A Study of Performance and Security Across the Virtualization Spectrum

Vincent van Rijn

Distributed Systems Group

TU Delft

A Study of Performance and Security Across the Virtualization Spectrum

by

Vincent van Rijn

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday February 22nd, 2021 at 11:00 AM.

Student number:	4961471	
Project duration:	May 1, 2020 – March 1, 2021	
Thesis committee:	Dr. J. S. Rellermeyer,	TU Delft, supervisor
	Dr. L. Chen,	TU Delft
	Dr.ir. S.E. Verwer,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Contents

1

Introduction

Virtualization is the fundamental technology that enabled the widespread adoption of cloud computing. No matter the size, physical location, or nature of any cloud, virtualization remains the cornerstone that cloud computing builds upon. With the ever-increasing pervasiveness of the cloud computing paradigm, strong isolation guarantees and low-performance overhead from isolation platforms are paramount. An ideal isolation platform offers both: an infallible isolation boundary while it retains a negligible performance overhead. In practice, this holy grail remains elusive, and real-world isolation platforms are to make complex trade-offs between the degree of isolation and performance overhead.

Particularly common in clouds is the use of virtual machines. These virtual machines are provisioned by the cloud provider and are generally difficult to distinguish from a physical machine from the perspective of the customer. This type of virtualization is purported to offer a thick isolation boundary, at the expense of a firm (although shrinking) performance overhead. A compelling alternative to virtual machines are container-based isolation platforms. Container-based isolation platforms are commonly regarded as a lightweight form of virtualization, characterized by high performance and the ability to be rapidly deployed. The general trend in industry, exacerbated by the advent of the microservices architectural pattern, has been increasingly receptive to this relatively new type of virtualization. As container technology matures, some of its limitations have become apparent as well. An particular limitation is the relatively thin isolation boundary between the container and host, exposing a large attack surface to potential adversaries.

The ever-present need for both high performance (i.e. low overhead) and a high degree of security in isolation platforms has resulted in a multitude of new isolation solutions, such as unikernels and secure containers. These new platforms take novel approaches to the existing trade-offs between isolation and overhead, and thus fall somewhere in-between or next to the existing isolation platforms of virtual machines and containers. In this thesis, we attempt to place each of these platforms on this emerging spectrum. This spectrum can be divided into multiple, orthogonal, other spectrums. For example, we can place the examined platforms into a spectrum of where the isolation boundary between host and guest is established. This spectrum can be constructed with relative ease, as careful examination of the architecture of the isolation platform is sufficient to yield an answer to this question.

1.1. Problem statement

Two core properties, the degree of isolation offered by a platform and the performance overhead incurred, are not trivially determined and warrant the conduct of a wide array of experiments. In this thesis, we attempt to measure those two essential properties, and place the various new isolation platforms and techniques into two spectrum ranges accordingly. Moreover, we provide answers to questions that are fundamental to the isolation and overhead trade-off through quantifiable measures. Besides assessing the performance of platforms through traditional benchmarking techniques, we also devise a new method to quantitatively measure the security, or isolation, of each platform through an extension of the Horizontal Attack Profiling (HAP) measure. Guiding our research throughout the thesis, we attempt to answer the following concrete questions **Research Questions (RQ)**:

> **RQ1**: Where do the new types of virtualization techniques position themselves on the spectrum of performance overhead incurred?
>
> **RQ2**: Is the degree of isolation offered proportional to the performance overhead imposed by an isolation platform?
>
> **RQ3**: Does the extended HAP metric accurately quantify the degree of isolation?

Finding answers to these questions may be beneficial in multiple ways. First, answering **RQ1** can serve as a guide for which particular isolation platforms would be well-suited under certain conditions. For example, in scenarios where performance is of utmost importance, it might be recommendable to use a platform that sits on the most low-overhead side of performance overhead spectrum. The second question, **RQ2**, addresses verification of a commonly held view on isolation platforms: whether a stronger isolation boundary by definition hinders performance. It indirectly also addresses the question of whether the aforementioned holy grail is merely extraordinarily elusive, or if its existence is simply impossible. Finally, **RQ3** attempts to build upon previous work of quantitatively assessing security, and deepens this research into a specific direction, potentially improving the quality of research into this specific field.

1.2. Approach and scope

Before we can answer any research questions, we select a subset of the numerous isolation platforms available today. One core property is that each platform should be open-source, so that we can adjust and thoroughly inspect each and every platform as desired. Even with this requirement in place, a vast landscape of various (and sometimes exotic) types of virtualization techniques remain. In order to effectively answer all of our research questions, we have decided on a relatively broad scope for this thesis. Naturally, this also limits the depth to which we can investigate the isolation platform. We have attempted to find a good middle-ground, at which we can both characterize where certain types of virtualization techniques tend to lie in the performance and security spectrum, while still retaining the ability to thoroughly inspect each platform.

For answering the first research question, **RQ1**, we perform extensive research along multiple axes. First of all, we investigate the different architectures of the various platforms, and try to find similarities and differences between these architectures. We conduct a large number of traditional performance benchmarks that stress the subsystems of the isolation platforms. Lastly, we propose and execute a new method to quantify the degree of isolation of each platform by looking at the interaction between the platform guest and the host that it is running on. This metric builds upon earlier work but also extends it by cross-linking it against historical software vulnerability databases (CVEs). We call this metric the extended Horizontal Attack Profile (HAP) metric. The reason why we

include the extended HAP metric is that it allows us to quantitively express the degree of isolation of the platforms, allowing for a more accurate inter-platform comparison.

The approach for answering **RQ2** relies on data that is also used for **RQ1**. In particular, it relies on the data obtained through the extended HAP metric. In essence, if we place each platform into a spectrum of degree of isolation, and the order is the same (or its inverse, depending on the axes) as that of the performance overhead spectrum, it indicates a clear correlation between the two. If the two spectrums do not align, this might be indicative of leeway in optimization along both spectrums (i.e. pareto efficiency has not yet been achieved), as the degree of isolation is not strictly proportional to the performance overhead.

Answering **RQ3** requires us to critically look at the obtained results of the extended HAP metric versus the plain HAP metric. Naturally, as the HAP metric is not an established metric, we attempt to critique this metric as well. As far as we are aware this metric is the only way to quantitatively measure security, and, as such, in order to be able to critique the HAP metric, we must compare it to qualitative research. To this end, we present qualitative research into the different types of virtualization techniques using historical vulnerability data, past research and architectural analysis into the isolation boundary placement.

All the experiments, both pertaining to the performance as well as the (extended) HAP, are conducted on a powerful machine with 64 CPU cores (2x AMD EPYC 7542 32-Core CPU) and 256Gb of DDR4 RAM, running Ubuntu version 20.04.

1.3. Thesis outline and key contributions

Chapter 2 provides information on cloud computing and hardware virtualization assumed background knowledge to understand the remainder of the thesis. Chapter 3 presents the examined isolation platforms, their main value propositions, as well as a discussion precisely why these and not other platforms are examined. Chapter 4 dives deep into the architecture of the various isolation platforms, and explores the employed isolation mechanisms. Chapter 5 showcases an extensive set of benchmarking results pertaining to all relevant subsystems of the isolation platforms, including both micro-benchmarks and real-world benchmarks. Chapter 6 discusses and realizes a novel method to quantitatively measure the degree of isolation of the platforms. Finally, Chapter 7 presents the final conclusions of the thesis.

The key contributions of this thesis are the following:

1. Provide a comprehensive survey of open-source isolation platforms. These solutions include containers, hypervisors, unikernels, and everything in-between.

2. Provide an in-depth quantitative comparison of the performance of the platforms, along multiple axes. These comparisons consider computer subsystems (CPU, I/O, network, memory), real-world benchmarks and other related performance characteristics (start-up time). Analysis is provided to distinguish whether the performance limitations are inherent to the chosen isolation virtualization mechanism or due to the specific implementation. This is made possible by the wide scope of the chosen platforms.

3. Propose and perform a new method to quantitatively measure the security of the isolation platforms, through an extension of the Horizontal Attack Profile metric. A qualitative evaluation of this new method is also provided.

4. Provide qualitative survey-like research on the degree of isolation of the researched isolation platforms.

2

Background

This chapter discusses background topics that are relevant to the rest of the thesis. In Section 2.1 we briefly discuss the world of cloud computing as a whole, and its association with isolation platforms. In Section 2.2 we discuss the underlying techniques of one of the more complicated isolation platforms, hypervisors.

2.1. Cloud computing

Cloud computing makes it possible to buy computing resources as a utility, and has seen widespread adoption throughout industry in recent years. One of the core techniques that enable cloud computing is the use of isolation platforms. Instead of renting out compute capacity at the granularity physical machines, isolation platforms provide a way to host multiple tenants within one physical machine. This leads to a numerous benefits, for both customers as well as cloud platform providers:

- Customers only have to pay for the computing resources they need. For example, customers pay for the amount of storage by the day, and the number of virtual CPUs by the hour. These resources can easily be released when they are not necessary anymore, reducing costs.
- Customers can easily and quickly scale the rented resources up and down, fast enough to combat potential load spikes that would traditionally require ahead-of-time provisioning of resources (elasticity). The resources are rented from a pool that appears to be infinitely large.
- Customers can avoid making a large up-front commitment by not having to buy private computing resources, and can instead gradually increase the amount of rented computing resources.
- Cloud platform providers can make more efficient use of their hardware by collocating multiple customers on the same hardware.
- Cloud platform providers can use economy of scale to their advantage.

Overall, for the customer, this leads to a reduction in cost of IT deployment and operation. For cloud platform providers this also proves a profitable business. Naturally, there are downsides to the use of cloud computing as well, for both customer (e.g. vendor lock-in) as well as the provider (e.g. scalability with the number of tenants). There are technical challenges that arise with the use of the multi-tenancy nature of cloud platforms as well, as workloads of different customers are not separated by hardware boundaries anymore (as is the case with on-premise, private datacenters). Isolation platforms are a way to reestablish this separation. An isolation platforms should thus provide separation of resources provided to each tenant. This means that one tenant, a noisy neighbor,

should not be able to consume all physical resources that are available (leading to degradation of compute capacity of other tenants), but should instead be constrained by the isolation platform isolation boundary. An isolation platform should also provide an environment that is private from the other tenants.

These days, cloud computing platforms provide a vast amount of different services. These services differentiate themselves through the type of resources they offer (e.g. storage or compute capacity), but also in their programming, for example the serverless computing paradigm prescribes a programming model in which the user only writes a function, which is then executed on the cloud infrastructure. It is typically paid for on a per-function invocation. Another popular example is the use of container orchestration frameworks, which are well suited for leveraging the elasticity cloud platforms offer.

Different cloud services necessitate the use of different isolation platforms. In the aforementioned serverless computing paradigm, an isolation platform that offers extremely low startup times is preferable. If the cloud provider rents out parts of physical hardware as a virtualized machine, support for various operating systems might be deemed a requirement for the isolation platform. As such, there are various trade-offs the isolation platforms have to make, such on the performance, ease-of-use, and even security.

2.2. Hypervisor virtualization techniques

First, some words about the terminology used in the following paragraphs. The realm of hypervisor technology uses some distinct terminology that is important to clarify and disambiguate for the context of this thesis. We will refer to the piece of software that can create and run virtual machines as hypervisors. Sometimes, hypervisors are also referred to as Virtual Machine Monitors (VMM). Confusingly, the term VMM can also refer to only the user-space subset of the entire hypervisor functionality (and typically 'makes use of' the underlying hypervisor to manage VMs). For clarity, we will restrict ourselves to the word hypervisor. This word finds its roots in the 1970's, during which the kernel of an operating system was called the supervisor. A hypervisor would thus be the supervisor of the supervisors, with 'hyper-' aptly used as a stronger variant of 'super-'.

The OS that runs the hypervisor (as far as this is applicable, see 2.2.3) is called the host OS, while the OS that is run by the hypervisor is called the guest. We assume that hypervisors only run guests that have the same CPU architecture as its guest, and call this virtualization. If the guest deviates from its host CPU architecture, e.g. an x86-64 host running a RISC-V guest, we speak of emulation. Although the technology behind containers is sometimes referred to as OS-level virtualization, strictly speaking, no form of virtualization is involved. As such, a container is not a virtual machine. A relationship similar to the guest-host relationship with hypervisors remains with the use of containers, where the container itself is the guest and the OS that runs the container is the host.

For the remainder of this section, we will first briefly touch upon the history of virtualization. We will then discuss the various techniques used in virtualization.

2.2.1. A brief history of virtualization

Originally published in 1974, the classic paper on virtualization by Popek and Goldberg [68] describes three essential characteristics for a system to be considered a hypervisor:

1. The hypervisor should provide an environment that is essentially identical with the original

(host) machine. Essentially here means that a program running under the hypervisor should exhibit an effect identical to that running on bare metal, excluding timing effects.

2. Programs running under the hypervisor should exhibit at worst only minor decreases in performance. This necessitates that a significant fraction of machine instructions must be executed without hypervisor intervention.

3. The hypervisor is in complete control of computing resources.

During this time, the only feasible way of implementing a hypervisor that satisfied all three requirements was a specific style of virtualization called trap-and-emulate. The tight vertical integration of the hypervisor, hardware and guest operating system (also typical for this time, as just one company provided all three components [2]) did allow for research into refining this trap-and-emulate virtualization. For example, IBM's System 370 introduced a new hardware execution mode called *interpretative execution* [69], which allowed for a reduction of traps to a privileged execution environment in comparison to trap-and-emulate. Another approach was enriching traps such that hypervisors could handle them more efficiently [49]. However, not every instruction set architecture makes it possible to implement this classic trap-and-emulate method. The popular x86 architecture has historically lacked support for this trap-and-emulate, and as such, new methods had to be devised to enable virtualization on this architecture.

2.2.2. Hypervisor techniques

The role of a hypervisor, in essence, is to run virtual machines. Virtual machines are emulations of real machines. Virtual machines can run and feel like regular non-virtual machines, but do provide a strict isolation boundary between potentially multiple virtual machines. In contrast to containers, virtual machine guests should run exactly as they would if they were not being virtualized, as per the first characteristic described by Popek and Goldberg [68]. In practice, this means that a virtual machine has its own kernel and its own hardware devices, separate from the host machine that it is running on. These hardware devices and their corresponding device drivers are also virtualized, and represent physical devices such as a disk drive or a network interface card.

To understand how CPU instructions are virtualized, we should first recall that x86 instructions can be run with varying degrees of privilege. On x86 this level of privilege is organized into different rings, ranging from ring 0 (most privileged) to 3 (least privileged). Generally speaking, kernel code is executed in ring 0 whereas user-space code is executed in ring 3. This setup is depicted on the leftmost diagram in Figure 2.2.

2.2.3. Type-1 vs. Type-2 hypervisors

The other element that is prerequisite to understanding virtualization is where the hypervisor fits into the overall picture of virtualization. A hypervisor is responsible for the entire lifecycle of creating, running and tearing down virtual machines. Hypervisors can be categorized into Type-1 and Type-2 hypervisors, although the line between the two is not always clear. Type-1 hypervisors run directly on the host hardware and manage all running operations systems. An example of a Type-1 hypervisor is Xen. Type-2 hypervisors run on a conventional operating system, and abstract the guest operating systems from its host. Type-2 hypervisors run as a regular process on the host operating system. An example of a Type-2 hypervisor is QEMU.

