applied sciences

Article

Impact of Secure Container Runtimes on File I/O Performance in Edge Computing

Kyungwoon Lee [1], Jeongsu Kim [2], Ik-Hyeon Kwon [3], Hyunchan Park [4,*] and Cheol-Ho Hong [5,*]

[1] School of Electronics Engineering, Kyungpook National University, Daugu 41566, Republic of Korea; kwlee87@knu.ac.kr
[2] Agency for Defense Development, Daejeon 34186, Republic of Korea; jeongsukim0122@gmail.com
[3] School of Electrical and Electronics Engineering, Chung-Ang University, Seoul 06974, Republic of Korea; inzulmi@cau.ac.kr
[4] Department of Computer Science and Artificial Intelligence/CAIIT, Jeonbuk National University, Jeonju 54896, Republic of Korea
[5] Department of Intelligent Semiconductor Engineering, Chung-Ang University, Seoul 06974, Republic of Korea
* Correspondence: hyunchan.park@jbnu.ac.kr (H.P.); cheolhohong@cau.ac.kr (C.-H.H.)

Abstract: Containers enable high performance and easy deployment due to their lightweight architecture, thus facilitating resource utilization in edge computing nodes. Secure container runtimes have attracted significant attention because of the necessity for overcoming the security vulnerabilities of containers. As the runtimes adopt an additional layer such as virtual machines and user-space kernels to enforce isolation, the container performance can be degraded. Even though previous studies presented experimental results on performance evaluations of secure container runtimes, they lack a detailed analysis of the root causes that affect the performance of the runtimes. This paper explores the architecture of three secure container runtimes in detail: Kata containers, gVisor, and Firecracker. We focus on file I/O, which is one of the key aspects of container performance. In addition, we present the results of the user- and kernel-level profiling and reveal the major factors that impact the file I/O performance of the runtimes. As a result, we observe three key findings: (1) Firecracker shows the highest file I/O performance as it allows for utilizing the page cache inside VMs, and (2) Kata containers offer the lowest file I/O performance by consuming the largest amount of CPU resources. Also, we observe that gVisor scales well as the block size increases because the file I/O requests are mainly handled by the host OS similar to native applications.

Keywords: container runtimes; security; I/O performance; kernel-level profiling

Citation: Lee, K.; Kim, J.; Kwon, I.-H.; Park, H.; Hong, C.-H. Impact of Secure Container Runtimes on File I/O Performance in Edge Computing. *Appl. Sci.* **2023**, *13*, 13329. https://doi.org/10.3390/app132413329

Academic Editor: Christos Bouras

Received: 21 November 2023
Revised: 8 December 2023
Accepted: 9 December 2023
Published: 18 December 2023

1. Introduction

Containers offer many advantages such as high performance and easy deployment owing to their lightweight architecture [1–4] that allows multiple containers to share the same operating systems (OSs) and system libraries [1]. Containers, which enable lightweight virtualization, also make it easier to utilize performance resources in edge computing nodes [5]. Even though the shared architecture provides high efficiency for running multiple applications on the respective containers, it has a large attack surface that contributes to its security vulnerabilities [6–8]. For example, it has been reported that attackers can overwrite the host runc binary by exploiting the ability of Docker containers to execute a command with a *root privilege* in the containers [6]. Compromised containers can leak important data in the host server to the attacker or manipulate the entire system. Thus, such security vulnerabilities can pose a key barrier to offering secure execution environments [8].

To overcome the security issues of containers, several container runtimes that enforce *isolation* (i.e., secure container runtimes) have been introduced [9–11]. Figure 1 illustrates the architectures of three representative secure runtimes: Kata containers, gVisor, and Firecracker. First, Kata containers [11] were developed by the Linux Foundation, which combines Intel Clear Containers and the Hyper.sh runV project, in 2017. Kata containers adopt a virtual machine (VM) layer to offer isolated execution environments to the containers as illustrated in Figure 1a. Each container runs on a VM equipped with a kernel image and a root filesystem. This differs from existing Docker containers (based on the *runc* container runtime) that run on the same kernel image [2]. Second, Google has introduced gVisor [10] (Figure 1b) that is based on the user-space kernel called *Sentry*. *Sentry* filters system calls executed by the container in order to prevent the container from performing unauthorized system calls directly. So, the system calls are translated by *Sentry* before they get delivered to the actual filesystem or host devices. By filtering the system calls, gVisor can eliminate the threat in which the host system is manipulated by malicious containers. Finally, Firecracker has been presented by Amazon as a virtualization platform for the serverless computing service, Lambda [12]. In Firecracker (Figure 1c), applications are run on a VM rather than containers, and each VM has an independent kernel image and root filesystem. However, in contrast to Kata containers that run with the existing KVM/QEMU techniques for virtualized environments, Firecracker utilizes the Firecracker virtual machine monitor (VMM) that is optimized from QEMU [13] for lightweight execution environments [9] to run multiple VMs on a host server efficiently.

Table 1 summarizes the architectures and I/O processing mechanisms of three secure runtimes. For example, Kata containers adopt virtio-fs [14] to handle the file I/O requests of containers running on the VM, whereas Firecracker offers the virtio block device [15] to the VM. Because the file I/O performance is one of the major factors determining the application performance [16,17], it is important to have an in-depth understanding of the file I/O performance characteristics of secure container runtimes. However, only a few papers [9,18–20] have been published on this topic because the runtimes have been introduced within the last two years.

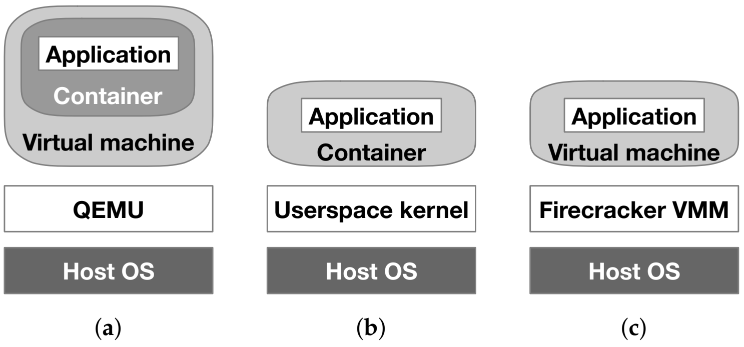


Figure 1. Different architectures of representative secure container runtimes: Kata containers, gVisor, and Firecracker. (**a**) Kata containers, (**b**) gVisor, and (**c**) Firecracker.

Table 1. Detailed mechanisms of secure container runtimes.

	Kata Containers	gVisor	Firecracker
Architecture	Containers on VMs	User-space kernel	Firecracker VMM-based microVM
VMM technology	KVM	KVM	KVM
I/O processing	virtio-fs	System call filtering	virtio-blk
Programming language	Go	Go	Rust

Previous studies [9,18–21] mainly focus on quantitative comparisons of the runtimes in terms of CPU, memory, and I/O performance without providing a detailed analysis of the differences between the runtimes. For example, the performances of gVisor and

Firecracker have been compared with runc [18] and QEMU [9]. Even though these studies offer evaluations of the CPU, memory, and I/O performance, they do not provide sufficient explanation for the reasons behind the performance differences. Similarly, research works [19,20] present the performance of gVisor and Firecracker in terms of CPU, memory, network, and file I/O. However, they do not describe the difference in their file I/O processing and the corresponding impact on the file I/O performance. In addition, previous studies lack an analysis of the resource usage such as the CPU in each runtime when performing file I/O processing. In edge computing, it is important to utilize computing resources efficiently [22,23] so that the resource usage analysis is essential for the container runtimes to properly provision computing resources. For a better understanding of the secure runtimes, the goal of this paper is to present a comprehensive and detailed analysis on the file I/O processing and CPU usage of the runtimes and find out how their architectures impact the file I/O performance (explained in Section 5).

