Platforms. https://gvisor.dev/docs/user_guide/platforms/.

[18] 2020. Linux containers. https://linuxcontainers.org/lxc/introduction/.

[19] 2020. LXC. https://en.wikipedia.org/wiki/LXC.

[20] 2020. Open Container Initiative. https://www.opencontainers.org/.

[21] 2020. Performance Guide. https://gvisor.dev/docs/architecture_guide/performance/.

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

[23] Antoine Beaupré. 2018. Updates in container isolation. https://lwn.net/Articles/754433/.

[24] James Bottomley. 2018. Measuring the Horizontal Attack Profile of Nabla Containers. https://blog.hansenpartnership.com/measuring-the-horizontal-attack-profile-of-nabla-containers/.

[25] Dan Williams and Ricardo Koller and Brandon Lum. 2018. Say Goodbye to Virtualization for a Safer Cloud. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotcloud18/presentation/williams

[26] Azhar Desai. 2019. The Firecracker virtual machine monitor. https://lwn.net/Articles/775736/.

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

[28] Brendan Gregg. 2019. Flame Graphs. http://www.brendangregg.com/flamegraphs.html.

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

[30] Zhanibek Kozhirbayev and Richard O. Sinnott. 2017. A Performance Comparison of Container-based Technologies for the Cloud. http://www.sciencedirect.com/science/article/pii/S0167739X16303041. *Future Generation Computer Systems* 68 (2017).

[31] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. http://dl.acm.org/citation.cfm?doid=3132747.3132763. *Proceedings of the 26th Symposium on Operating Systems Principles* (2017).

[32] Janakiram MSV. 2018. How Firecracker Is Going to Set Modern Infrastructure on Fire. https://thenewstack.io/how-firecracker-is-going-to-set-modern-infrastructure-on-fire/.

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

[34] Jeremiah Spradlin and Zach Koopmans. 2019. gVisor Security Basics - Part 1. https://gvisor.dev/blog/2019/11/18/gvisor-security-basics-part-1/.

[35] Venkatanathan Varadarajan, Thawan Kooburat, Benjamin Farley, Thomas Ristenpart, and Michael M. Swift. 2012. Resource-freeing Attacks: Improve Your Cloud Performance (at Your Neighbor's Expense). In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (Raleigh, North Carolina, USA) *(CCS '12)*. ACM, New York, NY, USA, 281–292. https://doi.org/10.1145/2382196.2382228

[36] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, Berkeley, CA, USA, 133–145. http://dl.acm.org/citation.cfm?id=3277355.3277369

[37] Xu Wang. 2018. 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.

[38] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2019. The True Cost of Containing: A gVisor Case Study. In *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing* (Renton, WA, USA) *(HotCloud'19)*. USENIX Association, Berkeley, CA, USA, 16–16. http://dl.acm.org/citation.cfm?id=3357034.3357054