An example of where the line between Type-1 and Type-2 hypervisors become blurry is with KVM, a popular hypervisor in Linux. Starting with Linux version 2.6.20, released February 2007, KVM was merged into the Linux kernel as a kernel module. KVM allows programs to use the hardware virtualized support that is available within modern CPUs. In essence, KVM allows for virtual

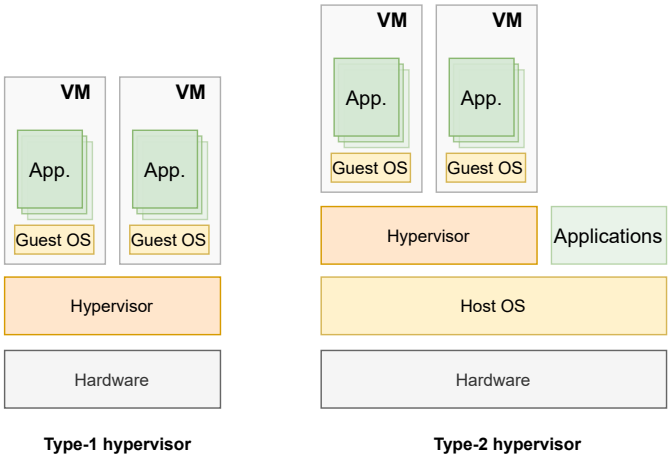

Figure 2.1: Architecture of type-1 vs. type-2 hypervisors

machines to be created within the kernel of Linux, thereby not quite satisfying the properties of neither a Type-1 hypervisor (it does not directly run on the host hardware, instead it uses the already running kernel) nor that of a Type-2 hypervisor (KVM does not run as a regular process as it is part of the kernel). Confusingly, KVM is often used in conjunction with other Type-2 hypervisors like QEMU, a combination often referred to as QEMU/KVM. In this setup, KVM is simply used to allow for the aforementioned CPU assisted hardware virtualization, and QEMU is the user-space process that manages parts like resource allocation and device driver emulation. The cooperation between the QEMU user-space process and the KVM kernel module takes inspiration from the original design of the Xen hypervisor, in which a small control VM (Domain0) is introduced that mediates (and thus establishes cooperation) between the other virtualized VMs and virtualized devices.

2.2.4. Virtualization of CPU instructions

Regardless of whether a Type-1 or Type-2 hypervisor is used to run virtual machines, they all employ the same techniques to run CPU instructions executed in a virtual machine. Most instructions can simply run within a virtual machine as they would on a traditional machine, meaning as a user-space process in ring 3. Some instructions are considered privileged. A privileged instruction can only execute in ring 0. Hypervisors, such as QEMU, run as a user-space process, and can therefore not execute these privileged instructions. If they were able to execute these instructions, (guest) virtual machines would be able to obtain complete control over the host it is running atop of. Instead, if an application in the virtual machine attempts to run privileged instructions, a trap is caused. This trap will be routed by the CPU to a handler in ring 0 code.

Another special case of instructions exist. These are called sensitive instructions, and are called so because they modify the resources of the machine. If all sensitive instructions would be privileged, then the hypervisor could simply create ring 0 handlers for each of these instructions. Unfortunately, instructions that are sensitive but not privileged, also exist. Such instructions are called non-virtualizable instructions, and are part of the reason why x86 was traditionally considered an architecture that could not be virtualized (as discussed in 2.2.1).

There are 3 general techniques for handling non-virtualizable instructions:

1. Binary translation. The non-virtualizable instructions are replaced at runtime by sequences of instructions that are virtualizable and have the same effect. In essence, the set of all x86 instructions is translated to a subset of x86 instructions that does not include non-virtualizable

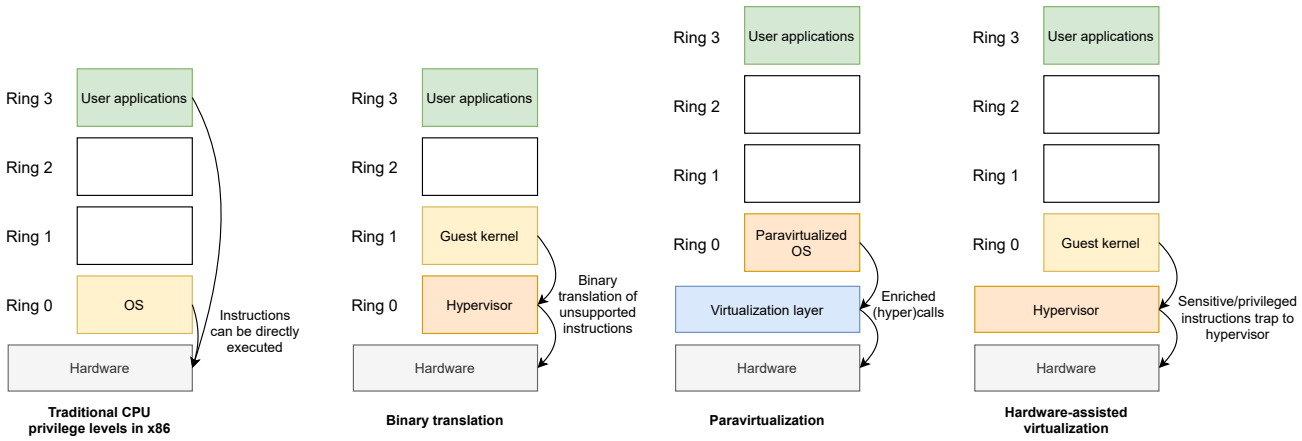

Figure 2.2: From left to right: x86 privilege levels without virtualization, binary translation, paravirtualization and hardware-assisted virtualization. Based on [59].

instructions. This technique is sometimes also referred to as a modern variant of trap-and-emulate [2], where trap refers to the mechanism that kicks into action when the CPU hits a non-virtualizable instruction, and emulate refers to the translation of this instruction. This setup is shown in the second diagram from the left in Figure 2.2. This technique was pioneered by VMWare in 1998 [59].

2. Paravirtualization. This technique avoids having to deal with non-virtualizable instructions altogether by rewriting the operating system such that non-virtualizable instructions are replaced by system calls to the hypervisor (dubbed hypercalls). An advantage is that this is relatively simple to implement (vs. binary translation), but requires modification to the guest OS. This setup is shown in the third diagram from the left in Figure 2.2.

3. Hardware assisted virtualization. Hardware vendors such as Intel and AMD have built-in support for virtualization in all of their recent (since 2006) x86 processors. These processors introduce a new privilege: ring -1. This allows guest virtual machine kernels to run at ring 0 (which is what the guest OS kernel expects to run at), while the hypervisor can run at the extra privileged ring -1. For early generations of CPUs with virtualization support, binary translation was faster [59], but in recent years it has been overtaken in performance by hardware-assisted virtualization. This setup is shown in the rightmost diagram in Figure 2.2.

Having explained how CPU instructions run in a virtualized way, we also need to look at how memory is virtualized using hypervisors. Even non-virtualized operating systems make use of virtualized memory. Each process that runs on a contemporary OS is under the illusion that it has a large contiguous chunk of memory at its disposal. In reality, the addresses that the process uses and thinks it controls, resolve to physical memory that may be dispersed all over the physical memory, or worse, may not even be present in memory at all. The allocation of memory is traditionally done at the granularity of chunks of 4KiB of memory (in x86), these chunks are called pages. The OS maintains a mapping from virtual page numbers to physical page numbers in its page table. The translation of virtual to real memory addresses is performed by the operating system, but is optimized by two hardware components: the memory management unit (MMU) and translation lookaside buffer (TLB), which functions as a cache.

To run multiple virtual machines on one machine, virtualization of virtual memory must be virtualized. In other words, the MMU needs to be virtualized for each VM. This way, the guest OS can continue to control the mapping of its own memory, but it cannot directly control the physical

memory. In this setup, the hypervisor performs the mapping from the guest memory to the actual memory. To do this more efficiently, it uses shadow pages. The hypervisor maps the virtual memory in shadow pages directly to physical memory, updating the TLB whenever the guest updates its virtual memory. Using shadow pages circumvents the need for having to go through the host OS virtual memory page table, resulting in efficiency gains.

The last element we need to look at besides CPU and memory is device virtualization. Contemporary hypervisors generally run guests with paravirtualized device drivers. Paravirtualized device drivers are in essence drivers that know that they are being virtualized, and exploit that knowledge for performance improvements over traditional virtualization. These drivers are presented to the guest operating systems by the user-space hypervisor process. QEMU in the QEMU/KVM equation is responsible for this, for example. Paravirtualized drivers consist of two parts, one talking to the guest, and the other part talking with the host. The driver that is presented to the guest is called the device front-end. The host-facing driver is called the back-end driver. These two ends need to be connected by a transportation mechanism, for which virtio is considered the industry standard and is used by virtually all common paravirtualized drivers. Virtio is not merely a way to transport information, but a generalized abstraction of a commonly required set of operations within paravirtualized drivers. This results in virtio becoming a standard interface (or, API) through which the front and backend drivers can communicate [81]. Host kernel support for the back-end drivers is not required as support is implemented in the host user-space hypervisor process.

3

Isolation platforms

In this chapter we first discuss the tradeoffs that naturally arise when implementing isolation platforms. We then continue by introducing each of the platforms that we will be examining in this thesis, and explain which problem they aim to solve, including its potential drawbacks. We conclude this chapter by arguing why we decided to examine these specific isolation platforms over others.

3.1. Tradeoffs in isolation platforms

Virtualization is a method to present computer resources in a logical way that is not constrained by the physical hardware. Virtualization does this by introducing an abstraction layer in between the hardware and the operating system kernel. This layer provides a means to *isolate*: users may make use of the same physical hardware, but are agnostic to both the fact that the hardware is being shared as well as the existence of the other users. There is no free lunch, however, and the additional layer establishes a set of tradeoffs to be made which are inherent to the use of an intermediary virtualization layer. For example, raw performance is typically penalized by introduction of an isolation mechanism. In the following paragraphs, we outline these tradeoffs.

The first tradeoff that becomes apparent within virtualization is the generality offered versus the size of the trusted computing base. On one hand, it might sound appealing to introduce a virtualization layer that implements support for all hardware that one can possibly imagine, as this will make it possible for the virtualization layer between any combination of (possibly exotic) hardware and operating system. However, this generality in the form of wide-ranging support necessarily also introduces complexity. Complexity implies, almost by definition, more code, and therefore more software vulnerabilities. These flaws can, in some cases, be exploited by adversaries, and consequently violate the isolation property that virtualization makes valuable in the first place. Figure 3.1 places each of the examined platforms on a spectrum, in which on one side we find the most general form of virtualization (in which general purpose operating systems are virtualized), and on the other side the most specific types of virtualization (where we find unikernels, in which only one specific application is to be virtualized). The hypervisors are placed in a zoomed-in part of the spectrum, from most general hypervisor to least general, coinciding with the size of the device model these hypervisor platform offer. In this spectrum we also make a distinction between application and system containers, of which the difference between the two types is covered later in this chapter, in Section 3.3.

A second tradeoff is the level at which the platform implements its functionality. In the paragraphs prior we assumed that the intermediary layer sits between the hardware and the operating

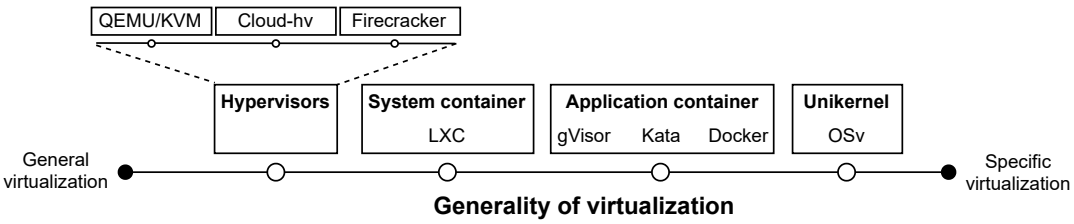

Figure 3.1: Spectrum of the generality of guests that the isolation platform offers. This can also be interpreted as the size of the inverse of the trusted computing base.

system. This, however, is not always the case, as exemplified by containers introduced later in this chapter. The isolation boundary is not placed between the hardware and operating system, but rather between the operating system and the processes that run on it. But even within one class of isolation platforms the location of the virtualization layer is not set in stone (e.g. Type-1 vs. Type-2 hypervisors, see Subsection 2.2.3). The location of this boundary determines where functionality of the platform gets executed: a boundary closer to the host kernel means that more functionality is executed in the guest kernel, a boundary closer to the guest means that more functionality will get executed in the host kernel. An overview of where each platform sits in this spectrum is illustrated in Figure 3.2. At the leftmost end of the spectrum we find a Linux native host, where there is no isolation boundary at all, and as such everything is executed in the host kernel. Secure containers generally use the host kernel, but find ways to reduce the lower the isolation platforms closer to the guest. The aim of this lowering of the boundary is to increase security, hence the name secure containers. At the right edge of the spectrum we place the hypervisors, that spin up a dedicated fully featured guest kernel for each guest and as a result make relatively little use of the host kernel.

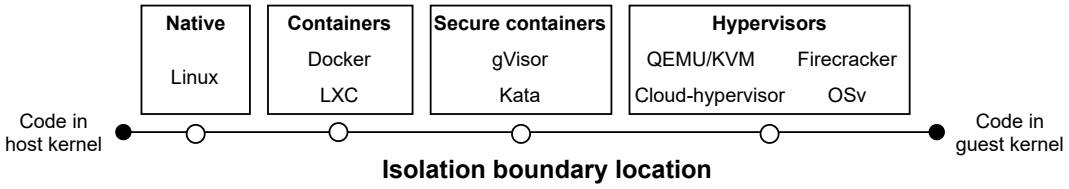

Figure 3.2: Graphical illustration of the spectrum of where functionality for each isolation platform is located. A boundary closer to the host indicate more functionality residing in the guest kernel, and vice versa.

Another tradeoff that is to be considered is the degree of isolation a virtualization platform provides versus the performance overhead it imposes. This could be visualized as the thickness of the intermediary layer between the host and guest: a lean virtualization layer may provide isolation to a lesser extent but quick virtualization translation from operating system to hardware, whereas a thick virtualization layer provides significantly more isolation at the cost of an increased execution path length from operating system to hardware (resulting in lower performance). A smaller function chain from user to hardware in general would mean better performance but at the same time fewer hurdles to pass to reach the supposedly isolated hardware. This tradeoff between security and isolation is essential for every isolation platform, but placing the platforms in this spectrum is not trivial, and requires thorough experimentation.

We will now introduce and discuss the platforms we will examine in this thesis.

3.2. Docker

Docker is the de facto standard for creating containers. Containers here refer to a way in which the host kernel facilitates the existence of multiple isolated user-space instances. This idea is not new: software like BSD jails and `chroot` provide similar functionality and have been around for decades. Building upon this idea, the release of Solaris 10 in 2005 introduced a modern and improved container implementation called Zones [70]. Linux, at the time, did not have a competing solution, and only with the merging of user-space namespaces in Linux kernel version 3.8 released in 2013, proposed a decent and mature alternative. Since then, container popularity steadily increased, accelerated by the ease-of-use of Docker and container orchestrators like Kubernetes. It was around this time the word 'container' was adopted as well. Early on, Docker built on top of the `liblxc` library (and is still being used by LXC, see next section), but now uses its own implementation to provide isolation.

Docker, and its parent company Docker Inc., also introduced various standards pertaining to containers. These, now popular, standards are regulated by the Open Container Initiative[1], formed by the Linux Foundation. Currently, the OCI regulates two standards: the container runtime and image specification. These standards ensure a standardized workflow independently of the used container engine. Examples include conventions for being able to run containers without any additional arguments (`docker run container`), copying data between host and guest (`docker cp`) and compatibility of container images across various container engines. Container technologies that adhere to these standards are considered 'OCI compliant', a few of which we will introduce later on in this chapter.

3.3. LXC

Like Docker, Linux Containers (LXC) provide a way to create containers that take advantage of features present in the host kernel. The distinguishing characteristic of LXC is that it provides a complete user-space of an operating system, whereas Docker emphasizes running one specific application per container. Another way of putting it is that LXC provides OS-oriented containers with VM-like behavior, in contrast with Docker its application-oriented containers.

In practice the management of LXC containers is generally performed through LXD. LXD employs the lower-level interface that LXC exposes, allowing for a better user experience. LXD offers integration into the OpenStack and OpenNebula platforms. Integration with the latter allows for the usage of `qcow2` and regular KVM-like images to be run on LXC, adding to the aforementioned VM-like behavior.

3.4. QEMU

QEMU is one of the three hypervisors we will closely examine in this thesis. Of the three, QEMU is by far the oldest (with its initial release in 2003 [71]), and supports a wide range of guests. The other hypervisors examined in this thesis are primarily focused on virtualizing Linux guests exclusively, whereas QEMU advertises itself as a machine emulator, supporting a wide range of both virtualization and emulation of guests. The originally intended primary usage of QEMU was, and still is, to run one operating system on another, such as Windows on Linux or Linux on Windows [11]. Another defining characteristic for QEMU in our hypervisors is its support for debugging use cases: as guests can easily be stopped, its state can be inspected at any point in time. This led to the popularization of QEMU for various purposes, including (toy) kernel research, fault injection simulators [4] and program instrumentation, among many other purposes [35]. A downside to the wide-ranging sup-

[1] `opencontainers.org`

port for even exotic guests is the complexity that it entails, which is one of the primary motivations for the creation of the hypervisors discussed in the following chapters.