In this paper, we first analyze the file I/O operations of each runtime (Section 4) and examine the entire file I/O stacks. We explain how the file I/O requests of containers are handled, including symbol-level profiling. Second, we present the experimental results on CPU usage as well as file I/O performance, which shows the performance and efficiency of the runtimes when executing file I/O operations (Section 5). The experimental results demonstrate that among the three runtimes, Firecracker has the best performance with the lowest CPU usage. Kata containers offer the lowest performance with the highest CPU consumption. Last, we provide symbol-level profiling results that point out the root cause of the differences in the performance and CPU usage, which can suggest research directions for improving the file I/O performance of the runtimes (Section 6). We find that the virtio block driver of Firecracker allows the applications to exploit the page cache inside the VM, which results in the high performance of Firecracker. On the other hand, Kata containers incur high overhead for two reasons. First, it handles page faults caused by the VM when the application performs read and write operations on the shared filesystem. Second, Kata containers utilize virtio-fs that involves an asynchronous event notification for every file I/O request.

The contributions of this paper are as follows.

- We analyze the detailed file I/O operations of three secure container runtimes and describe the entire file I/O stack of the runtimes thoroughly.
- We present the experimental results on both file I/O performance and CPU usage, which demonstrate the performance and efficiency of the runtimes.
- We offer symbol-level profiling results that point out the root cause of the differences in performance and CPU usage, which can suggest research directions for improving the file I/O performance of the runtimes.

The remainder of this paper is organized as follows. First, we explain the related work (Section 2) and the foundation technologies for secure container runtimes (Section 3). Second, we discuss the details of the file I/O stack in each container runtime (Section 4). We then evaluate the file I/O performance and present the characteristics of the CPU usage (Section 5). Next, we analyze the bottlenecks of the file I/O stack using symbol-level profiling (Section 6). Finally, we discuss future works to further improve this paper (Section 7) and summarize the key findings of this paper (Section 8).

2. Related Work

2.1. Performance Comparison and Analysis

Recent years have seen the introduction of secure container runtimes, leading to numerous studies assessing their performance across various metrics, like startup time, CPU, memory, and I/O. Previous research [9,18] introduces gVisor and Firecracker, comparing their performance with existing container runtimes. Similarly, Kumar and Thangaraju [24] and Randazzo and Tinnirello [11] delve into Kata containers' performance via qualitative and quantitative analyses. Debab and Hidouci [20] explore the runc, gVisor, Kata containers, and Firecracker performance alongside other runtimes. Anjali et al. [19] conduct an architecture analysis comparing native Linux, gVisor, and Firecracker, finding differences in the kernel code execution and network latency. Viktorsson et al. [25] demonstrate deployment and execution speed differences between gVisor and Kata containers, highlighting their security strengths but with significant performance trade-offs.

Additionally, studies by the authors of [26,27] investigate gVisor's and Firecracker's performance with other serverless computing platforms during concurrent execution, emphasizing the runtime architecture and scheduling approaches as key performance determinants. Another study [26] shows Firecracker's superiority over gVisor in concurrent execution.

While prior research offers quantitative performance measures, it often lacks an in-depth analysis explaining runtime performance disparities. This paper fills this gap, presenting a sophisticated performance analysis focusing on file I/O performance and CPU usage. Moreover, it provides a detailed, symbol-level analysis of the file I/O stack, elucidating the root causes of performance differences and CPU usage among representative secure container runtimes.

2.2. Runtime Optimization

Secure container runtimes, due to additional security layers, incur performance degradation, prompting various efforts to enhance metrics, like the startup time, networking, parallel execution, and storage interfaces. Cadden et al. [28] showcase snapshot and restore techniques achieving millisecond-level cold startup times, significantly reducing deployment times and enabling substantial in-memory caching. Ustiugov et al. [29] tackle long cold startup times by addressing frequent page faults, proposing proactive instance memory page prefetching, resulting in a 3.7x reduction in startup times.

Thomas et al. [30] propose Particle, an optimized networking architecture reducing the network provisioning time by utilizing an ephemerally generated pool of network addresses. Conversely, Dukic et al. [31] leverage workload parallelism to reduce memory consumption per invocation without performance degradation and decrease the total memory utilization and cold starts.

Koller and Williams [32] propose a storage interface addressing the gap between the host and guest, employing read-only data files and a read–write block device, requiring host file mapping and metadata preparation for a copy-on-write filesystem. While orthogonal to runtime optimization studies, their detailed file I/O stack analysis can guide future efforts aiming to enhance file I/O performance or reduce the CPU usage in file I/O processing.

3. Background

Secure container runtimes adopt a *secure* layer to enforce security over the execution environments in which the applications run. This section explains the key technologies for the layer: the kernel-based virtual machine (KVM) and Linux secure computing mode (seccomp).

3.1. Kernel-Based Virtual Machine with Intel Virtualization Technology

The kernel-based virtual machine (KVM) is a virtualization platform that utilizes a Linux kernel as a hypervisor (e.g., Xen in other virtualization platforms). KVM runs as a loadable kernel module (i.e., kvm.ko) in the Linux kernel (i.e., host OS) and constructs virtualized execution environments (i.e., VMs) by exploiting the hardware-assisted virtualization techniques, such as Intel virtualization technology (VT) [33]. Intel VT offers two different execution modes, the root mode and non-root mode: the host OS runs in the root mode and VMs run in the non-root mode [34]. The mode transition that switches the execution mode of the CPU from the root mode to the non-root mode is called VMentry, which occurs when the instructions of VMs are executed. On the other hand, when the applications in VMs execute privileged instructions, the execution mode is switched from the non-root mode to the root mode, which is called VMexit. This is because the privileged instructions cannot be handled in the VM. For example, when a hardware interrupt occurs, it triggers VMexit to execute the corresponding interrupt handler in the host OS [35]. Upon every mode transition, the state information (e.g., the values of the registers and memory address) of the current execution mode is stored in the specialized data structure, the virtual machine control structure (VMCS), so that the states are restored later.

In addition, Intel VT provides an extended page table (EPT) [36] that enables the translation from the virtual address of a VM into the physical address of the host server. MMU with EPT offers two page tables: the guest page table for the VM and the host page table for the host OS. For each virtual memory accessed by the VM, MMU retrieves the physical address of the host server from the virtual address of the VM using the guest and host page tables. The EPT translation requires two page table walks so that inherently it is slower than native (without KVM) address translation. Moreover, when the memory access made by a VM is not permitted, it causes an EPT violation that is a page fault, which needs extra processing to update the page tables [37]. KVM typically utilizes QEMU to support virtualization together. QEMU is a fast machine emulator using a dynamic translator. QEMU supports full system emulation, where a complete and unmodified kernel runs. Also, QEMU provides the CPU and device emulation, memory management, and I/O for VMs.

3.2. Linux Secure Computing Mode (Seccomp)

To enhance the security, the Linux kernel offers several functionalities, such as Netfilter and Linux Security Modules. The Linux secure computing mode (seccomp) is one of them, which offers secure state management in system call processing. Seccomp prevents applications from executing unauthorized system calls in order to avoid negative impacts (e.g., a bug or malicious attack) on the host OS. If an application attempts to execute the unauthorized system call, the Linux kernel immediately terminates the application using SIGKILL or SIGSYS.