3.5. Firecracker

Firecracker, developed by Amazon, uses KVM to create and run Linux virtual machines. Firecracker started out as a fork of Google's crosvm, a hypervisor written in Rust, but has since diverged significantly to serve different needs [30]. Crosvm has been built for virtualizing and thereby securing Google's Chrome OS, whereas Firecracker focusses on cloud use cases. Firecracker aims to retain a minimalist design, excluding unnecessary virtual hardware devices and guest-facing functionality, in order to reduce the memory footprint and attack surface of each virtual machine. Specifically, Firecracker focusses solely on virtualizing specific bare Linux guests on a Linux host, and for example does not offer a BIOS, cannot boot arbitrary kernels, does not emulate legacy devices nor PCI, and does not support VM migration [3] (although the latter is under development). The driving factor for this minimalism is Firecracker its focus on use in the serverless paradigm, which requires low startup overhead.

3.6. Cloud-hypervisor

The final hypervisor we will be examining in this thesis is Cloud hypervisor. Cloud-hypervisor is the most recent of the three, with its original (development) release in 2019. Like Firecracker (and crosvm, for that matter), Cloud-hypervisor is a hypervisor written in Rust and makes use of KVM and emphasizes safety and security. As such, with Cloud-hypervisor being the latest hypervisor to come to existence, it makes extensive use of functionality already implemented by Firecracker (and consequently, crosvm) through the rust-vmm crate. The goal of Cloud-hypervisor is to support all modern cloud workloads, i.e. full Linux distribution images currently in use by cloud tenants. One could argue that Cloud-hypervisor establishes a middle-ground between QEMU (supporting a multitude of operating systems and CPU architectures, and legacy devices like floppy disk drivers) and Firecracker (running everything as bare-bones as possible, focusing on extreme minimalism).

3.7. Kata containers

Kata containers is an open-source security-oriented platform that promises a way to run OCI compliant containers by using hardware-assisted virtualization as a second layer of defense. The Kata containers project finds its origin in the merging of Intel's Clear Containers and Hyper.sh's runV open-source projects, both of which aimed to create more secure containers by making use of virtualization technology. Kata containers provides an OCI compatible runtime and can thus be used within the existing Docker ecosystem. Under the hood, Kata containers spins up a VM using a regular hypervisor and within that VM constructs a OS-level virtualized container. Strictly speaking, this setup violates the definition of a container, since the host kernel is not used to isolate guests. Instead, a VM with guest kernel is started to provide this functionality. The main value proposition of Kata containers, however, is that it can provide the usability of regular containers but the security of hardware-assisted virtualization. In the words of the authors on the project homepage [2], Kata containers is an isolation platform that have "the speed of containers, and the security of virtual machines". Implementing a platform like Kata containers necessarily introduces complexity, thus an increased trusted compute base, even more so when taking into account that virtual machines and containers are not necessarily built to be compatible with one another.

[2] `katacontainers.io`

3.8. gVisor

Another alternative to security-oriented containers is provided by gVisor [36]. Released by Google in May 2018, gVisor intercepts all of the calls from the container and executes them directly, adding an abstraction layer in between the host kernel and the container. This setup may be reminiscent of the platform discussed in the previous section (see the previous Section 3.7), but instead of using virtualization technology, gVisor executes all system calls from containers within a user-space application kernel that has been reimplemented from scratch. This approach thus provides another layer of abstraction (which should increase isolation) but does not require the use of existing virtualization platforms (allowing for a flexible resource footprint, and as a consequence, higher container density). A disadvantage may be found in system call intensive workloads, since all system calls have to be redirected and executed within a potentially less optimized user-space kernel.

3.9. OSv

Unikernels take a distinctive approach to operating systems. This relatively new approach enables the creation of small virtual machine images. The main idea with unikernels is that the image is tightly integrated with the OS it is running on, only including the components that are strictly required by the application. This image thus includes both the application and the operating system, and therefore removes the need for any external operating system. From the perspective of the application, the operating system becomes nothing more than a library it can call into, and everything can run into a single address space. Although the popularity of unikernels is relatively new, its ideas are not, and have already been successfully implemented in prior academic projects such as the Exokernel [27] and Nemesis [54] operating system architectures.

The design of unikernels are in stark contrast to how virtual machines are currently run on public cloud infrastructures, where full GNU/Linux distributions (e.g. Ubuntu Server) are the norm [51]. The primary value proposition of unikernels is the reduction of required run-time software complexity, since it only includes what is strictly required to run its workload, instead of a complete operating system that provides mostly irrelevant functionality. The advantage of this approach is twofold:

1. Unikernels reduce the amount of code, therefore reducing attack surface, increasing security.

2. Application images are very small, reducing the resource (memory and storage) footprint.

The reduction in complexity naturally comes at a cost, most often in the form of decreased interoperability with existing software. Since applications must be (re)compiled to run in a unikernel, possession of this source code is a hard requirement. Moreover, due to the single address space and single process design of unikernels, there is no support for running multiple processes in one unikernel (as is often exemplified by the absence of a `fork()` system call in unikernels).

The unikernel that we will be looking at in this document is OSv [51]. We have chosen OSv because (1) it is a unikernel that is able to compile and run existing source code, so we can fairly compare it to to other isolation platforms, and (2) it is under active development. OSv started was originally designed and implemented by Cloudius Systems (now ScyllaDB[3]), but is currently maintained and being refined by volunteers [65].

[3] `scylladb.com`

3.10. Selection of isolation platforms.

In the previous sections we have introduced all of the platforms we will closely examine in later chapters. In this concluding section, we argue why we picked certain platforms. The reasons why we have selected some isolation platforms and did not pick others are best described on a per-category basis:

- Containers: where picking Docker is an obvious choice due to its massive popularity, LXC might be a less obvious choice. Given the shared history of Docker and LXC (through `liblxc`) and LXC being the only real competitor to Docker at the moment of writing, we decided to also include LXC. Other container technologies exist, such as `systemd-nspawn` [79], but in our experience did not offer the same level of maturity. Another container runtime that was considered for inclusion is `crun`, a reimplementation of Docker its default `runc` in the C programming language. This however would turn into a discussion of the performance and security of the programming *language*, rather than that of the isolation techniques we focus on in this thesis.

- Hypervisors: The inclusion of the ever-popular QEMU might not come as a surprise, nor the inclusion of Firecracker, which has been relatively popular since its introduction on Amazon AWS Lambda[4]. We also decided to include Cloud-hypervisor as a third hypervisor, as it sits in the middle of feature-complete QEMU and bare-bones Firecracker. The exclusion of the other well-known Xen hypervisor was decided upon in order to retain a feasible scope for our research.

- Secure containers: at the moment of writing, there are two main project that focus on creating secure containers, Kata containers and gVisor. Both are included. A third secure container project called Nabla container exists, but active development ceased in early 2019.

- Unikernels: To fairly compare the performance of unikernel versus the other platforms, we need a unikernel that can compile and run existing source code (i.e. a unikernel of the second type as outlined at the end of Section 3.9). Out of the available unikernels of this type, we have found that only the OSv kernel is stable enough to undergo our performance and security experiments (and only barely so, e.g. OSv crashes at load-time during the I/O benchmark in Subsection 5.3.1).

The reason for looking beyond just the classic hypervisors is the emergence of new programming models and services offered by cloud platforms, as is also discussed in Section 2.1.

[4] `https://aws.amazon.com/lambda`

4

Isolation platform architectures

In this chapter we consider the architecture and technical aspects of each platform.

4.1. Docker

Docker refers to an entire software suite including a container lifecycle manager, packaging software and interface for communicating with a repository of online container images (Docker Hub). Broadly speaking, Docker uses a client-server architecture. The CLI client is what an end-user interacts with through familiar commands prefixed with `docker` (e.g. `docker run`). These commands are sent to the Docker daemon `dockerd` (the server part in the client-server), and performs all the heavy lifting of building, running and distributing the Docker containers [26]. The client and server communicate through a rest API over a local or remote UNIX socket. Instead of having to create a new container from scratch, Docker also features a built-in way to communicate with so called registries. These registries host a multitude of existing container images, and these can be downloaded from (and pushed to) through the client (with `dockerd` again performing the actual interaction). An overview of the architecture is presented in Figure 4.1

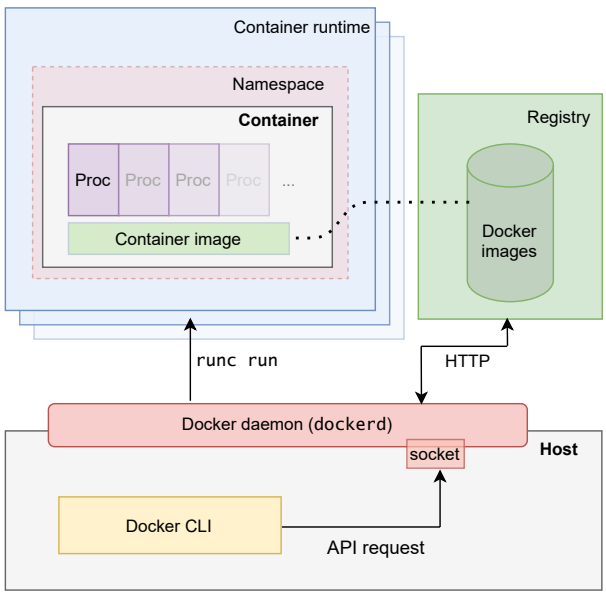

Figure 4.1: Architecture of the Docker container platform

The missing piece in the architecture so far is the component that actually creates the isolated containers: the container runtime. The default Docker container runtime is `runc`. This runtime, given a layered filesystem and related container metadata, creates a new isolated container. As such, `containerd` uses this runtime whenever a user has requested to create a new container. It is also this component that gets replaced by other container isolation platforms that we will introduce at a later point in this chapter. `runc` uses functionality exposed by the Linux host kernel to enforce isolation between a container and the host operating system. The kernel is thus shared between the host operating system and the container, and no new kernel is booted. The two main kernel features that are core to `runc` its isolation are namespaces [64] and cgroups [17].

In the Linux kernel namespaces are a way to manage the visibility of a certain set of resources from the perspective of a process. By putting a process into a specific namespace, the kernel can restrict the resources that are visible to this process. Thus, Docker uses namespaces as a way to provide an isolation boundary between guest and host. There are various types of namespaces available in the Linux kernel, each of which manage the visibility of a specific set of namespaced resources. Docker currently uses the following namespaces:

1. The mnt (mount) namespace: mount namespaces restrict the list of mount points that can be seen by the process. In practice this implies restricting the files that are visible to the process, such that a process in a namespace cannot see or alter files outside of this namespace.

2. The PID namespace: provide processes with a set of process IDs that are unique to this namespace. PID namespaces are nested. In a newly created PID namespace the initial process gets assigned PID 1. This new `init` process is still visible to the parent PID namespace, albeit with a PID other than 1.

3. The network namespace: each and every container gets its own virtual network stack. By default, only the loopback device is present in a new network namespace. Docker adds its own bridge as a network device to this namespace, connected to the host network bridge through a virtual ethernet pair. Each network device is, at any time, part of exactly one network namespace.

4. The UTS namespace: provides isolation of the hostname and NIS domain name system identifiers.

5. The IPC (interprocess communication) namespace: every IPC namespace gets its own set of System V IPC identifiers (for use in message queues, semaphore sets and shared memory segments) and its own POSIX message queue filesystem (as present at `/proc/sys/fs/mqueue` on most unix distributions).

The namespaces mentioned above are a subset of all the types of namespaces available in the Linux kernel. At the time of writing, a total of 8 different types are available. A process is always part of exactly one namespace for each distinct type. Thus, a process is always part of exactly 8 namespaces, one for each type. A process can become part of a namespace either when the process gets created through `clone()`, the process calling `unshare()` by itself, or by `setns()` which sets the namespace given a file descriptor.

Whereas namespaces limit the visibility of resources, control groups (cgroups) limit the usage of resources, such as CPU, memory and network input/output. Note that this also reduces the denial of service attack vector, since resource hogging should in theory not be possible from within a container. By default, processes on Unix systems inherit nearly everything from their parents.

This also holds for cgroups: upon creation of a new child process, it inherits the cgroups of its parent. The Linux kernel exposes an interface with cgroups through the cgroupfs pseudo-filesystem. This pseudo-filesystem typically resides at `/sys/fs/cgroup`, and lists the available subsystems that cgroups can limit. An example of such a subsystem is 'memory', and in turn lists the properties pertaining to the memory subsystem that can be limited using cgroups. An example property that limits the maximum number of bytes in a cgroup is `/sys/fs/cgroup/memory/memory.limit_in_bytes`.

4.2. LXC

Linux containers (LXC) approach the implementation of containers in a way similar to `runc`, using namespaces and cgroups as the main isolation mechanisms. In fact, up until a year after the release, Docker used LXC as a library (`liblxc`) to set up its containers (but now uses its own separate reimplementation called `libcontainer`). The characteristic that sets LXC apart from Docker is its ability to create an environment as close as possible to a standard Linux installation, without the need for a separate kernel. Concretely, this means that LXC containers:

1. Run a fully-fledged `init` system such as systemd, whereas Docker uses `tini` that markets itself as 'the simplest init you could think of'.

2. Use a feature-complete general filesystem instead of a layered filesystem like Docker. By default, LXC uses the ZFS filesystem for its containers. Through use of the OpenNebula platform[1], it is even possible to create containers based on `qcow2` and KVM-like images.

It is worth mentioning that LXC already provides the user with a way to run non-root unprivileged containers, making use of the newer cgroups v2. Docker, at the time of writing, only offers running containers using root privileges.

4.3. QEMU

Although all hypervisors closely follow the architecture and techniques mentioned in Section 2.2, the architecture of these platforms can differ substantially. Recall that for example QEMU also supports guests that have a foreign CPU architecture (that is, a CPU architecture different from the host), potentially affecting the overall architecture of QEMU. QEMU supports two general execution modes:

1. User-mode emulation: Emulate and run a Linux/BSD process that is compiled for a foreign CPU architecture.

2. System-mode emulation: Emulate or virtualize a complete system, including (virtual) CPUs and hardware devices.

The focus in this thesis is on the latter mode of emulation.

An end-user creates a QEMU VM through the `qemu` (or architecturally specific variations thereof, such as `qemu-system-x86_64` for `x86_64` guests) program. For every VM there is a separate process, and is scheduled on the host OS like any other process. The host does not and can not see which processes are running within a VM, unlike the case with namespace-based isolation platforms. Memory for guests is provided through allocation by the host process, and is then mapped to the guest its address space using `mmap()`. The allocation can be backed by either RAM or file-backed memory (e.g. `hugetlbfs`). The guest sees this memory as its own physical memory.

[1] `opennebula.io`

QEMU processes requests from multiple sources while virtualizing guests. At its core, QEMU uses an event-driven architecture and reacts to events by continuously polling whether an event has happened, and if so, dispatch it to the appropriate event handler [38]. In QEMU, this main loop is `main_loop_wait()`, and handles the following types of events:

- Waiting for registered file descriptors to become available. These file descriptors get registered by various resources, such as the TAP device for networking, audio (ALSA), and recent virtio (see later chapter) implementations.
- Run expired timers.
- Requests for invoking a function in another thread (such requests are called called bottom-halves).

In order to continuously and consistently execute iterations of this main event loop, all of the handlers for these events need to be executed quickly. This means that they should be non-blocking, and not require any form of synchronization. However, there are cases where it is impossible to avoid either of these two, in which QEMU employs separate dedicated worker threads to move the blocking computation out of the main loop. For example, the POSIX `aio` I/O interface uses worker threads in order to implement asynchronous I/O. Another example that uses worker threads is whenever the graphical display of the guest needs to be encoded (to be viewed by the host through a VNC viewer). There are also special vCPU threads (one for each virtual CPU) and even multiple main event-loop threads.