Seccomp offers STRICT and FILTER modes, and users can opt for one of them to utilize the seccomp functionalities. First, the STRICT mode allows applications to execute only four system calls, exit(), sigreturn(), and read() and write(), to already-open file descriptors. Second, the FILTER mode enables users to specify the available system calls for their applications depending on their purposes. This can be performed by initializing a system call filter that consists of available system calls and registering the filter to seccomp using Linux kernel APIs. For example, gVisor enables the applications in the container to execute 211 system calls out of 319 Linux system calls.

4. File Operations in Secure Container Runtimes

Secure container runtimes adopt additional layers such as VMs and user-space kernels to enforce isolation between the containers. The commonality among three representative secure runtimes lies in their use of KVM-based VMM technology. However, they implement different architectures and techniques to offer isolated execution environments. From an architectural standpoint, Kata containers leverage containers on VMs, gVisor utilizes a user-space kernel, and Firecracker employs VMM-based microVMs. Regarding I/O processing, Kata containers utilize virtio-fs, gVisor employs system call filtering, and Firecracker utilizes virtio-blk. Therefore, they have different file I/O stacks depending on the architectures. This section describes the file I/O stacks of the Kata containers, gVisor, and Firecracker. This paper presents a deep-dive analysis of the file I/O stack at the symbol level, illustrating how the host OS manages file I/O requests from applications. It is worth noting that this file I/O processing differs from that in the existing native environments and has not been explored in previous studies.

Specifically, we categorize the symbols related to file I/O processing by using the functionalities: mode transition, handling EPT, memory processing, filesystem processing, scheduling, and application.

- Handling EPT: Operations that deal with EPT violation and the reconstruction of the EPT.

- Memory processing: Miscellaneous memory operations such as memory mapping and copy, except the *handling EPT*.
- Filesystem processing: The file I/O processing operation conducted in the host OS (e.g., vfs_read).
- Scheduling: Symbols related to the process scheduling.
- Application: Operations executed by the application running on containers or VMs.

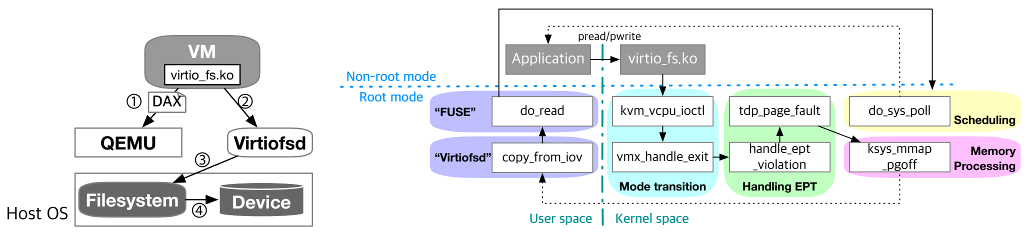
4.1. Kata Containers

First, Kata containers utilize *virtio-fs* [14] as illustrated in Figure 2a. Virtio-fs consists of virtio_fs.ko in a VM and *virtiofsd* on a host OS. This allows VMs to access a directory tree on the host OS, which means the host and the VM share the filesystem. Although the default filesystem is 9 pfs, Kata containers have supported virtio-fs since version 1.7. This paper focuses on Kata containers with virtio-fs because virtio-fs provides a significant performance improvement compared to 9 pfs [38].

Figure 2a shows the steps of the file operations. First, the Kata runtime runs the application on a container deployed in a VM that consists of an independent kernel image and root filesystem as in Figure 1. The application submits I/O requests directly to the filesystem (i.e., virtio_fs.ko) using Direct Access (DAX) [39] (①). DAX enables the application to read or write data to the virtio-fs using the memory mapping offered by the memory management of QEMU. ② Then, virtiofsd, a filesystem daemon running on the host OS, performs the I/O requests to the actual block device on behalf of virtio-fs in the VM. ③ The requests are handled as FUSE (FUSE (filesystem in user space) is the most widely used user-space filesystem that offers simple APIs and high performance) [40] requests asynchronously because virtio-fs utilizes the FUSE library for communication between the VM and the host OS. As a result, ④ they are finally delivered to the storage device of the host OS.

Figure 2b depicts the call flow of the file I/O stack of Kata containers. The file I/O stack can be divided into categories depending on their functionalities as described above. When Kata containers execute the file I/O system calls such as `pread` or `pwrite`, the virtio-fs driver in the VM submits the file I/O requests to the virtiofsd. This brings VMexit, where the execution mode is switched to the root mode [41]. Upon every VMexit, `kvm_vcp_ioctl` and `vmx_handle_exit` are executed to check the reason of the VMexit (i.e., the type of the privileged instruction that causes the VMexit). This includes handling EPT violations, such as `handle_ept_violation` and `tdp_page_fault`. An EPT violation is caused by a page fault in the EPT and requires the reconstruction of the EPT entries. After reconstructing the EPT entries, memory mapping is performed in `ksys_mmap_pgoff` and the execution mode can be returned to the VM after calling `do_sys_poll`.

In Kata containers, the actual filesystem processing is conducted by virtiofsd asynchronously. Virtiofsd copies the file I/O requests of the VM and executes the proper fuse filesystem requests. Note that the symbols related to virtiofsd belong to the source code of QEMU rather than the Linux kernel and run in the user space.



(a) (b)

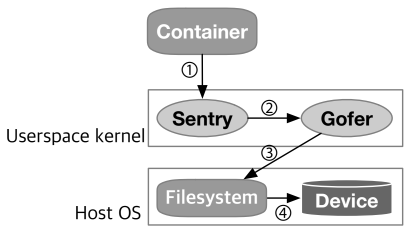
Figure 2. File operations of Kata containers. (**a**) Overview, and (**b**) symbol-level analysis of the file I/O stack.

4.2. gVisor

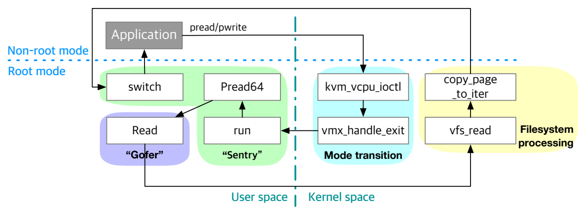
gVisor offers the virtualized environment to sandbox containers and filters system calls by utilizing the user-space kernel. To construct the virtualized environment, gVisor supports ptrace [42] and KVM [43]. Even though ptrace is offered by default, the KVM-based virtualization environments provide better performance than ptrace because of the hardware acceleration [18]. Thus, this paper focuses on gVisor with KVM to eliminate the performance bottlenecks in the system call filtering with ptrace when analyzing the bottlenecks in the file I/O stack itself.

In gVisor, the filesystem is located in the host OS, and the containers can only access the filesystem through the user-space kernel that consists of Sentry and Gofer. Different from Kata containers, gVisor does not share filesystems so that the containers cannot execute filesystem-related system calls directly. As shown in Figure 3a, ① Sentry captures the system calls from the containers. When the system call is related to the filesystem, ② Gofer receives the request and ③ executes the corresponding file I/O function, such as `vfs_read`. ④ This leads to the actual file I/O processing that works on the storage device of the host.

Figure 3b shows the call flow of the filesystem-related system calls in gVisor. First, when applications execute `pread` or `pwrite`, it involves VMexit from the user space of the non-root mode to the kernel space of the root mode, which is similar to Kata containers. However, the VMexit does not require handling an EPT violation because the application running on the container of gVisor cannot perform file I/O processing directly. Instead of handling the EPT violation, after the VMexit, Sentry receives the file I/O requests and calls the corresponding system call such as `pread64`, and in turn, Gofer initiates the file I/O system call.



(**a**)



(**b**)

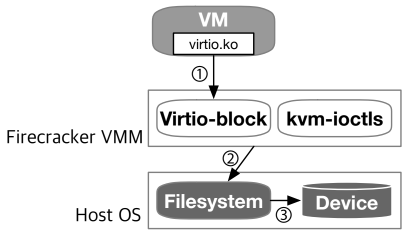
Figure 3. File operations of gVisor. (**a**) Overview, and (**b**) symbol-level analysis of the file I/O stack.

4.3. Firecracker

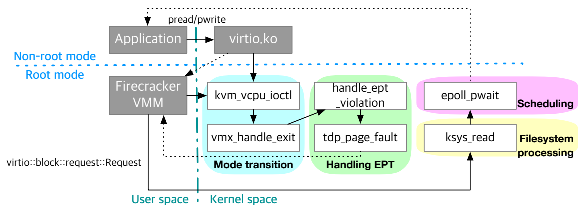
Firecracker offers a file I/O stack using virtio block devices [15], an open API for emulated devices. The virtio block device operates in a pair: the virtio front-end driver and virtio back-end driver. The virtio front-end driver is located in a VM and sends/receives I/O requests to/from the host OS. The virtio back-end driver is typically located in the host OS, and it performs read/write operations on the actual storage devices. Instead, Firecracker developed a new VMM (i.e., Firecracker VMM) that utilizes the virtio back-end driver to process I/O requests from VMs.

As illustrated in Figure 4a, ① when the application of the VM submits I/O requests, the virtio block in the VMM receives the requests asynchronously through the virtio front-end driver (virtio.ko) of the VM. ② To handle the requests, the virtio block executes the file I/O functions of the host OS. For example, when the application running on a VM executes `pread` in ①, this is delivered to the virtio block through virtio::block::request::Request that directly leads to vfs_read that processes the `pread` request in the host OS (③).

Figure 4b shows the call flow of Firecracker's file I/O stack. When an application in the VM executes a file I/O system call such as `pread` or `pwrite`, the file I/O requests are executed in the virtio front-end driver of the VM. As the virtio front-end driver is exposed as a block device in the VM, the application reads or writes directly to the filesystem of the VM. Then, the virtio front-end driver (virtio.ko) in the VM submits the file I/O requests to the virtio block of the Firecracker VMM, which causes VMexit. In order to identify the cause of VMexit, *kvm_vcp_ioctl* and *vmx_handle_exit* are executed. Then, *handle_ept_violation* and *tdp_page_fault* are executed to handle the EPT violation. Because the application performs read and write operations on the block device inside the VM, memory should be allocated in the VM. At this time, page faults occur and are handled by using *handle_ept_violation* and *tdp_page_fault*. The Firecracker VMM receives the file I/O requests asynchronously from the VM and then calls virtio::block::request::Request in order to process the requests in the host OS. Then, *ksys_read/write* is executed to process the filesystem operations. When the filesystem processing is completed, *epoll_pwait* is called.



(**a**)



(**b**)

Figure 4. File operations of Firecracker. (**a**) Overview, and (**b**) symbol-level analysis of the file I/O stack.

5. Performance and CPU Usage Analysis

Now, we evaluate the file I/O performance of the secure container runtimes and compare the results with the most popular container runtime, runc. In particular, we measure the CPU usage when each runtime executes the file I/O benchmark, which indicates the efficiency of the runtime. In addition, we measure the effects of the file operations and block size in order to evaluate their impact on performance and CPU usage.

5.1. Experimental Setup

For the experiments, we utilize a host server equipped with an Intel Xeon processor (E5-2650 v3@2.30 GHz, 10 cores), 256 GB RAM, and Intel 400 GB PCIe 3.0 x4 NVMe SSD as a local disk, running Ubuntu 18.04 LTS (Linux kernel v5.4.0). Table 2 lists the specifications of the host server and the versions of the runtimes. We construct a single container (and also a VM) running Ubuntu 18.04 LTS on each runtime and configure it to have a single virtual CPU (vCPU). Note that we do not limit the memory size of the container (and the VM) to eliminate performance bottlenecks in the memory. The host server is based on an ext4 filesystem, while the container (runc and gVisor) or the VM (Kata containers and Firecracker) in each runtime utilizes the following filesystems: overlayfs (runc), ext4 (gVisor and Firecracker), and virtio-fs (Kata containers).

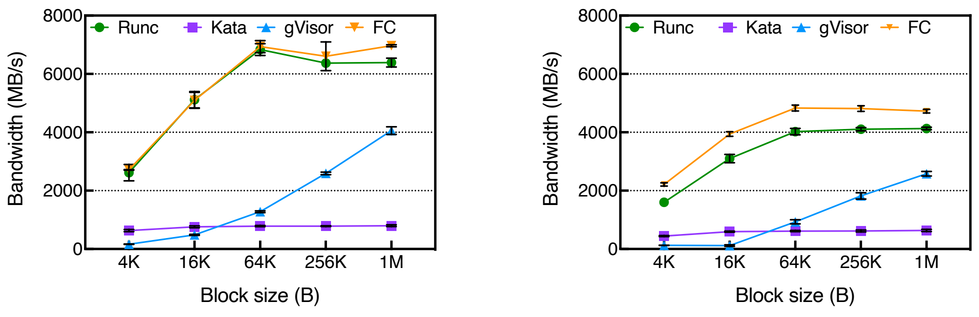
We measure the I/O bandwidth and CPU usage with different block sizes (i.e., 4 KB, 16 KB, 64 KB, 256 KB, and 1 MB). First, for the I/O bandwidth, we utilize *fio* [44], a standard Linux I/O benchmark that performs read/write operations of various sizes. We conduct four different operations (i.e., sequential read/write and random read/write) on a 1 GB file. We run each operation for 10 s and perform three iterations of each experiment. To eliminate the effect of the page cache on the I/O bandwidth, we restart the container and VM before each iteration, and the file is flushed to the disk so that the page cache is cleared between each iteration. Also, for CPU usage, we utilize *mpstat* [45], a standard Linux CPU profiling tool, which monitors the number of CPU cycles spent for the following categories: User, System, Iowait, Softirq, and Guest. User, System, and Guest indicate the percentage of CPU cycles spent in the user level, the system level, and the VM level, respectively. Iowait means the percentage of CPU cycles that are idle while the system has outstanding disk I/O requests, while Softirq indicates the CPU consumption for processing software interrupts.

Table 2. Experimental configuration.