As we have discussed in a previous chapter, hardware-assisted virtualization enables the native execution of guest code in a special guest CPU mode. The KVM Linux kernel module takes care of this. QEMU, when enabled, interfaces with KVM to create and run the KVM VM in guest CPU mode. QEMU thus only has to handle requests from the guest again whenever it traps out. A typical loop that creates, runs and handles traps from a KVM guest looks as follows[39][38]:

```
1  open("/dev/kvm")
2  ioctl(KVM_CREATE_VM)
3  ioctl(KVM_CREATE_VCPU)
4  for (;;) {
5       ioctl(KVM_RUN)
6       switch (vcpu->kvm_run->exit_reason) {
7          case KVM_EXIT_IO: /* ... */
8          case KVM_EXIT_HLT: /* ... */
9       }
10 }
```

Listing 4.1: Code snippet in C that implements a loop that creates, runs and handles traps from a KVM guest. Included libraries are omitted for brevity.

As we see in Listing 4.1, the special `/dev/kvm` file is opened first (and is only available on hosts that support KVM), and the VM and vCPUs are created through specific `ioctl()` system calls. Executing `ioctl(KVM_RUN)` (resulting in a VM_ENTRY) hands over control to the guest, and keeps running unless the guest traps back to QEMU. The exit reason is read by QEMU and handled accordingly, and is then resumed again. While the guest is executing its own code, in extreme cases, it would be possible for the guest to never yield back control to the host, and thus the main event loop of QEMU would never continue iterating.

A figure that illustrates the global architecture of QEMU and summarizes the previous paragraphs is provided in Figure 4.2. The figure is split up into two separate domains: host and guest. Note that both host and guest could be further divided into Ring 0 and Ring 3 parts, but are omitted to retain the clarity of the figure. In this figure we see how the vCPU threads in the host are presented to the guest as physical CPUs, as well as device drivers like a network and disk driver.

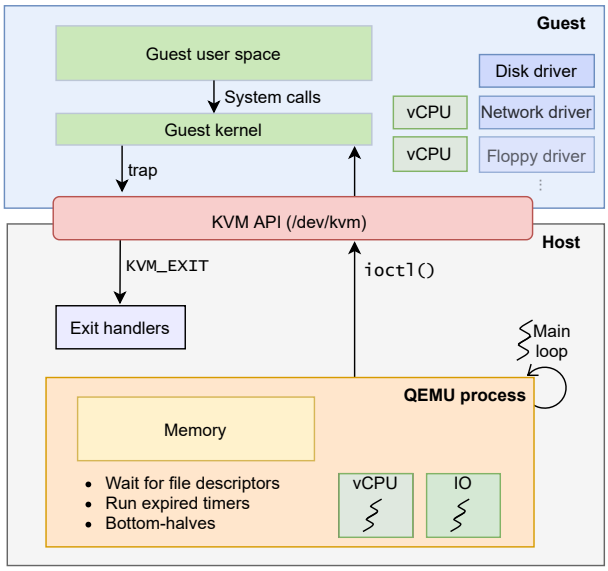

Figure 4.2: Architecture of the QEMU hypervisor

4.4. Firecracker

Firecracker adopts an event-driven architecture and uses KVM to create and run VMs, much like QEMU. The usage of threads for the main loop and virtual CPUs is also adopted in Firecracker. What sets Firecracker apart however is its introduction of HTTP server threads. These threads expose a REST API through a socket on the host and is used to manage the VM instance. Once a Firecracker VM has been started (with the `InstanceStart` API call), the API server threads will block on the `epoll` file descriptor and wait for another API request to come in.

In an effort to contrast Firecracker to QEMU/KVM, the developers claim that Firecracker has a minimal device model. In total, Firecracker supports only a handful of emulated devices, divided into 3 groups:

1. Virtio devices: network, vsock, block and a memory balloon device (implemented in `virtio-net`, `virtio-vsock`, `virtio-blk`, and `virtio-balloon` respectively)
2. A legacy (i8042) serial and PS/2 mice and keyboard controller.
3. A pseudo clock device that records the time since booting.

Whereas QEMU supports significantly more devices, like USB and GPU devices. In recent releases, QEMU also features support for running virtual machines with a minimal device model (constructed in response to the advent of Firecracker). The minimal device model of Firecracker was initially conceived to reduce start-up time for virtual machines, which is particularly beneficial in a serverless context, in which VMs tend to be short-lived (created just before and killed after executing the function), and as such reducing the start-up time is essential for keeping the overall

function completion time low.

Another technique Firecracker implements to reduce boot time is by making use of the Linux 64-bit boot protocol. This allows for booting directly into 64-bit mode, skipping the usual x86 mode-by-mode (from the 16-bit real mode to 64-bit long mode) booting protocol. Furthermore, Firecracker boots directly into an uncompressed Linux kernel, starting at the 64-bit entry point [46]. This is different from typical Linux platforms, in which the kernel decompresses itself at startup.

The minimal device model of Firecracker also allows for a reduced attack surface, allowing for fewer system calls and execution with fewer privileges than QEMU/KVM. The emphasis on extreme minimalism in the Firecracker project should particularly be apparent when we discuss the security aspect of the isolation platforms in Chapter 6.

4.5. Cloud Hypervisor

From an architectural point of view, Cloud-hypervisor is similar to the other hypervisors discussed in this text. It is particularly similar to Firecracker. Cloud-hypervisor uses a similar architecture and applies similar techniques to reduce boot time, for example by using the Linux 64-bit boot protocol. The main difference is that Cloud-hypervisor finds a balance between a minimal hypervisor (Firecracker) and a very feature-complete hypervisor (QEMU), slightly leaning towards the minimalism of Firecracker. As such, the architectural properties of Cloud-hypervisor are expressed here in terms of how it deviates from the Firecracker design.

Cloud-hypervisor supports 16 different devices, in contrast to the 7 of Firecracker and 40+ of QEMU. The majority of the devices in this device model are paravirtualized `virtio` devices. In contrast to Firecracker, Cloud-hypervisor also supports vhost-user devices: these are devices that have `virtio` backends running outside of the hypervisor as a separate process. Vhost-user device backends implement a master-slave architecture, where the hypervisor is the master and the slave is the separate process. The communication between the two is implemented in the `vhost-user` protocol through an extension of the `ioctl()` interface. An example of such a device, `virtio-fs`, is discussed in a later chapter. At the time of writing, implementations of `virtio-vhost-user` are being developed, taking vhost-user one step further. Virtio-vhost-user make it possible for the master and slave components to run within the VMs, establishing a direct (but virtualized) connection between the two. This is implemented by tunneling the vhost-user protocol over a `virtio` device. This has the advantage of bypassing any extra layers in the host (like a network switch), possibly leading to an increasing in performance. Vhost-user devices are also implemented in QEMU.

Hotplugging memory and vCPUs is also supported by Cloud-hypervisor. Requests for hotplugging are performed via the API that Cloud-hypervisor exposes. Memory is hotplugged by first allocating memory on the host (and must be a multiple of 128MiB) and then mapped from the hypervisor userspace process to the virtualized physical memory of the guest. Hotplugging extra CPUs is implemented by the host performing a `_CREATE_VCPU ioctl()` call, and are then advertised to the running guest kernel using `ACPI`. The newly provisioned vCPUs are not automatically used within the guest but have to be brought online by manual interaction with the guest Linux kernel `sysfs` interface.

The two features described above are simply to illustrate its differences from Firecracker. For our purposes, the Cloud-hypervisor and Firecracker hypervisors are quite similar, and should exhibit similar performance and security characteristics. In later chapters we will verify whether this is indeed the case.

4.6. Kata containers architecture

The Kata containers architecture consists of multiple smaller entities. Figure 4.3 illustrates the architecture of Kata containers, which shows how a hypervisor is used in conjunction with namespaces. The entry point of the Kata containers architecture is `kata-runtime`, which the user interacts with through the Docker `dockerd` daemon (for example by running `docker run`) by implementing the Containerd Runtime V2 interface [24].

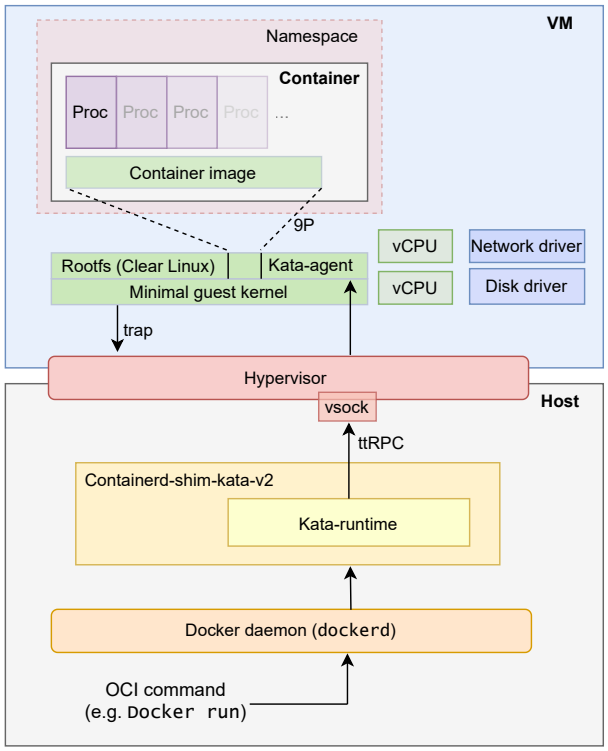

Figure 4.3: Architecture of the Kata containers secure container platform

The `kata-runtime` component is responsible for starting the hypervisor. A hypervisor needs a kernel and root filesystem to start, and Kata containers handle that the following way:

1. Kernel: shipping with the kata-runtime, there is a Linux kernel that 'is highly optimized for kernel boot time and minimal memory footprint'. This optimization in practice boils down to disabling almost all kernel features for the guest kernel using `kconfig`.

2. Root filesystem: It passes a 'mini OS' as root filesystem. Although this mini OS is customizable while building from source, by default it is based on Clear Linux. Both `kata-agent` and systemd (which starts the `kata-agent` immediately) are included in this mini OS.

The `kata-agent` is a process for managing containers and processes running within a hypervisor. This agent communicates with `kata-runtime` using a ttRPC server (a reimplementation of gRPC specifically for low-memory environments [84]) that is exposed on the host by QEMU through a vsock file.

A confined (namespaced and cgrouped) context is created by the `kata-agent`, within the hypervisor. The root filesystem of this newly created confined context is that of the original Docker image, passed as a shared mount point from the host through QEMU. Other settings, such as which command should be run at start are initially passed to the kata-runtime in the Docker image. This

is set up within the new container context, and the workload is run. Recall that a 'Docker image' here simply means an OCI bundle, that is, a layered file system with a `config.json` at the top level, specifying options such as the entry point. So the workload is specified by the creator of the docker image using the `ENTRYPOINT` keyword in the Dockerfile, but is presented to the kata-agent in the `config.json` of the OCI bundle.

Whenever a `docker exec` statement is issued to kata-runtime, and a Kata container is set up already, it simply forwards this command to the kata-agent running inside the hypervisor, which delegates it to the confined context to create a new process with this new command.

One of the main components in Kata containers that requires extra attention is the I/O subsystem. Unlike most hypervisor use-cases, the filesystem of the docker image needs to be shared between the host and guest of the hypervisor. Sharing is required for OCI commands like `docker cp` (for copying data between host and guest). By default, Kata containers does this through the Plan9 filesystem, but a newer faster alternative called `virtio-fs` can be enabled as well (see Subsection 5.3.2). In the past, the Docker devicemapper I/O driver could be used, a driver that presents the root filesystem as a block device to kata. This resulted in much better I/O performance, but has been removed from Docker.

4.7. gVisor

gVisor takes a different approach in which no hypervisor is used. Instead, system calls in gVisor are intercepted and redirected through use of a 'platform'. Concretely this platform leverages either `ptrace` or `KVM`. The `ptrace` system call interception implementation employs `PTRACE_SYSEMU` to stop and intercept the execution of system calls into the host kernel. With KVM as the platform, the main gVisor process is run as a KVM VM. In general, the KVM mode ought to be faster because `ptrace` has a relatively high context-switch penalty while KVM can make use of hardware assisted virtualization features like fast address space switching [37]. This platform, and the rest of the general architecture of gVisor, is illustrated in Figure 4.4

Regardless of which platform is used, system calls get intercepted and consequently bounced back to a particular process in user-space. This process is called the Sentry. The Sentry is a kernel in user-space, implementing not just system calls but also features like signal delivery, memory management and the threading model. To reduce the attack surface, the system calls in the Sentry process are implemented using a small subset of system calls to the host kernel. This is enforced through seccomp filters, meaning that other system calls can never be made to the host kernel, even if an adversary has complete control over the Sentry process. The Sentry process itself runs as an unprivileged user and uses namespaces in the same way as Docker does (networking namespaces, filesystem namespaces, and so on).

The underlying idea in gVisor is that there is defense in depth. Not only does the Sentry process reimplement system calls to reduce attack surface, it also runs within its own namespace. Even if the Sentry process were to be compromised, the attacker would have to break out of the namespaces. This pattern is also applied with the Kata containers platform, in which breaking out of the namespaced context would only lead you to the next hurdle to pass: the hypervisor.

The seccomp filters applied to the Sentry also include all I/O related system calls. This means that the Sentry can not dispatch any I/O related system calls to the host kernel. Instead, the system calls coming in from the application that is run under the Sentry are dispatched to another gVisor

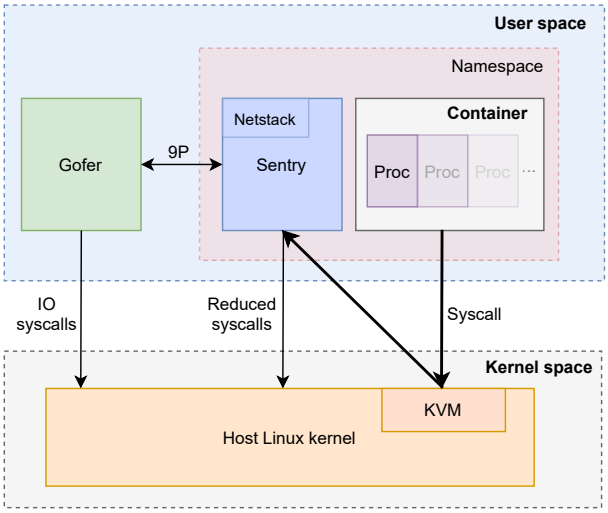

Figure 4.4: Architecture of the gVisor secure container platform

component called Gofer. The Sentry and Gofer process communicate via the 9p protocol, similar to how the file system is shared between the hypervisor guest and host in Kata containers. Recall that this 9p filesystem required extra attention in the Kata containers, which is also the case for gVisor.

Another point that needs attention in gVisor, as also noted by the developers themselves, is networking. The Sentry implements its own, written from scratch, network stack, made specifically for gVisor. This network stack is called Netstack, and just like other components in the Sentry intercepts system calls and reimplements them using fewer system calls to the host kernel. Contemporary network stacks are extremely vast however, meaning that it takes a lot of effort to implement every single feature and RFC specification currently in use by mature network stacks, such as the Linux network stack . Not implementing these features inevitably results in lower performance. Good examples of features missing in gVisor that affect performance are RACK [19] and BBR [15].

4.8. OSv

OSv is a unikernel that uses existing compilers and a custom kernel to call into. Specifically, the OSv kernel includes a dynamic ELF linker that can run standard code compiled for Linux. This linker maps the executable and its dependencies to memory. Whenever application code calls functions from the Linux ABI (through the standard C library), the linker dynamically resolves it to the corresponding function implemented by the custom OSv kernel. This means that system calls, called through the wrappers implemented in `glibc`, are treated as regular function calls, and do not lead to a (user-to-kernel) mode switch. Both the application and kernel (i.e. OS library) run in the privileged ring 0. An architectural overview of OSv is given in Figure 4.5

Running a unikernel image on OSv is done through existing hypervisors. The unikernel image is simply the image the hypervisor boots off. OSv images consist of a base image that is fused together with cross-compiled source code that calls into this base image. The OSv base image (kernel) exposes an interface that follows the Linux ABI convention. This setup makes it so that as long as executables are compiled as a relocatable shared object (`.so` in Linux) as well as position-independent (‘ PIE’, using the `-fPiC` compiler flag), recompilation and thus application source code is not required, and existing source code can transparently call into the OSv kernel. Note that there is a caveat here: since there is no support for multiple processes within one OSv VM, system calls such as `fork()` and `exec()` are not available. Down the line, this can lead to compatibility issues that

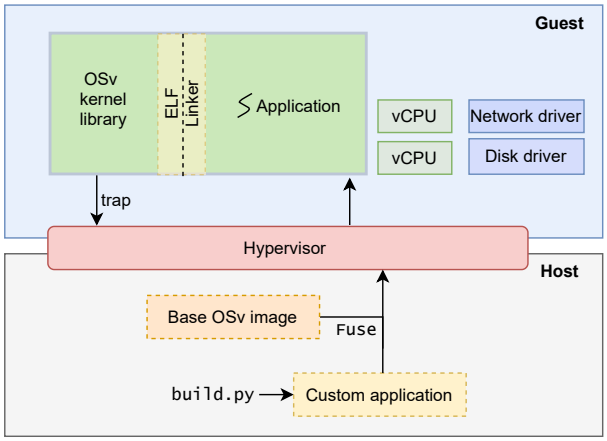

Figure 4.5: Architecture of the OSv unikernel

one may only find out about at run-time.