Component	Configuration	Component	Configuration
Processor	Intel E5-2650 v3@2.30 GHz (10 cores)	Container/VM OS	Ubuntu 18.04 LTS
RAM Storage	256 GB Intel 400 GB PCIe 3.0 x4 NVMe SSD	Docker container Kata containers	v19.03.6 1.12.0-alpha0
Operating system	Ubuntu 18.04 LTS	gVisor	release-20201208.0
Linux kernel	v5.4.0	Firecracker	v0.21.0

5.2. Sequential Read/Write

Figure 5 shows the results of the sequential read and write performance (y-axis) with different block sizes (x-axis). Among the three runtimes, Firecracker achieves the best performance for both the read and write operations. For read, Firecracker shows an even higher performance than runc by 3.6% on average. For write, Firecracker achieves a higher performance than runc consistently by 21.0% on average. This is due to the different filesystems in runc and Firecracker. runc adopts the union filesystem [46] that needs to go through the storage driver of the runtime (i.e., overlayfs) to access the host backing filesystem (i.e., ext4). On the other hand, Firecracker uses the virtio block device inside the VM, which allows applications to exploit the page cache in the VM after a file block is stored in the cache. For example, when a fio benchmark writes the 1 GB file at first, the corresponding blocks of the file are transferred to the VM through the virtio block in the host OS. After the first write operation, the file blocks are stored in the page cache of the VM, and the fio benchmark can write from the page cache without asking the virtio block in the host OS. As a result, Firecracker achieves superior file I/O performance compared to the other runtimes, even better than runc.



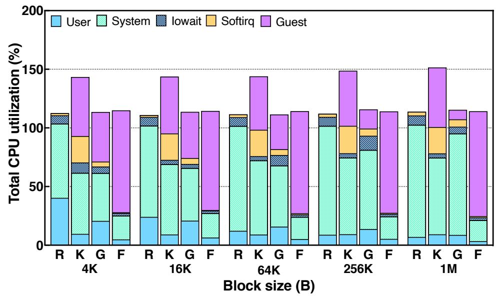
(**a**) (**b**)

Figure 5. Sequential file I/O performance of runc, Kata containers (Kata), gVisor, and Firecracker (FC) with different block sizes. (**a**) Sequential read, and (**b**) sequential write.

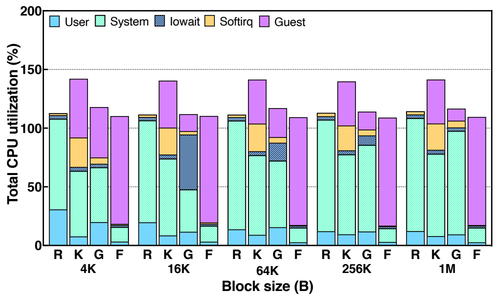
Kata containers have the lowest performance regardless of read and write and the block size. For example, Kata containers achieve a bandwidth ranging from 581 to 752 MB/s on average, which is only 13.7% of the Firecracker performance. The reason for the low performance is that Kata containers use the shared filesystem that requires a request transfer between the VM and virtiofsd for every file I/O operation in an asynchronous manner. *virtiofsd* runs on the host OS and communicates with the VM by using socket data. This is very different from Firecracker, which can allow applications to utilize the page cache inside its VM.

The performance of gVisor increases when the block size increases from 4 KB to 1 MB. For the 1 MB blocks, gVisor achieves 4058 MB/s for read and 2578 MB/s for write. This shows that gVisor scales well with the block size, which is consistent with the results of previous research [18]. The lack of an increase in the sequential write performance from 4K to 16K in Figure 5b is due to CPU saturation. However, although the performance of gVisor increases with the block size, gVisor still shows quite a lower performance than Firecracker by 69.7% and 72.8% for read and write, respectively. This is because the user-space kernel of gVisor requires mode transition and system call filtering for every file I/O request, which is different from Firecracker that can eliminate the additional mode transition by utilizing the page cache inside the VM.

In terms of CPU usage, Figure 6a,b illustrate that Kata containers consume the largest CPU resources (143.7% on average) for both read and write and for all block sizes. This result explains the low performance of the Kata containers. Through profiling, we find that virtiofsd of the Kata containers performs a memory copy frequently to deliver the requests of the VM to the filesystem of the host OS and executes file I/O operations, so it burns the CPU heavily in the user and system categories as shown in Figure 6a,b.



(**a**)



(**b**)

Figure 6. CPU usage in processing sequential file I/O operations under runc (R), Kata containers (K), gVisor (G), and Firecracker (F). (**a**) Sequential read, and (**b**) sequential write.

Firecracker and gVisor utilize 112.8% of the CPU on average, which is similar to the usage of runc (112.4% on average). However, the categories that consume most of the CPU resources are different between them due to their architectural differences. Firecracker mostly consumes the CPU resources in Guest (i.e., 87.6% for read and 92.2% for write), which means that most of the file operations are conducted in the VM. On the other hand, in gVisor, the CPU spent in Guest (i.e., the container in gVisor) is smaller than Firecracker, which is 27.2% for read and 21.5% for write. This is because the container in gVisor does not perform file I/O requests, but Gofer executes the filesystem-related system call that conducts actual memory copy and file I/O processing, which is counted in the System category. As a result, in gVisor, the CPU spent in System increases as the block size increases.

5.3. Random Read/Write

Figure 7 depicts the performance for random read and random write operations. First, for random read, runc shows the highest performance (2458 MB/s on average), while the Kata containers have the lowest performance (422 MB/s on average) on average. Also, the performance of all runtimes increases as the block size increases. Similar to sequential read, Firecracker achieves a higher performance than gVisor and Kata containers. For the 1 MB blocks, Firecracker shows the highest performance (4749 MB/s on average), which is 4.3 times higher than the Kata containers (1113 MB/s on average) and 26.1% higher than gVisor (3766 MB/s on average). Regarding the performance for random write, Firecracker achieves the highest performance (3842 MB/s on average), while the Kata containers show the lowest performance (302 MB/s on average) independent of the block size. For the 1 MB blocks, Firecracker reaches 4671 MB/s, which is 5.7 times higher than the Kata containers (815 MB/s on average).

Also, we measure the CPU usage of the runtimes for the random read and write as in Figure 8. It shows that there is the significant Iowait for random read with small blocks such as 4 KB and 16 KB, which explains the low performance of Figure 7a for the small blocks. Also, comparing with the sequential read in Figure 6a, the random read in Figure 8a has a larger Iowait for all runtimes: For the sequential and random read, this is 7.4% and

54.0% (runc), 4.6% and 18.0% (Kata containers), 7.1% and 32.6% (gVisor), and 2.0% and 19.9% (Firecracker) on average. This indicates that there is a performance bottleneck in processing random read requests. Figure 8 also shows that the random read has a higher Iowait than random write: For read and write, this is 54.0% and 3.0% (runc), 18.0% and 14.6% (Kata containers), 32.6% and 15.5% (gVisor), and 19.9% and 1.4% (Firecracker) on average. This is because the read and write operations have different characteristics: read operations are performed synchronously, while write operations are asynchronous. Such differences render random write to have less Iowait CPU usage than random read.

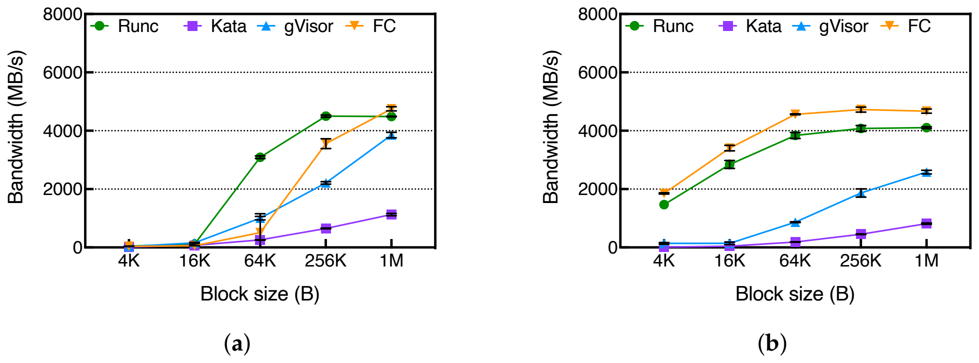


Figure 7. Random file I/O performance of runc, Kata containers (Kata), gVisor, and Firecracker (FC) with different block sizes. (**a**) Random read, and (**b**) random write.

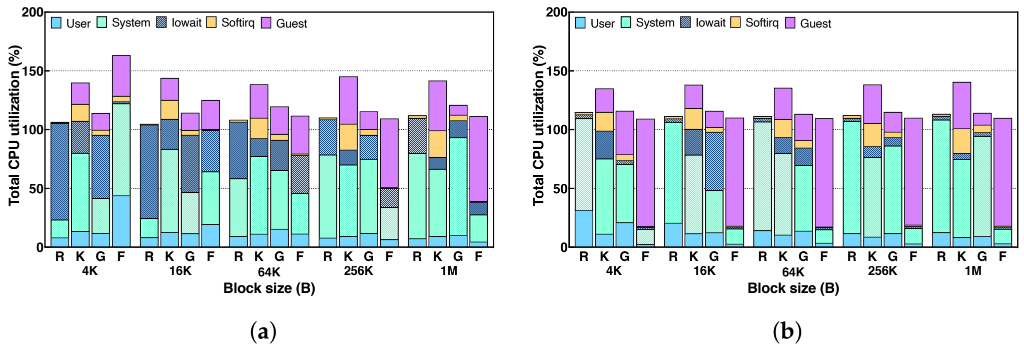


Figure 8. CPU usage in processing random file I/O operations under runc (R), Kata containers (K), gVisor (G), and Firecracker (F). (**a**) Random read, and (**b**) random write.

6. Bottleneck Analysis Using Symbol-Level Profiling

To further investigate the file I/O performance, we profile the operations and categorize the data based on the symbols presented in Section 4. We utilize the Linux profiling tool, perf, to gather the performance counter statistics for the symbols. Note that we run perf for sequential read and write operations because random read and write have significant Iowait CPU usage, which means that the experiment settings are not optimized for random read and write.

6.1. Kata Containers

Figure 9 illustrates the CPU cycles that each symbol occupies for the read and write operations with 4 KB and 1 MB blocks. Note that the y-axis is the normalized CPU cycles (over the total CPU cycles of an operation on the x-axis, such as read with 4 KB). First, *handling EPT* shows the highest overhead regardless of the block size and the type of operations, and it occupies more than 26.8% in all cases. The profiling results show that *handling EPT* involves a large number of page table walks [37]. In particular, the overhead of *handling EPT* increases in the write operation because the I/O requests should be stored in the memory space of the VM and delivered to the host OS.

Second, the overhead for *mode transition* increases in the operations with small blocks (i.e., 4 KB) because small blocks have more I/O request rates than large blocks. In addition, *memory processing* increases when the block size is small (4 KB). This is because Kata containers require memory mapping for every file I/O request to deliver the requests to the host OS, which increases the amount of memory (un-)mapping and (de-)allocation. In addition, *memory processing* includes virtiofsd clearing the dirty flag of the memory used. These results show that the architecture of Kata containers incurs additional memory overhead, such as reconstructing EPT entries, frequent memory mapping, and clearing flags. Note that the overhead of *memory processing* decreases with the 1 MB blocks because the number of memory (un-)mapping operations decreases with large blocks.

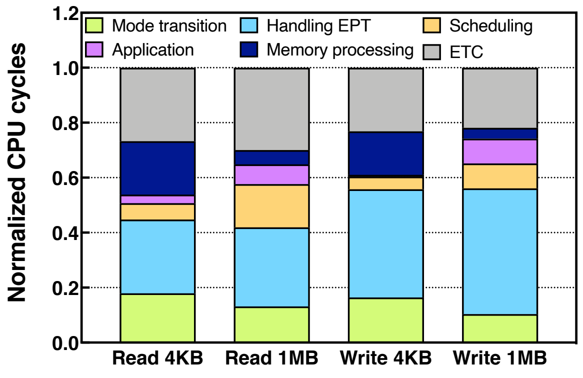


Figure 9. Symbol-level profiling of I/O processing in Kata containers.

6.2. *gVisor*

In gVisor, the file I/O requests from the containers are not directly applied to the filesystem, but they are filtered by Sentry. If the requests are examined as valid, Gofer executes the corresponding filesystem API, such as `vfs_read`. This is different from the Kata containers that allow the containers to write the file I/O requests to the memory space allocated to the VMs. Because the containers in gVisor cannot access the memory space directly, they do not incur the overhead of *handling EPT*. Instead, Figure 10 shows that *mode transition* and *filesystem processing* have a major impact on the file I/O performance in gVisor. Specifically, *filesystem processing* increases when the block size increases from 4 KB to 1 MB for both the read and write operations. This is because the I/O bandwidth and CPU usage in System increases with the increase in block sizes as depicted in Figures 5 and 6.

In addition, *mode transition* decreases when the block size increases for both read and write. The reason is that the number of system calls decreases when the block size increases from 4 KB to 1 MB, which causes the mode transition to decrease as well. Moreover, for the 4 KB block, the overhead in Application is 8.6% for read and 7.7% for write, and for 1 MB, Application becomes negligible for both read and write (1.2% and 1.5%, respectively).

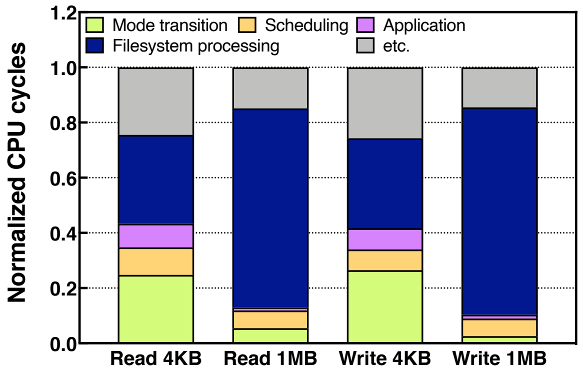


Figure 10. Symbol-level profiling of I/O processing in gVisor.

6.3. Firecracker

Figure 11 shows that for read, *filesystem processing* occupies the largest CPU cycles at more than 41.3%. *filesystem processing* indicates file I/O processing in the host OS, which delivers the file I/O requests between the storage devices in the host server and the VM. Note that this is different from the results in Figure 6a, which shows that the largest fraction of the CPU usage is spent in Guest. The reason for this difference is that we conduct the profiling on the host OS rather than on the VM. Because our profiling result only contains filesystem-related functions executed on the host OS, this leads to amplifying the overhead of *filesystem processing*. We plan to conduct additional profiling to include the overhead caused by the filesystem-related functions executed in the VM.

For write, *handling EPT* is the symbol that occupies more than 38.5% of the CPU cycles. This is because the I/O requests should be stored in the virtio block device in the VM. This makes the write performance lower than the read (as shown in Figure 5b). The mode transition is similar regardless of the type of operations or block sizes, which is distinct from the other runtimes. This is because Firecracker prevents VMs and its VMM from interfering with each other, which reduces the mode transition. In other words, the architecture of Firecracker leads to a superior file I/O performance over other runtimes.