As OSv is made to be run within VMs using these hypervisors, it can make assumptions that general purpose OSs cannot. Hypervisors expose a relatively simple hardware model and therefore OSv only has to implement a small set of hardware device drivers, such as VGA, SATA and serial ports. For performance reasons OSv also leverages existing paravirtualized drivers such as `virtio-blk`, and more recently (2020) `virtio-fs`.

4.9. Concluding remarks architectures

In this chapter we discussed the architecture of all the isolation platforms. There are both commonalities as well as significant differences between the platforms. In this section we highlight a few prominent differences between these designs.

Every platform devises its own way of interacting with a kernel. Secure container gVisor as well as the unikernel OSv take a relatively unique approach by implementing a kernel in user-space. Despite the fact that these platforms both implement a kernel in user-space, they differ in how they interact with it. gVisor uses the host kernel to redirect system calls to its user-space kernel by using either `ptrace` or KVM, thereby, as a direct result of its architecture, imposing overhead. OSv directly executes system calls because they are regular (dynamically linked) function calls. This can be exploited for performance gains. But even so, OSv requires a hypervisor that handles VM exits whenever the unikernel guest traps out, just like regular hypervisors. Both OSv and the regular hypervisors employ a dedicated guest kernel for nearly everything they do, but ultimately rely on a host kernel (through KVM). The only type of isolation platform discussed that do not introduce another layer of abstraction but rather just an isolation mechanism are containers. There are two potential downsides to this architecture: 1) isolation support from the host kernel is a hard requirement, and 2) there is no defense-in-depth, while the interface to the host kernel is wide. There are always tradeoffs to be made, as outlined at the beginning of Chapter 3.

Despite the tradeoffs, similarities between platforms can be observed as well. For example, the reuse of the `virtio` paravirtualized devices in every virtualized environment, enabling developers to only having to implement the guest-facing backend device drivers for the virtualization platforms. Even more so, the entire project of Kata containers mostly focuses on gluing existing components (Docker's `dockerd` and a hypervisor) together. This results in interfaces that are sometimes incompatible, leading to various types of shims, as also testified by the network architecture of the

platform (which we discuss in more detail in Section 5.14). At the other end of the spectrum we have gVisor, which implements nearly everything from scratch. This includes the implementation of its entire network stack. This naturally comes at the expense of keeping implementation complexity within the project low.

The isolation mechanisms employed within the platforms are thoroughly discussed in Chapter 6.

5

Performance

For measuring overall system performance we perform micro-benchmarks for each platform. These micro-benchmarks stress CPU performance, I/O performance, network throughput, network latency and memory (RAM) performance. We then also perform several real-world workload performance measurements. We perform real-world benchmarks in order to draw conclusions about the general performance of the system, including the interaction between the different subsystems. These benchmarks should thus be indicative of performance measured in e.g. production systems. In addition we have also performed experiments to measure the startup time for each platform, which are particularly relevant in a serverless computing context.

5.1. CPU

In this section we benchmark the CPU performance of the different isolation platforms. This benchmark is representative of any application that is CPU-bound, such as compression and decompression applications, encryption and decryption (prime factorization) applications as well as any other application that spends a substantial fraction of their runtime on performing mathematical calculations (e.g. matrix multiplication). The particular benchmark we have chosen entails loading a 30MB video file[1] into memory, and then encoding that video file from H.264 to the H.265 video codec. For this benchmark we make use of the `ffmpeg` [29] program. The task is executed on guests that have access to 16 CPU cores, and the job itself is executed using 16 threads.

By making use of the different presets `ffmpeg` exposes, we have an instrument to control the speed at which the media is encoded at. In this experiment, we have used the 'slower' preset. This preset, as the name suggests, has a relatively slow compression speed but high compression ratio. Slower compression speed increases overall total compression time, leading to more CPU cycles, reducing the ratio of total running time spent on loading the file into memory. The difference in performance between platforms in this benchmark can thus be attested to actual differences in CPU performance, not I/O (which we found to be the cause of significant variations between platforms in initial runs of this benchmark).

For containers, there should only be a minimum of overhead, since the instructions run on the bare host on the same kernel. Theoretical overhead could lie in the implementation of the bookkeeping of cgroups, but this is not likely to be the case for a mature feature of the Linux kernel. By virtue of hardware-assisted virtualization, the hypervisors should also be able to perform well.

[1] Downloadable from the Blender project website at `https://peach.blender.org/download/`

There are two major characteristics that could impair CPU performance in guests using virtualized CPUs:

1. The CPU performance is impaired because the workload attempts to execute sensitive instructions. Sensitive instructions can not be immediately handled by the guest kernel and are dispatched back to the hypervisor. This incurs an expensive `vmexit` instruction (which typically take over 10.000 CPU cycles [7]), switching address spaces, and once completed a `vmenter` or `vmresume` instruction is executed to switch back. Examples of such instructions are `CPUID` and `HLT` [41]. An example of a workload that execute relatively many sensitive instructions are databases that frequently schedule otherwise idle virtual CPUs [63]. Paravirtualized device drivers try to avoid executing sensitive system calls altogether.

2. CPU scheduling of the virtual CPUs within the hypervisor guest should not be interfering with the host kernel CPU scheduler. One way to reduce this CPU contention is by employing CPU pinning, in which the CPU time the hypervisor exposes to the guest is pinned to a fixed subset of the host's CPU cores. An interesting observation was made in [74], in which CPU pinning was found to be particularly beneficial for I/O heavy workloads.

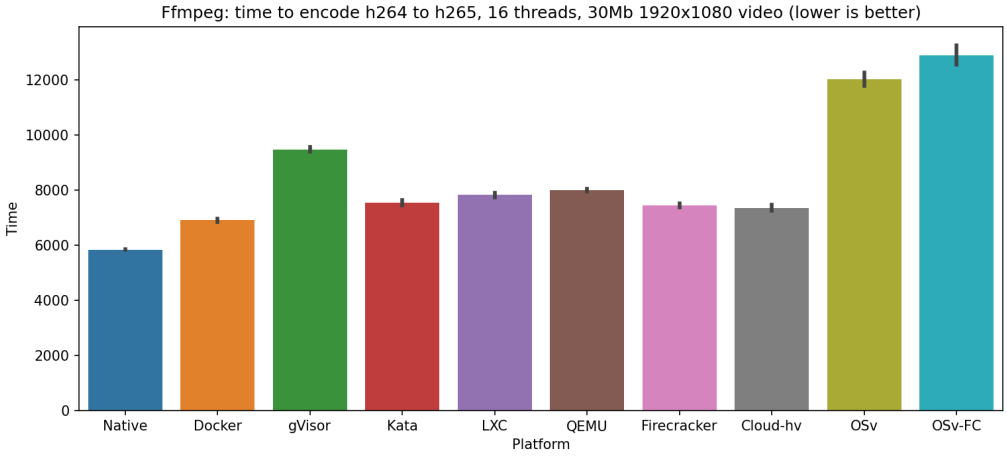

Figure 5.1: ffmpeg video re-encoding CPU bound benchmark, re-encoding a 1080p 30Mb video from H.264 to H.265.

As the results of this benchmark show in Figure 5.1, most of the results end up at around 65000 milliseconds, while some differences between platforms can still be observed. A relatively extreme outlier is OSv, taking up significantly more time to encode the video file. In order to eliminate some potential causes of this discrepancy we have carried out another, highly specific, microbenchmark. This benchmark is part of the Sysbench [78] CPU benchmark, and verifies whether a number is prime using the following simple code (reimplemented in Python, based on Sysbench source code):

```
    for c in range(3, max_prime):
        t = math.sqrt(c)
        isprime = True
        for l in range(2, int(t+1)):
            if c % l == 0:
                isprime = False
                break
        if isprime:
            n += 1
```

Listing 5.1: Prime number verification algorithm, based on Sysbench source code [78] reimplemented in Python

This benchmark stresses a very small basic subset of all CPU capabilities. This benchmark was deliberately carried out to verify whether the overhead exhibited by OSv is inherent to the CPUs on the platform, or due to another more complicated piece of functionality exposed by modern CPUs. The results of this benchmark is given in Figure 5.2. The results show that there is little to no variance among all isolation platforms, including OSv, when it comes to executing simple CPU instructions. Hence, the overhead as imposed by OSv in the `ffmpeg` benchmark likely finds its cause in another set of CPU instructions. Upon inspection of used CPU technologies by `ffmpeg`, all platforms were allowed the same capabilities (specifically: `MMX2 SSE2Fast LZCNT SSSE3 SSE4.2 AVX FMA3 BMI2 AVX2`). Compilation flags as enabled by the GCC compiler on each platform also did not bring any differences to light that should significantly affect this benchmark.

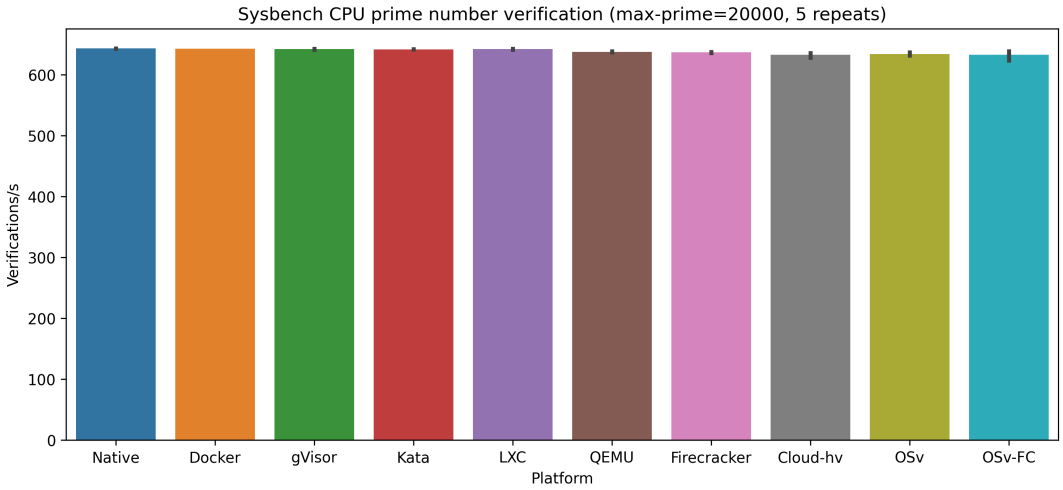

Figure 5.2: Sysbench CPU prime verification benchmark

As such, overhead within OSv must be caused by another factor. For example, although hard to guarantee, it could be attested to the custom implementation of the thread scheduler in OSv (an indication for this is that the gVisor, the other platform that shows a relatively high performance overhead, also has a custom thread scheduler). The scheduler keeps a distinct run queue for each CPU, listing the runnable threads for each CPU. In order to enforce a sense of fairness, the scheduler also employs an active load balancer between these queues. In practice, this means that when there is a run queue that contains more runnable threads than another queue, and a thread asks to be moved or a certain expires, the load balancer might move a thread to another queue. In this case, movement of threads between queues does not make sense since each thread roughly has an equal amount of work to do, and the number of processors and `ffmpeg` threads within the OSv guest are equal, potentially resulting in the overhead as measured in our benchmark. This custom thread scheduler, in conjunction with suboptimalities in implementation in OSv, could be the root cause for the observed performance overhead.

In conclusion, we see that CPU-bound code that exercises a basic subset of all available CPU instructions there is no performance overhead. However, with more complex CPU-bound tasks, such as re-encoding a video using `ffmpeg`, differences in performance overhead become apparent. In particular the platforms that implement custom thread schedulers (gVisor and OSv) appear to suffer a severe performance penalty.

5.2. Memory

There are two important aspects to the performance of memory: raw throughput and access latency. We have evaluated both, the former of which using two different benchmarks. The memory benchmark we used Tinymembench [83] and STREAM [62]. Tinymembench is a relatively simple benchmark which reports both the maximum bandwidth achieved through sequential memory accesses as well as the latency of random memory accesses in increasingly larger buffers. The bandwidth maximum of memory can be measured in many different ways, and indeed, `tinymembench` implements this in 29 different ways. These different ways can roughly be categorized in 3 distinct groups:

1. copy tests: how many bytes can be copied per second between variables residing on the stack. There are different types of copies within this group, such as forward copies (source address increases), backwards (source address decreases, while destination address increases), 2-pass (first, the data is written to a small buffer residing in the L1 cache, and only then it is written to the main memory), and variations thereof.

2. allocating memory through the `memcpy` libc system call.

3. allocating memory using modern CPU features such as sse2, making use of larger CPU registers.

In the results only group 1 and 3 of bandwidth measurements are represented. The second group, allocating memory through the `memcpy` system call, resulted in allocation that was significantly faster than native speed for most of the platforms. This can be attested to caching or lazy allocation of chunks of memory, a recurring problem within benchmarking isolation and virtualization platforms (which becomes most apparent in a later section, Section 5.3). We therefore do not present the results of the second group of memory benchmarks.

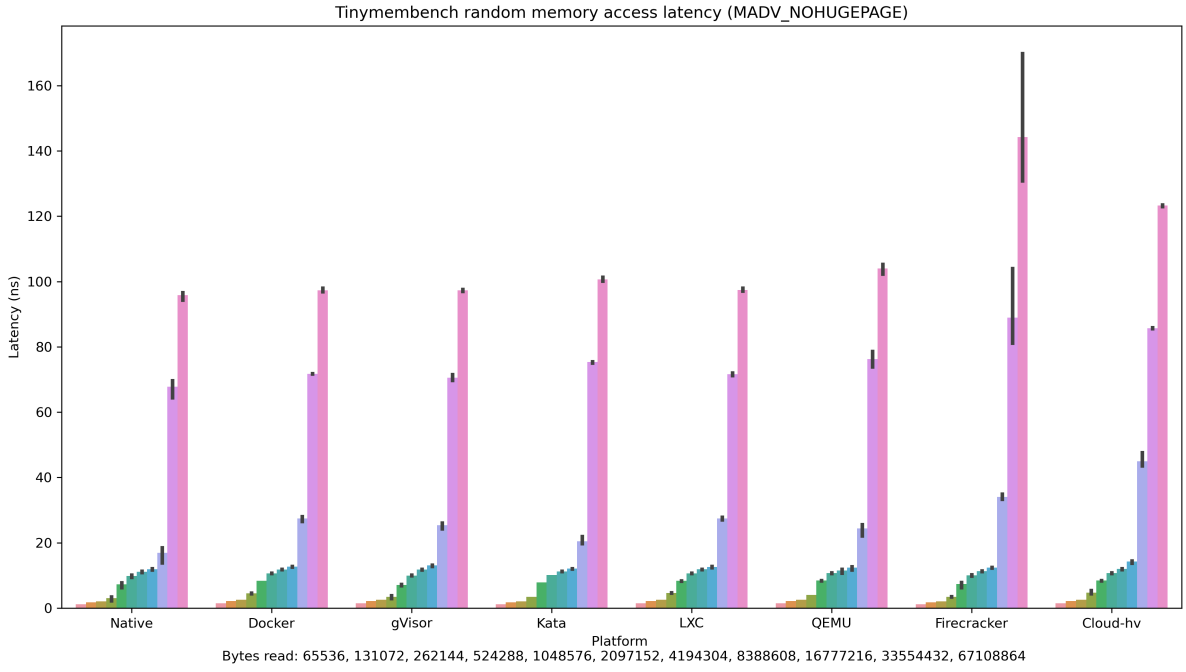

Figure 5.3: Memory latency tinymem benchmark

Figure 5.3 shows the average time for accessing a random element within buffers of increasing sizes. We can see that the larger the buffer is, the higher the latency. This is due to the increasing

proportion of accesses that miss the TLB cache and need to be dispatched to L1/L2 cache, and for even larger buffers to SDRAM. The numbers displayed here indicate the extra time that was needed on top of the L1 cache access latency. The latencies of writing to HugePages are omitted because both Kata containers do not support them, and more importantly, the results are almost equal to those of regular sized pages shown above.