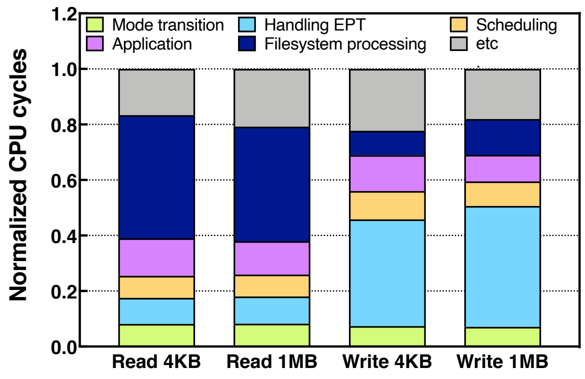


Figure 11. Symbol-level profiling of I/O processing in Firecracker.

7. Discussion

This paper focuses on evaluating the file I/O performance of secure container runtimes. Our evaluation results show that Firecracker offers the highest file I/O performance for all types of operations with high efficiency compared to the other secure container runtimes. Firecracker outperforms the other runtimes in read and write operations by leveraging the virtio block device within the VM for efficient page caching, enhancing the overall performance. In contrast, Kata containers' reliance on a shared filesystem leads to asynchronous request transfers between the VM and virtio-fs, impacting performance. gVisor exhibits significantly lower performance than Firecracker due to its user-space kernel, which necessitates mode transition and system call filtering for each file I/O request, unlike Firecracker's streamlined utilization of the VM's page cache.

When we determine which runtime to choose for constructing container environments, the security aspect of the runtime should be considered also. As each runtime adopts a different architecture (e.g., the VM layer or user-space kernel) to enhance the security of the containers, the security characteristics can be different. The threat models can be categorized into three cases: (1) protection between a container and an application running in the container, (2) inter-container protection, and (3) protection between containers and the host OS as described in the previous study [47]. The security assessment of each runtime depends on the threat cases. For example, to protect the host OS from malicious containers, Firecracker and Kata containers may be better choices than gVisor. This is because the containers in gVisor execute more system calls (74) to the host OS than those of Firecracker (37) and Kata containers (none) [48,49]. Moreover, Firecracker and Kata containers assign an independent VM to each container, so most of the system calls are handled inside the VM. Therefore, they do not affect the host OS.

There have been attempts to assess the security of container runtimes [47,49]. However, previous studies only provide results with limited aspects, such as the evaluation of a single container runtime (i.e., runc) or focusing on a system call-level security assessment. Hence, it is necessary to evaluate in detail the security aspects of secure container runtimes, similar to what we perform for file I/O performance. In future work, we plan to perform security assessments for secure container runtimes and find the trade-off between security and overall performance.

8. Concluding Remarks

This paper analyzes the file I/O stack of the representative secure container runtimes, Kata containers, gVisor, and Firecracker. We evaluate the runtimes in terms of file I/O performance and CPU usage and compare the results to the popular container runtime, runc. This paper points out that each runtime adopts different file I/O mechanisms, such as virtio-fs and virtio block. Also, we present the architectural comparison that leads to the significant differences in the file I/O performance and CPU usage of the runtimes. The key findings of this paper are three-fold. First, Firecracker shows the highest file I/O performance because it can allow applications to exploit the page cache inside the corresponding VMs. Second, Kata containers offer the lowest file I/O performance while consuming the largest amount of CPU resources. We find that the root cause of the inefficiency lies in their file I/O stack based on the asynchronous event notification between a VM and virtiofsd. Last, we observe that gVisor scales well as the block size increases. The reason is that the file I/O requests are mainly handled by the host OS so that gVisor scales as the host OS.

Author Contributions: Conceptualization, K.L. and J.K.; methodology, I.-H.K.; software, K.L. and J.K.; validation, I.-H.K., H.P. and C.-H.H.; formal analysis, K.L.; investigation, I.-H.K., H.P. and C.-H.H.; resources, C.-H.H.; data curation, K.L.; writing—original draft preparation, K.L., J.K. and I.-H.K.; writing—review and editing, H.P. and C.-H.H.; visualization, K.L.; supervision, H.P. and C.-H.H.; project administration, C.-H.H.; funding acquisition, C.-H.H. All authors have read and agreed to the published version of the manuscript.

Funding: This paper was supported by Korea Institute for Advancement of Technology (KIAT) grant funded by the Korea Government (MOTIE) (P0017011, HRD Program for Industrial Innovation). This research was supported by the Chung-Ang University Research Scholarship Grants in 2021.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on request from the corresponding author. The data are not publicly available due to privacy.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Soltesz, S.; Pötzl, H.; Fiuczynski, M.E.; Bavier, A.; Peterson, L. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, Lisbon, Portugal, 21–23 March 2007; pp. 275–287.
2. Merkel, D. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.* **2014**, *2014*, 2.
3. Felter, W.; Ferreira, A.; Rajamony, R.; Rubio, J. An updated performance comparison of virtual machines and linux containers. In Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Philadelphia, PA, USA, 29–31 March 2015; pp. 171–172.
4. Li, Z.; Cheng, J.; Chen, Q.; Guan, E.; Bian, Z.; Tao, Y.; Zha, B.; Wang, Q.; Han, W.; Guo, M. RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing. In Proceedings of the 2022 USENIX Annual Technical Conference (USENIX ATC 22), Carlsbad, CA, USA, 11–13 July 2022; pp. 53–68.
5. Hong, C.H.; Varghese, B. Resource management in fog/edge computing: A survey on architectures, infrastructure, and algorithms. *ACM Comput. Surv. (CSUR)* **2019**, *52*, 1–37. [CrossRef]

6. National Vulnerability Database. CVE-2019-5736 Detail. 2019. Available online: https://nvd.nist.gov/vuln/detail/CVE-2019-5736 (accessed on 14 April 2021).
7. Walsh, D.J. Are Docker Containers Really Secure? 2014. Available online: https://opensource.com/business/14/7/docker-security-selinux (accessed on 23 March 2021).
8. Sultan, S.; Ahmad, I.; Dimitriou, T. Container security: Issues, challenges, and the road ahead. *IEEE Access* **2019**, *7*, 52976–52996. [CrossRef]
9. Agache, A.; Brooker, M.; Iordache, A.; Liguori, A.; Neugebauer, R.; Piwonka, P.; Popa, D.M. Firecracker: Lightweight virtualization for serverless applications. In Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), Santa Clara, CA, USA, 25–27 February 2020; pp. 419–434.
10. gVisor: Application Kernel for Containers. Available online: https://github.com/google/gvisor (accessed on 6 May 2021).
11. Randazzo, A.; Tinnirello, I. Kata containers: An emerging architecture for enabling mec services in fast and secure way. In Proceedings of the 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS), Granada, Spain, 22–25 October 2019; pp. 209–214.
12. Barr, J. Firecracker—Lightweight Virtualization for Serverless Computing. 2018. Available online: https://aws.amazon.com/ko/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/ (accessed on 14 April 2021).
13. Bellard, F. QEMU, a fast and portable dynamic translator. In Proceedings of the USENIX Annual Technical Conference, FREENIX Track, Anaheim, CA, USA, 10–15 April 2005; pp. 41–46.
14. virtio-fs. Available online: https://virtio-fs.gitlab.io/ (accessed on 25 May 2021).
15. Russell, R. virtio: Towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Oper. Syst. Rev.* **2008**, *42*, 95–103. [CrossRef]
16. Klimovic, A.; Wang, Y.; Stuedi, P.; Trivedi, A.; Pfefferle, J.; Kozyrakis, C. Pocket: Elastic ephemeral storage for serverless analytics. In Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), Carlsbad, CA, USA, 8–10 October 2018; pp. 427–444.
17. Kang, J.; Hu, C.; Wo, T.; Zhai, Y.; Zhang, B.; Huai, J. Multilanes: Providing virtualized storage for os-level virtualization on manycores. *ACM Trans. Storage (TOS)* **2016**, *12*, 1–31. [CrossRef]
18. Young, E.G.; Zhu, P.; Caraza-Harter, T.; Arpaci-Dusseau, A.C.; Arpaci-Dusseau, R.H. The true cost of containing: A gVisor case study. In Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19), Renton, WA, USA, 8 July 2019.
19. Anjali.; Caraza-Harter, T.; Swift, M.M. Blending containers and virtual machines: A study of firecracker and gVisor. In Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Lausanne, Switzerland, 17 March 2020; pp. 101–113.
20. Debab, R.; Hidouci, W.K. Containers Runtimes War: A Comparative Study. In *Future Technologies Conference (FTC) 2020, Proceedings of the Future Technologies Conference, Vancouver, BC, Canada, 5–6 November 2020*; Springer: Cham, Switzerland, 2020; pp. 135–161.
21. Goethals, T.; Sebrechts, M.; Al-Naday, M.; Volckaert, B.; De Turck, F. A functional and performance benchmark of lightweight virtualization platforms for edge computing. In Proceedings of the 2022 IEEE International Conference on Edge Computing and Communications (EDGE), Barcelona, Spain, 10–16 July 2022; pp. 60–68.
22. Armbrust, M.; Fox, A.; Griffith, R.; Joseph, A.D.; Katz, R.; Konwinski, A.; Lee, G.; Patterson, D.; Rabkin, A.; Stoica, I.; et al. A view of cloud computing. *Commun. ACM* **2010**, *53*, 50–58. [CrossRef]
23. Ngenzi, A.; R, S.; Nair, S.R. Dynamic Resource Management in Cloud Data Centers for Server Consolidation. *arXiv* **2015**, arXiv:1505.00577.
24. Kumar, R.; Thangaraju, B. Performance Analysis between runC and Kata Container Runtime. In Proceedings of the 2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT), Bangalore, India, 2–4 July 2020; pp. 1–4.
25. Viktorsson, W.; Klein, C.; Tordsson, J. Security-Performance Trade-offs of Kubernetes Container Runtimes. In Proceedings of the 2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Nice, France, 17–19 November 2020; pp. 1–4.
26. Park, J.; Kim, H.; Lee, K. Evaluating Concurrent Executions of Multiple Function-as-a-Service Runtimes with MicroVM. In Proceedings of the 2020 IEEE 13th International Conference on Cloud Computing (CLOUD), Beijing, China, 19–23 October 2020; pp. 532–536.
27. Barcelona-Pons, D.; García-López, P. Benchmarking Parallelism in FaaS Platforms. *arXiv* **2020**, arXiv:2010.15032.
28. Cadden, J.; Unger, T.; Awad, Y.; Dong, H.; Krieger, O.; Appavoo, J. SEUSS: Skip redundant paths to make serverless fast. In Proceedings of the Fifteenth European Conference on Computer Systems, Heraklion, Greece, 27–30 April 2020; pp. 1–15.
29. Ustiugov, D.; Petrov, P.; Kogias, M.; Bugnion, E.; Grot, B. Benchmarking, analysis, and optimization of serverless function snapshots. *arXiv* **2021**, arXiv:2101.09355.
30. Thomas, S.; Ao, L.; Voelker, G.M.; Porter, G. Particle: Ephemeral endpoints for serverless networking. In Proceedings of the 11th ACM Symposium on Cloud Computing, Virtual Event, 19–21 October 2020; pp. 16–29.
31. Dukic, V.; Bruno, R.; Singla, A.; Alonso, G. Photons: Lambdas on a diet. In Proceedings of the 11th ACM Symposium on Cloud Computing, Virtual Event, 19–21 October 2020; pp. 45–59.

32. Koller, R.; Williams, D. An ounce of prevention is worth a pound of cure: Ahead-of-time preparation for safe high-level container interfaces. In Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19), Renton, WA, USA, 8–9 July 2019.
33. Neiger, G.; Santoni, A.; Leung, F.; Rodgers, D.; Uhlig, R. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technol. J.* **2006**, *10*, 167–177. [CrossRef]
34. Uhlig, R.; Neiger, G.; Rodgers, D.; Santoni, A.L.; Martins, F.C.; Anderson, A.V.; Bennett, S.M.; Kagi, A.; Leung, F.H.; Smith, L. Intel virtualization technology. *Computer* **2005**, *38*, 48–56. [CrossRef]
35. Zhang, B.; Wang, X.; Lai, R.; Yang, L.; Wang, Z.; Luo, Y.; Li, X. Evaluating and optimizing I/O virtualization in kernel-based virtual machine (KVM). In *Network and Parallel Computing, Proceedings of the IFIP International Conference on Network and Parallel Computing, Zhengzhou, China, 13–15 September 2010*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 220–231.
36. Yang, S. Extending KVM with new Intel Virtualization technology. In Proceedings of the KVM Forum, Napa Valley, CA, USA, 11–13 June 2008.
37. Dong, Y.; Xue, M.; Zheng, X.; Wang, J.; Qi, Z.; Guan, H. Boosting GPU Virtualization Performance with Hybrid Shadow Page Tables. In Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC 15), Santa Clara, CA, USA, 8–10 July 2015; pp. 517–528.
38. Kunwar, B. Disk I/O Performance of Kata Containers. Available online: https://www.stackhpc.com/images/IO-Performance-of-Kata-Containers-TheNewStack.pdf (accessed on 12 May 2021).
39. Wilcox, M.; Zwisler, R. Linux DAX. Available online: https://www.kernel.org/doc/Documentation/filesystems/dax.txt (accessed on 8 April 2021).
40. Szeredi, M. FUSE: Filesystem in Userspace. 2010. Available online: http://fuse.sourceforge.net (accessed on 8 April 2021).
41. Kim, H.; Kim, S.; Jeong, J.; Lee, J. Virtual asymmetric multiprocessor for interactive performance of consolidated desktops. In *ACM SIGPLAN Notices, Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments—VEE'14, Salt Lake City, UT, USA, 1–2 March 2014*; ACM Press: New York, NY, USA, 2014; pp. 29–40. [CrossRef]
42. gVisor—Ptrace. Available online: https://gvisor.dev/docs/architecture_guide/platforms/#ptrace (accessed on 20 May 2021).
43. gVisor—KVM. Available online: https://gvisor.dev/docs/architecture_guide/platforms/#kvm. (accessed on 20 May 2021).
44. fio. Available online: https://linux.die.net/man/1/fio (accessed on 8 April 2021).
45. mpstat. Available online: https://linux.die.net/man/1/mpstat (accessed on 8 April 2021).
46. Brown, N. Overlay Filesystem. Available online: https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt (accessed on 6 May 2021).
47. Laurén, S.; Memarian, M.R.; Conti, M.; Leppänen, V. Analysis of security in modern container platforms. In *Research Advances in Cloud Computing*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 351–369.
48. Skarlatos, D.; Chen, Q.; Chen, J.; Xu, T.; Torrellas, J. Draco: Architectural and Operating System Support for System Call Security. In Proceedings of the 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Athens, Greece, 17–21 October 2020; pp. 42–57.
49. Yang, J.; Tak, B. Security Assessment Technique of a Container Runtime Using System Call Weights. *J. Korea Soc. Comput. Inf.* **2020**, *25*, 21–29.