The results sketch an outcome that is mostly consistent between all the platforms, with the exception of the hypervisors. In particular Firecracker both has a higher average latency as well as standard deviation for accesses in larger buffers. The average access latency for Cloud-hypervisor is larger as well, but not to the same extent as Firecracker. These two platforms, Firecracker and Cloud-hypervisor, share substantial parts of their source code in the form of the 'rust-vmm' Rust crate dependency, in which the cause for these higher access latencies could potentially reside.

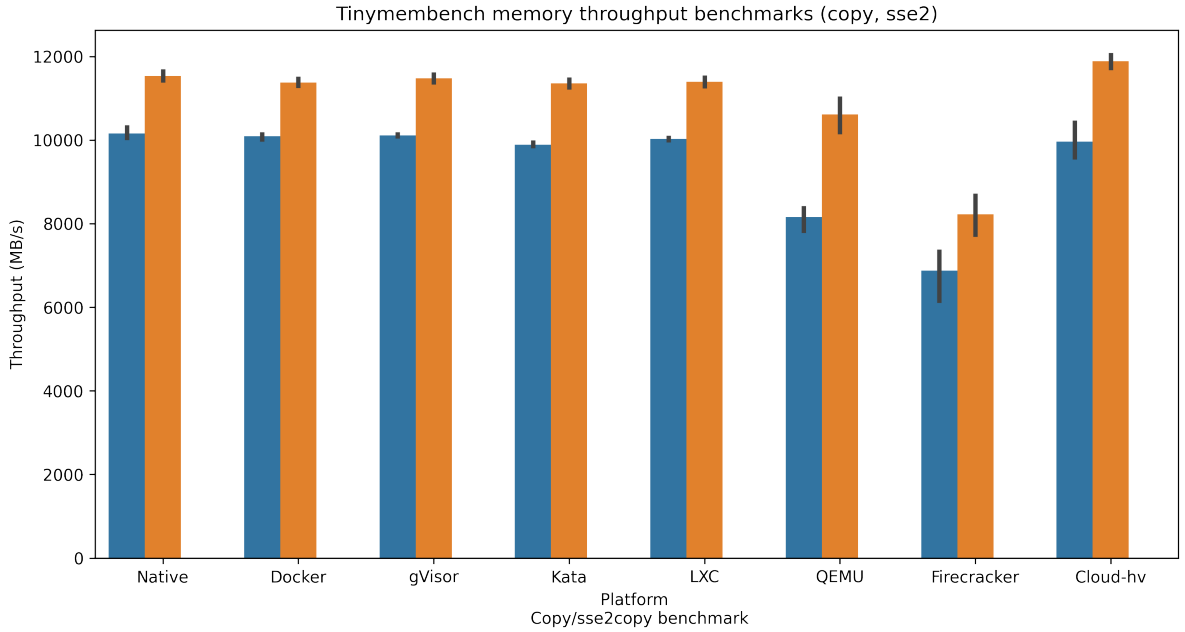

Figure 5.4: Memory throughput tinymem benchmark

We use two benchmarks to measure memory throughput. The first benchmark, in Figure 5.4, is a benchmark that shows how many bytes can be copied per second using both regular as well as sse2 instructions. This benchmark is part of the Tinymembench benchmark. The second benchmark is the popular STREAM benchmark, a simple synthetic benchmark for measuring sustained memory bandwidth by performing simple operations on vectors [62]. The STREAM benchmark consists of 4 different vector operations, but we only present the COPY operation results here as the operations yielded similar relative performance (the other results are provided in Appendix A). The COPY benchmark executes code of the form `a[i] = b[i]`, transferring 16 bytes per iteration, and executes no floating point operations. Both of these benchmarks have a sequential access pattern, meaning that performance is minimized by memory bandwidth rather than latency (as hardware typically prefetches the data that will be requested later on).

In Figure 5.4 and Figure 5.5 we see the results of these benchmarks. The throughput performance is reminiscent of the latency plot at the beginning of this section. All platforms generally perform equal, or close to equal, with the hypervisors underperforming. Although paravirtualization and hardware-assisted virtualization have substantially reduced the overhead of hypervisors,

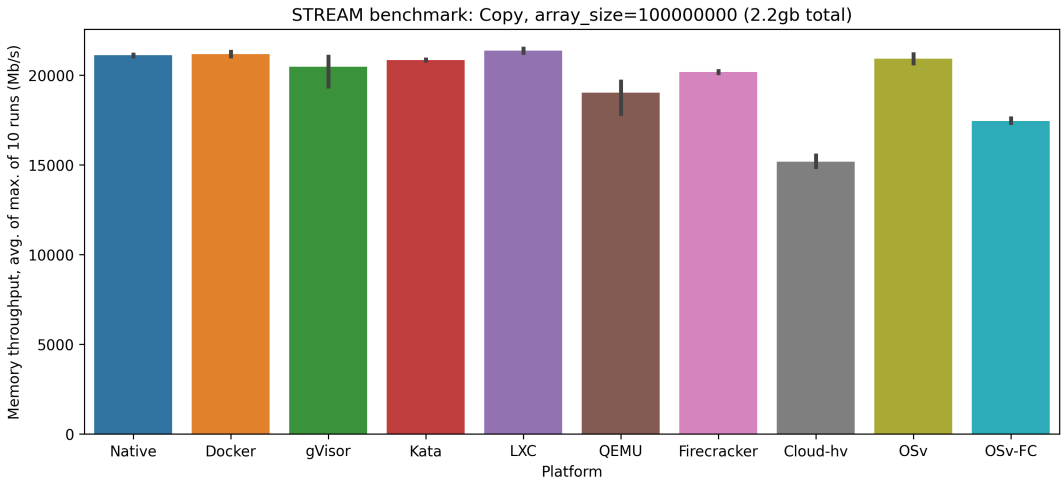

Figure 5.5: Memory copy throughput STREAM benchmark

the additional layer of hypervisor indirection seems to cause overhead. However, there is one remarkable result that contradicts this finding. Kata containers uses a hypervisor, yet is is not victim to the reduced memory latency and throughput. We thus conclude that the overhead is not inherent to the use of hypervisors. Concretely, Kata containers avoids this virtualization penalty by techniques such as the QEMU NVDIMM feature, which provides a memory-mapped virtual device that directly maps between the VM and host, bypassing the intermediary virtualized layer. The I/O subsystem can employ similar techniques, which are discussed in Section 5.3. Another technique that can provide improved memory performance for virtualized guests is Kernel Samepage Merging (KSM) [6]. KSM enables the sharing of memory between multiple processes (like VMs), which increases density, and therefore the reuse of hot pages (for a higher cache hit ratio). Although direct access techniques such as the NVDIMM feature and KSM potentially lead to performance gains, it also weakens the isolation boundary between tenants of the same host (as shown in e.g. [44], in which the authors present a vulnerability introduced by KSM).

In this section we have seen that most of the isolation platforms do not impose significant overhead on the use of the memory subsystem, of which we have quantified both access latency as well as throughput. We draw the following conclusions based on the results in this section:

- All containers, including secure containers, perform on-par with native.
- Although most hypervisor-based platforms exhibit some form of slowdown in both latency and throughput, the Kata container platform is not significantly impaired, despite its use of the QEMU hypervisor. Furthermore, the OSv platform running under QEMU also does not show any slowdown. As such, we can conclude that the usage of a hypervisor does not unconditionally lead to memory performance overhead.
- The memory performance outlier is Firecracker, scoring substantially lower than the other platforms. Cloud-hypervisor shows a similar (although weaker) effect on memory access latency but not for throughput, while the opposite holds for QEMU. This suggests a trade-off between latency and throughput for general hypervisor-based platforms.
- OSv its memory performance is strongly affected by its hypervisor. OSv running under the Firecracker hypervisor underperforms in comparison to OSv running under QEMU, which yields results close to native.

5.3. I/O

We first provide a breakdown of how I/O is implemented in each isolation platform. We then present the benchmarks and analysis.

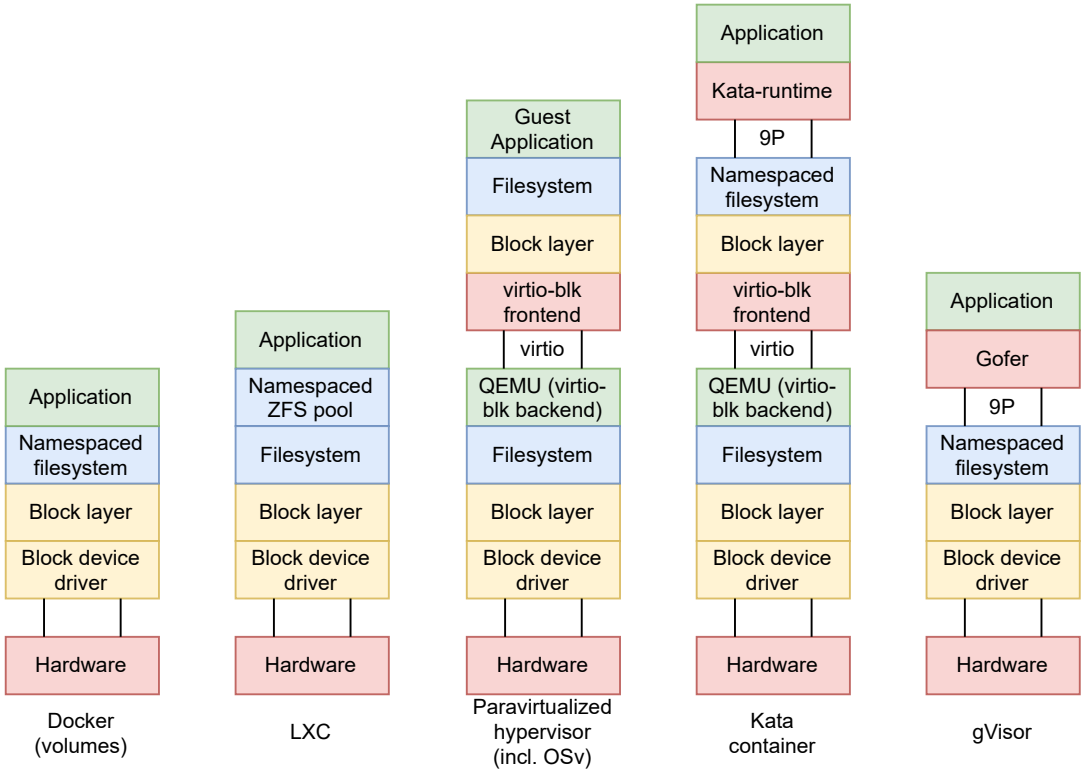

Figure 5.6: Path for I/O requests per isolation platform.

The path a request for I/O, whether it be a read or write request, is different for each isolation platform. We discuss this path per platform, and is illustrated in Figure 5.6:

- Docker: A feature that makes Docker unique is the use of a so-called Dockerfile. This file describes, in an imperative manner, how the image should be constructed. This file is tightly integrated with the layered filesystem Docker uses: each layer represents one instruction in the Dockerfile. These layers are all read-only, except the topmost layer. The Docker storage driver implements what is needed to perform interactions between these layers. For example, with the `overlay2` storage driver, each layer is present in a different folder on the host filesystem (typically at `/var/lib/docker/overlay2`). Every layer contains the files that are different from its lower layer, as well as a symbolic link to this lower layer. This layered filesystem however is typically not used for I/O intensive applications, and for any persistent or performant I/O workloads Docker volumes should be used. A volume is simply a bind mount (‘mount –bind‘) from the container to the host filesystem, constrained by a namespace.

- LXC: Creates a loop device for the ZFS pool it creates. This means that the ZFS pool is presented on the host system as a regular file. It then chroots into the root filesystem folder that is present in the ZFS pool for that specific container.

- Hypervisors: All of the hypervisors make use of virtio, a paravirtualized I/O driver exposed to the guest (called the ‘front-end driver’). The host exposes this paravirtualized driver by an

implementation in user-space by the hypervisor process. Because this implementation is in user-space, there is no driver needed in the host kernel. The image of the guest is presented to the guest as a block device (and can thus contain its own filesystem), while the host only sees a file on its filesystem. This is made possible by making use of the I/O loop device implementation in the Linux kernel.

- OSv: Although the I/O performance of OSv is not benchmarked due to its inability to run the used benchmarking tool, we will shortly discuss how OSv handles I/O. OSv uses a loop device and virtio, just like the hypervisors. As such, the I/O architecture of OSv is the same as that of the hypervisors (as can also be seen in Figure 5.6). The only difference lies in which filesystems are supported. In the case of OSv, the only supported filesystem is ZFS.
- Kata containers: In terms of architecture and technology Kata is not more than just the sum of Docker and the hypervisor. This means that a folder in the host is mounted onto the root filesystem of the container (much like Docker), and within this container, a hypervisor exposes this entire root filesystem to the hypervisor guest through the 9P filesystem.
- gVisor: all I/O requests (and system calls) are reimplemented in the Sentry process. In this reimplementation, all requests are dispatched to the Gofer process, which performs the actual I/O operations on the host using the 9P filesystem.

5.3.1. I/O Benchmarks

For benchmarking the I/O subsystem we use the Flexible I/O tester (`fio`) benchmarking tool [32], version 3.25. Specifically, we use `fio` to benchmark the block I/O performance of the different isolation platforms. By benchmarking on the block level, rather than on the filesystem level, we solely capture the overhead imposed by the actual virtualization mechanisms, rather than a combination of overhead imposed by both the filesystem and block layer.

Fio measures the average read and write throughput by pre-allocating a file two times the size of the amount of memory available to the platform, using `fallocate()`, and then uses this file to write to and read from in blocks of 128kb using the `libaio` I/O engine. For these benchmarks, we start the platform that is being tested, and then attach a separate storage medium through the user interface exposed by that platform. For Docker this could be as simple as passing a bind mount through the `--volume` flag, whereas for LXC this entails creating a new ZFS storage pool on the separate storage medium and recreating the LXC container within this new pool. For hypervisors, the target storage medium is attached as an additional drive and mounted within the guest. The amount of data read/written for each I/O test is equal across the different platforms, in order to keep the chance of anomalous seek times imposed by a larger test file to a minimum.

Since Firecracker does not support attaching extra storage devices, it is excluded for this benchmark. For OSv there is no working implementation of the `libaio` engine, and picking other I/O engines leads to either an unfair comparison or underutilization of the maximum throughput the storage offers. As such, we have left the OSv platform out of the following benchmarks.

In Figure 5.7 we see the results of this 128kb write/read throughput benchmark. Generally speaking, the read performance of Docker, LXC and QEMU/KVM is equal to that of a native platform that does not virtualize anything. The write speeds of these platforms come close to that of native as well, although overhead comes in the form of a higher standard deviation. The other hypervisor, Cloud-hypervisor, performs significantly worse, with lower throughput in both read and write performance as well as standard deviation. This however is not inherent to the use of hypervisors, as QEMU demonstrates. The secure containers gVisor and Kata containers severely suffer

from the extra layers of indirection, in the best case reaching only half of the speeds achieved using other isolation platforms.

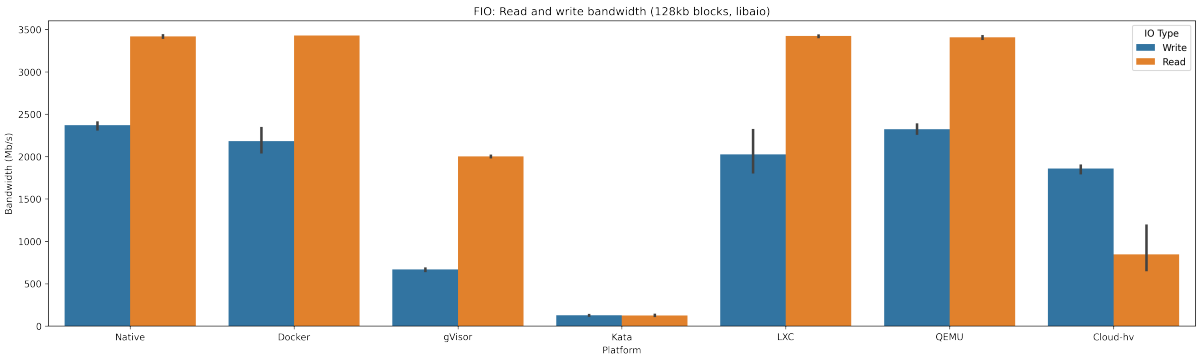

Figure 5.7: Fio I/O throughput benchmark

We found running the I/O benchmarks to be the most difficult to run, due to caching problems with the chosen isolation platform. It is particularly difficult with hypervisors. The main issue when using a hypervisor, both the guest and host have a separate page cache. The Linux page cache stores (parts of) files within memory, leading to greatly accelerated access to these files, since the operating system can read from the in-memory page cache rather than actually having to read from the slower non-volatile storage device. While this is desired behavior in many cases, since it can provide good speedups, it also makes benchmarking all the more difficult.

I/O Benchmarking tools like fio support writing directly to storage effectively bypassing the page cache (using the `direct=1` option). With some isolation platform however, there are two separate kernel instances running (the host and guest kernel), both using their own page cache. Despite fio being instructed to write directly to storage and skip the page cache, due to the additional guest kernel, it is only able to circumvent the guest kernel page cache. The root filesystem of the guest is presented as a block device to the guest by creating a loop device on the host, and flags like `direct` are not propagated properly (and perhaps this is not even desirable, considering the potential security implications), and I/O requests executed within the guest can still be cached inside the host page cache. This also implies that even though the Linux kernel supports dropping its page caches manually (by running `echo 3 > /proc/sys/vm/drop_caches`), if run within a guest, the host page cache is unaffected. This can lead to misleading benchmark outcomes, in which hypervisors outperform the native I/O speeds by a large margin. An effective and rather crude way to remedy this issue is by dropping the page cache on the host manually before each benchmark run.

Most of the platforms roughly achieve the same throughput, leaving gVisor, Kata containers and Cloud-hypervisor behind. When considering the random read latency for the platforms, of which the results are indicated in the Figure 5.8, the relative performance of the platforms is mostly consistent. The hypervisors incur a latency issue that is inherent to the extra virtualization layer these platforms have to go through. QEMU exhibits overhead similar to what is shown in prior research. The newer Cloud-hypervisor platform performs remarkably well in this latency benchmark, but considering its poor throughput performance, it cannot be concluded that it performs better than the QEMU I/O subsystem overall. As for the secure containers, Kata container performs exceptionally poor. Although the gVisor platform is excluded in this particular benchmark, as all its reads got cached even when both host and guest page caches were dropped, it is reasonable to assume that its performance would be similarly lackluster due to the use of the 9P filesystem.

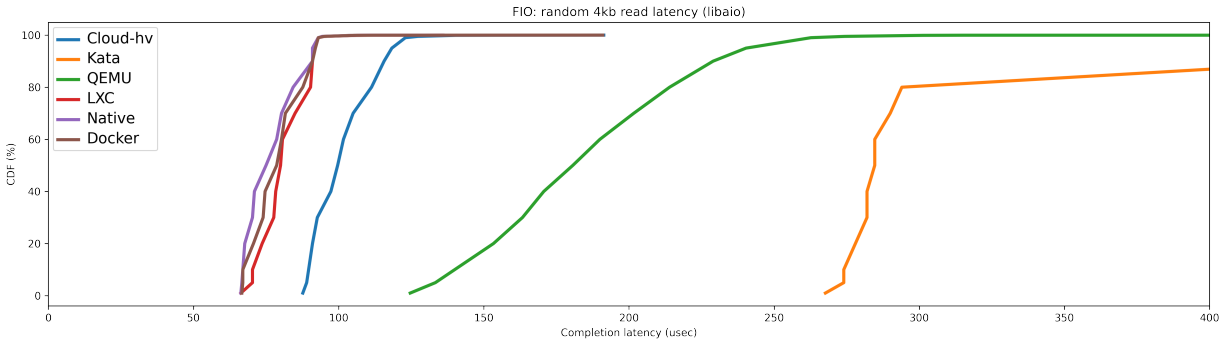

Figure 5.8: Fio randread latency benchmark

5.3.2. Plan 9 and virtio-fs

The relatively poor performance of gVisor and Kata containers can be attributed to the fact that these platforms employ the Plan 9 filesystem. This is a network-based filesystem part of the Linux kernel (and referred to as p9fs within the Linux kernel) [88]. Sharing a filesystem between the guest and host of containers is required to implement a container runtime compatible with the Docker engine (it needs OCI compliancy), for features such as `docker cp` and the aforementioned bind mounts using the '–volume' option.

Although 9pfs is a mature piece of software by most standards, active development ceased in 2012. With the increasing interest in containers in industry, the need for a better and mostly more performant shared filesystem became clear. This led to the creation of `virtio-fs`: a filesystem implemented in FUSE [2] using `virtio` as the transport layer. Since hosts and guests of isolation platforms are not physically separated by a network, an assumption that traditional networked filesystems are built upon (such as the Plan9 filesystem), do not longer hold. The `virtio-fs` filesystem can take advantage of these new conditions, and can gain a significant performance speedup relative to existing networking filesystems. Naturally, we have benchmarked results for this new storage platform as implemented by Kata containers as well.

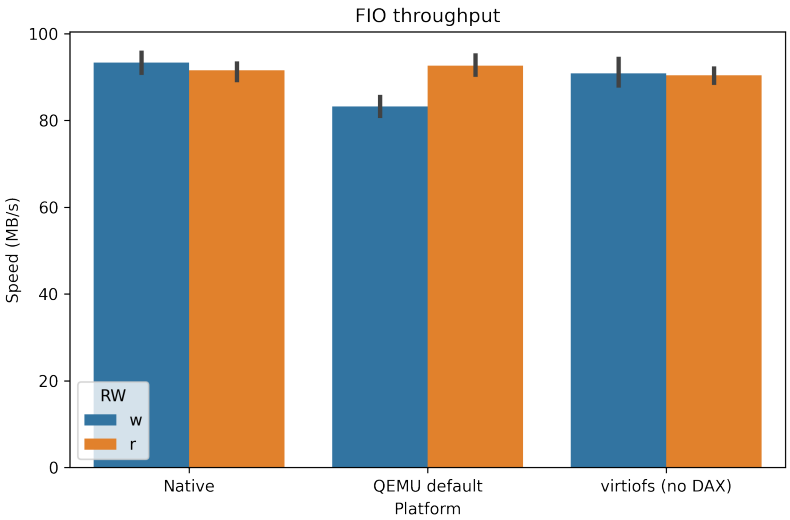

Figure 5.9: Fio randread virtiofs benchmark

[2] `https://www.kernel.org/doc/html/latest/filesystems/fuse.html`

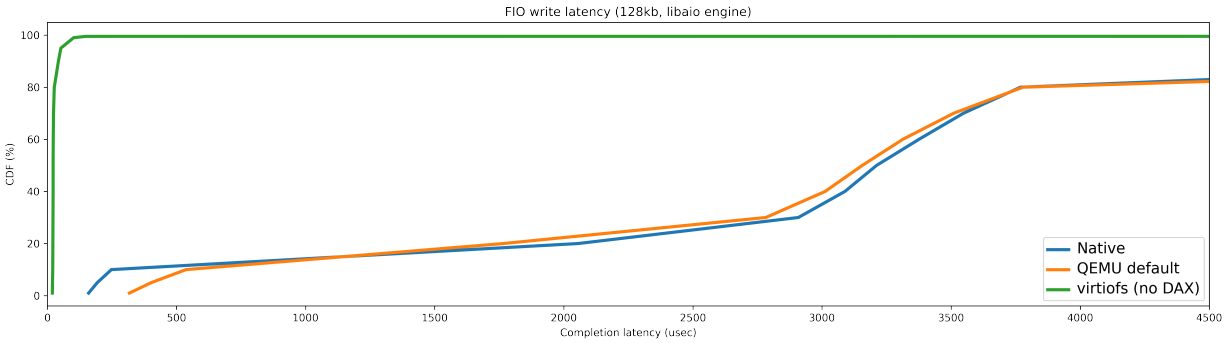

Figure 5.10: Fio randread virtiofs latency benchmark

In Figure 5.9 we see three different platforms we have used to evaluate the performance of `virtio-fs`. The baseline is the read and write performance to the native filesystem, without any virtualization layer in between. The second platform is plain QEMU, with a storage medium shared to the host using the `-drive` flag. The third platform is plain QEMU with the new shared `virtio-fs` filesystem with default settings. We can see that the `virtio-fs` filesystem throughput is roughly on par with native, losing by a margin of around 5%. This is an improvement even over the QEMU with `-drive` flag setup, which presents the passed device as a block device using `virtio-blk` (with its filesystem mounted in the guest).

By investigating the source code of `virtio-fs`, one can see that several aggressive optimizations are used to obtain these rather impressive results. An example of such an optimization is that when writing data from the guest to the host, the data is copied from the guest's memory directly to the host and written from there. This effect is also demonstrated in the figure above, in which the guest (or at least Fio) is under the illusion that writes are completed more or less instantaneously. This particular way of writing to a filesystem is something that can only be realized because there is no actual network between the guest and the host, as networked filesystems like 9pfs and NFS do assume. Another feature of `virtio-fs` is called DAX (for direct access). DAX maps guest file content directly onto a memory window on the host (called the DAX window), enabling the guest to directly fetch data from the host page cache. Note that this sounds similar to the aforementioned clever write trick, but it is in the opposite direction, and is about the interaction with the host page cache.

The clear advantages of DAX are illustrated in the figure above. `Virtio-fs` with DAX enabled achieves read throughput upwards of 9GB/s. Clearly, this involves caching within memory, which is something we have been trying to avoid within all benchmarks. The point we are trying to make here however is that there are great gains to be had in real-life workloads in shared filesystems that do not assume there is a network between the host and guest. Note that this automatically also leads to coherency between the guest and host, since there is only one source of truth about the file contents of shared files: the DAX window.

With these results in mind, in case `virtio-fs` did not exist, we would like to make note of an architecture in which there is no I/O overhead at all that can be trivially implemented in container runtimes. Even in a container runtime that is separated by a hypervisor as is the case with Kata containers. Concretely, if an NFS server is set-up within the guest, and the host functions as a client to that server, you get native I/O performance within the guest. The only downside is that when data is written from the host to the guest the performance is poor in particular with smaller block sizes, as demonstrated in Figure 5.13. The host can also run the NFS server and have the container

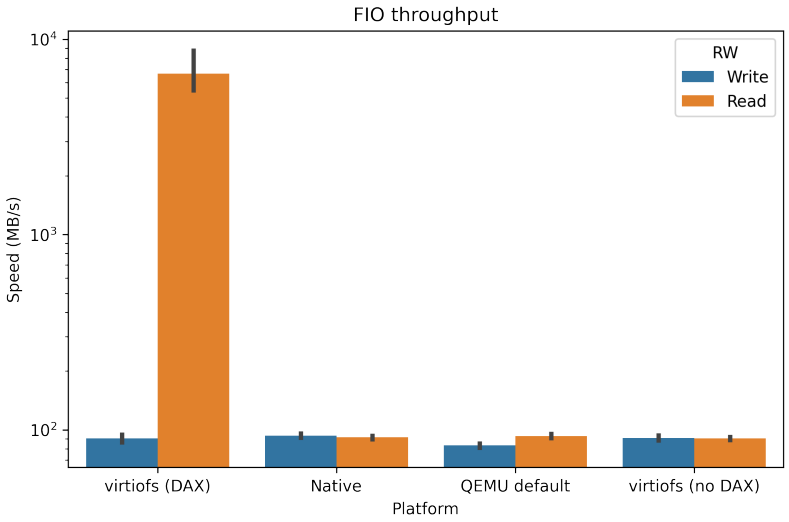

Figure 5.11: Fio virtiofs with DAX throughput (logarithmic scale)

act as a client, but this hampers all I/O performance that is local to the container (since it is, in fact, not local anymore, but rather all I/O has to be dispatched to the host through the NFS protocol). Results of this setup, in which the host acts as the NFS server, are shown in the Figures 5.12 and 5.13 below. Thus, the performance from writing from the guest to the host is shown below, and writing from host to guest is on par with native. But if you were to move the NFS server to the guest and have the host act as client, these speeds are naturally swapped as well (leading to native I/O performance within the guest).

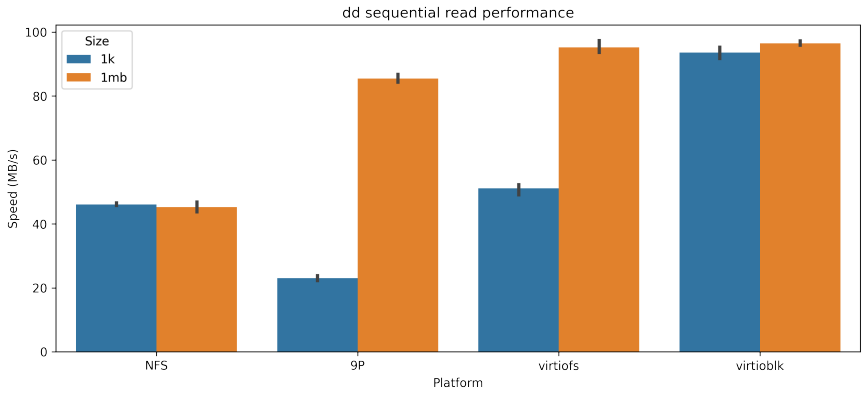

Figure 5.12: dd read throughput

The results show four different platforms, the first three a way of providing a shared filesystem (NFS, 9p and `virtio-fs`) and the last exposing a block device to the guest. Note that the comparison between NFS, 9P, `virtio-fs`, and `virtio-blk` is an apples with oranges comparison because the I/O performed using `virtio-blk` should be able to reach speeds that the other platforms cannot, even when running native, because:

- The writes and reads are to a block device, not a filesystem
- It is not shared, while the other platforms do share their filesystem.

`Virtio-blk` is included here to give an indication of the price that is paid for having a filesystem at all. The block device indicates the theoretical maximum speed you should be able to get writing

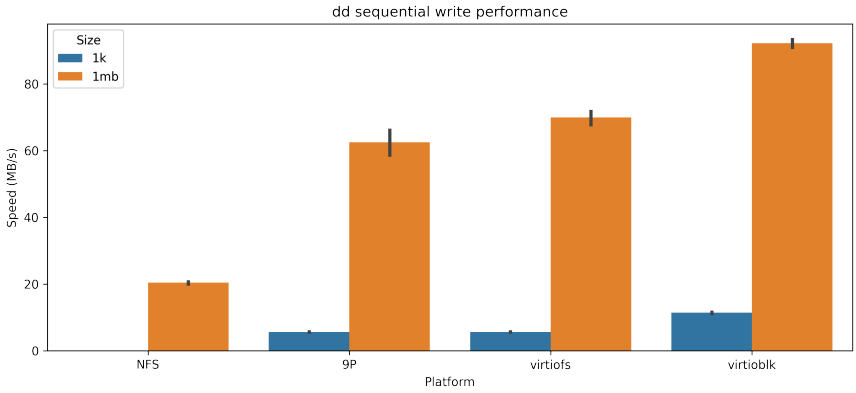

Figure 5.13: dd write throughput

directly to the storage device, and the discrepancy between the block device and the others is the price you pay for introducing a shared filesystem.

Naturally, the shared filesystems perform worse than the block device. Between the three shared filesystems virtio-fs outperforms all other platforms in all categories (read/write and 1kb/1mb block size writes). In this comparison, the 1kb block size write speed for the NFS filesystem is significantly lower, clocking in at around 0.03 Mb/s.

In conclusion, in this section, we have seen that quantifying I/O performance for the various isolation platform proved to be difficult due to the various layers at which caching happens. In terms of quantitative results, we make the following observations:

- The I/O performance of most systems is close to native except for the secure containers gVisor and Kata containers, and for the hypervisor Cloud-hypervisor.
- For Kata containers there is a very promising alternative in virtio-fs. We have also proposed and measured performance of an architecture in which Kata containers uses the mature network-based filesystem NFS, which allows for zero overhead in local writes and reads.
- gVisor performance, in its current form, is severely hampered by the use of both the 9P protocol and separate Gofer architectural component.
- Cloud-hypervisor should get better as it matures, but for now remains the outlier. For this platform, there should not be an architectural bottleneck, as QEMU performs close to native.

5.4. Networking

In this section we first describe how networking is implemented in each isolation platform. Then, we look at the benchmarking results.

The networking setup can be summarized for each platform as follows:

- Docker: By default, Docker containers are attached to a bridge network. The Docker bridge driver automatically installs iptables rules on the host machine. This is set up so that containers on different bridge networks cannot communicate directly with each other. The bridge and the container are connected through a virtual ethernet (veth) pair.
- LXC: the LXC platform supports various networking types. The default networking type is an (L2) bridge, equivalent to the type of bridge used in Docker. When this bridge is created at

LXC installation time, it will also set up a local `dnsmasq` DHCP server and perform NAT for the bridge [57].

- Hypervisors: similar to how the I/O subsystem is implemented, networking is done through the paravirtualized virtio-net driver. Access from the host to the supervisor is done through a TAP device which is exposed as a local network interface to the guest.

- OSv: Like the other hypervisor platforms, OSv makes use of the paravirtualized virtio-net driver. However, unlike the other hypervisor platforms which all run using the linux kernel, OSv does not, and implements its own TCP/IP network stack. This network stack is based on (and originally copied over from) FreeBSD, but was later overhauled to implement Van Jacobsen's 'network channels' design [87], reducing the number of locks and lock operations.

- gVisor: the gVisor platform implements its own network stack called Netstack. As Netstack is part of the Sentry process, this means that everything related to the network subsystem is implemented in the Sentry process. This is unlike how I/O is implemented in gVisor, for which the developers created a separate entity and thus a layer of indirection. In the Netstack implementation, packets are directly written to the network bridge that is installed by Docker.

- Kata containers: Much like Docker, the host networking namespace is connected to the container networking interface using a bridge, with a virtual ethernet pair connecting the two sides. This results in a virtual network interface device being exposed to the container, that is 'plugged in' to the bridge on the host. The local virtual NIC in the container networking namespace is connected to a TAP device in the same namespace, and is connected to the guest by passing this TAP device in the hypervisor invocation. In Figure 5.14 we see the architecture of the various components that make up the networking subsystem in Kata containers. When looking at this figure, one might wonder why the Docker bridge in the Kata containers architecture is not directly connected to the TAP device `tap0_kata` in the guest. This is mostly due to interoperability with libraries that Kata containers depend on. The Kata containers project makes use of the Container Network Interface [23], allowing for reuse of functionality such as creating networking namespaces. As Kata containers assumes Docker is installed, it also assumes the existence of its `docker0` bridge interface. Generally, whenever a regular Docker container is created, the container engine will create a virtual ethernet pair between the host `docker0` bridge and the container networking namespace. Virtual ethernet pairs are commonly used to connect networking namespaces. As such, while it is technically possibly to create two TAP devices on the host and guest and connect them, this is inefficient for two reasons: it results in an architecture where there is an additional TAP device on the host, and interoperability with the CNI library is lost. Whereas if the CNI library is used (and consequently a virtual ethernet pair to connect namespaces), it is possible to use the pre-created Docker network interface that is exposed in the container networking namespace. Hypervisors generally do not implement support for virtual ethernet interfaces but instead use TAP devices. This is why in the case of Kata containers, a Traffic Control (TC) filter is used to connect the container networking virtual ethernet cable to the TAP device. This TC filter should impose relatively little overhead, as will be verified through experimentation in the next section.

5.4.1. Throughput and latency benchmark

For network we take a look at microbenchmarks for both throughput as well as latency. For measuring the network bandwidth we have used the iperf3 benchmark [43], in which the host acts as

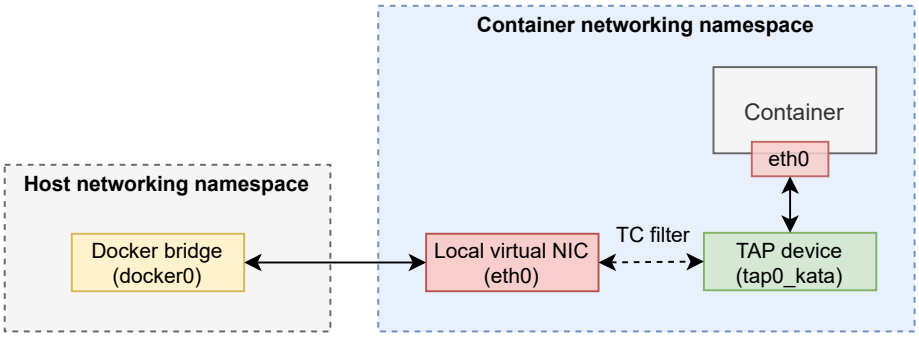

Figure 5.14: Network architecture overview Kata containers

client to the server that is run inside the virtualized guest. For measuring the native benchmark performance, the host machine acts as a server while the client requests are sent from a device that is directly connected to its NIC. The iperf3 benchmark aims to reach the maximum achievable throughput over an IP network. In this context that implies that any score below native indicates overhead from the platform used.

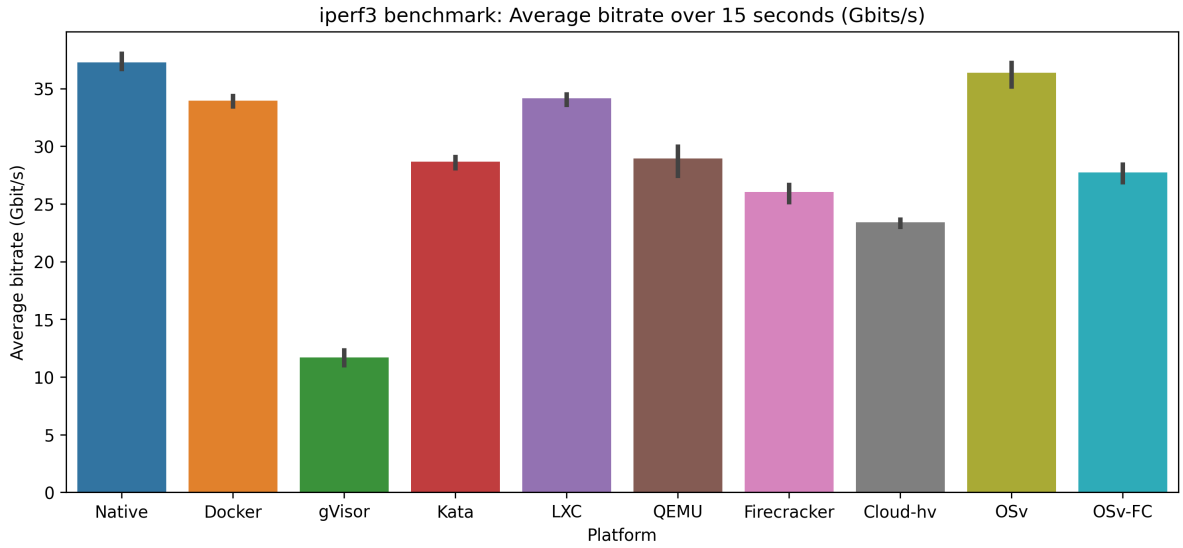

Figure 5.15: iperf3 network throughput benchmark, maximum throughput achieved.

From the results we can conclude, unlike in the memory benchmark, there is always a price to be paid for virtualization (or isolation). The host achieves a mean throughput of 37.28 Gbit/s whereas the second highest, OSv, achieves a mean throughput of 36.36 Gbit/s. The performance advantage of OSv, which runs under QEMU, and a plain QEMU guest is a very significant 25.7% (in the figure, QEMU vs. OSv). The network performance throughput as exhibited here does not necessarily reflect a superior architecture of OSv however, as the results of the benchmark using the Firecracker hypervisor to run an OSv guest only results in less significant 6.53% increase (in the figure, Firecracker vs. OSv-FC).

The results can be further divided into several groups employing different techniques to separate the host network from the guest network, either through namespacing or virtualization. Docker and LXC, use a network bridge approach as mentioned before, and incur a 9.84% and 9.19% performance penalty, respectively.

The hypervisors besides OSv use a TAP device and virtio-net setup, and incur a more severe performance penalty in the order of 25%. The less mature platforms, particularly Cloud-Hypervisor suffers from severe inefficiencies in its implementation, considering the high-level architectural setup of QEMU and Cloud-hypervisor are equal. The Kata containers performance is bottlenecked by the weakest link. Kata container employs both bridges and a QEMU (TAP device + virtio-net) setup. This means that the performance of Kata containers should be equal to the performance of this weakest link, which in this case is the QEMU part of the architecture, and indeed it is.

In this figure gVisor the outlier. As described above, gVisor implements its own network stack. Implementing a network stack from the ground up, however, is not a trivial task, and as a consequence, gVisor does not yet implement all RFCs related to networking, of which many also promise increasing network throughput. Although at this time it is expected that eventually all relevant RFCs will be implemented in Netstack, for now, its performance is not competitive.

The average latency as measured with the Netperf benchmark yields similar results, with the containers using bridges (Docker, Kata containers and LXC) performing very well, followed by the hypervisors. OSv does not outperform every other platform but has slightly lower latencies than the hypervisors. Again, gVisor is a notable negative outlier here, with a 90th percentile response time 3 to 4 times that of its competitors. In this chapter, only the 90th percentile scores are presented. Two other percentile scores, 50 and 99, are provided in Appendix A.

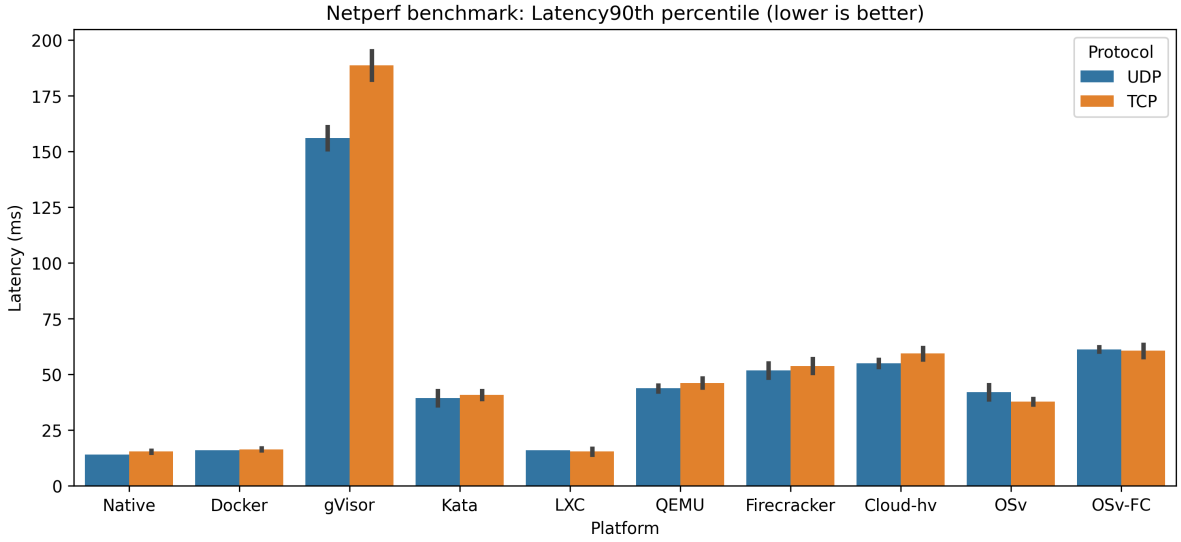

Figure 5.16: Netperf network latency benchmark (90th percentile)

In conclusion, unlike the other benchmarks we have seen up until this point, there are differences between nearly all platforms. We observe the following:

- Docker, LXC and OSv perform the best of all isolation platforms. In particular the last platform is interesting, as it confirms again that the use of a hypervisor does not necessarily lead to impaired (network) performance.
- As we have seen before, the general hypervisors QEMU, Firecracker, and Cloud-hypervisor perform in order of maturity.
- Despite its relatively complicated architecture, Kata containers performs on-par with QEMU.

- The limitations of reimplementing kernel functionality in user-space, as gVisor does, is particularly apparent in network performance, in both throughput and latency.

5.5. Startup

Figure 5.17 illustrates the startup time for every platform in this thesis. Startup time here is the total end-to-end process time, from process creation to termination. The termination of the process is done through a patched `init()` system (for e.g. the hypervisors and LXC) and an 'exit' entrypoint in the containers (which in Docker containers, is scheduled through the 'tini' [82] `init()` system). OSv startup time is measured by invoking it without a program to run, resulting in an immediate shutdown after it completes its boot sequence. As published in [3], the authors of Firecracker have measured booting time as reported by the system itself using a patched kernel. We believe, however, that this is a misleading metric, as the other results are measured from end-to-end, from process creation until termination. When startup time for Firecracker is measured in the same way as is done for other platforms, its performance is, in fact, relatively unimpressive, as we will show in the rest of this chapter.

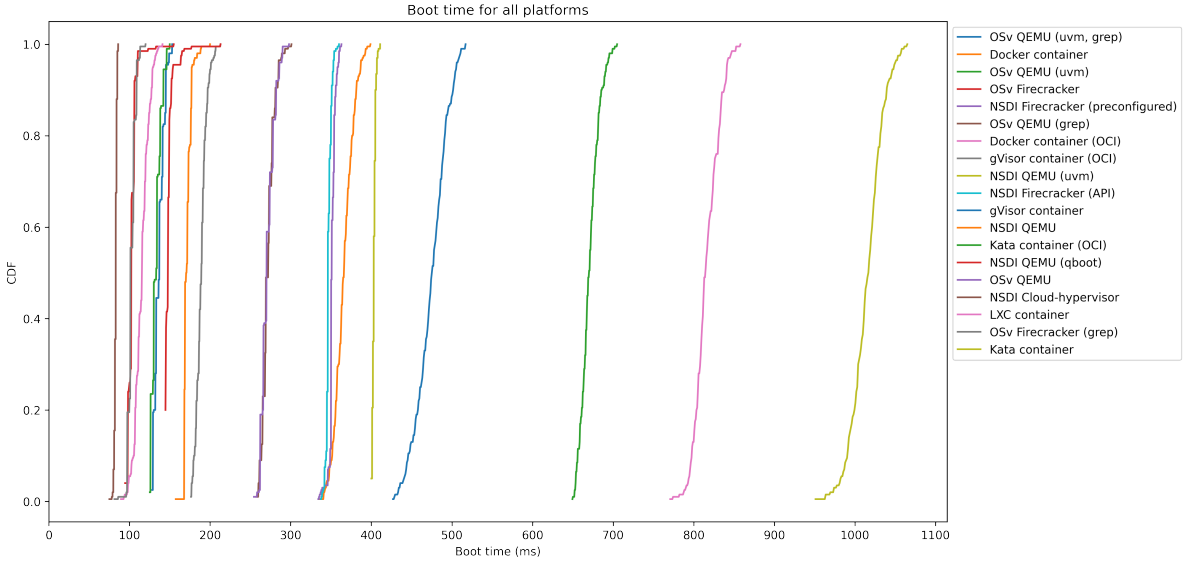

Figure 5.17: Time taken to boot all isolation platforms (CDF)

Although one might assume that including process termination in the total measurement time may not be ideal, the alternative would be to use measurement methods which can not be applied to every single platform. Moreover, in practice, the overhead for process termination was minimal, as we found out through experimentation. The results will be presented in separate parts in the rest of this section, since the results of all platforms and its variations are too numerous to draw clear conclusions from one figure.

For the startup time taken for the container runtimes in Figure 5.18 we look at Docker, gVisor, Kata containers and LXC. There are two variants for each container runtime, one invoked through `docker run` and one by directly invoking the runtime with an OCI bundle ('Docker image'), indicated by *-oci. LXC has no corresponding way to be directly invoked, and as such, there is only one LXC variant. We can see that plain Docker clocks in at an average startup time at just over 100 ms, closely followed by gVisor. Around the 400 milliseconds mark, we again see these two platforms. The difference between their respective OCI counterparts is the overhead for starting and stopping

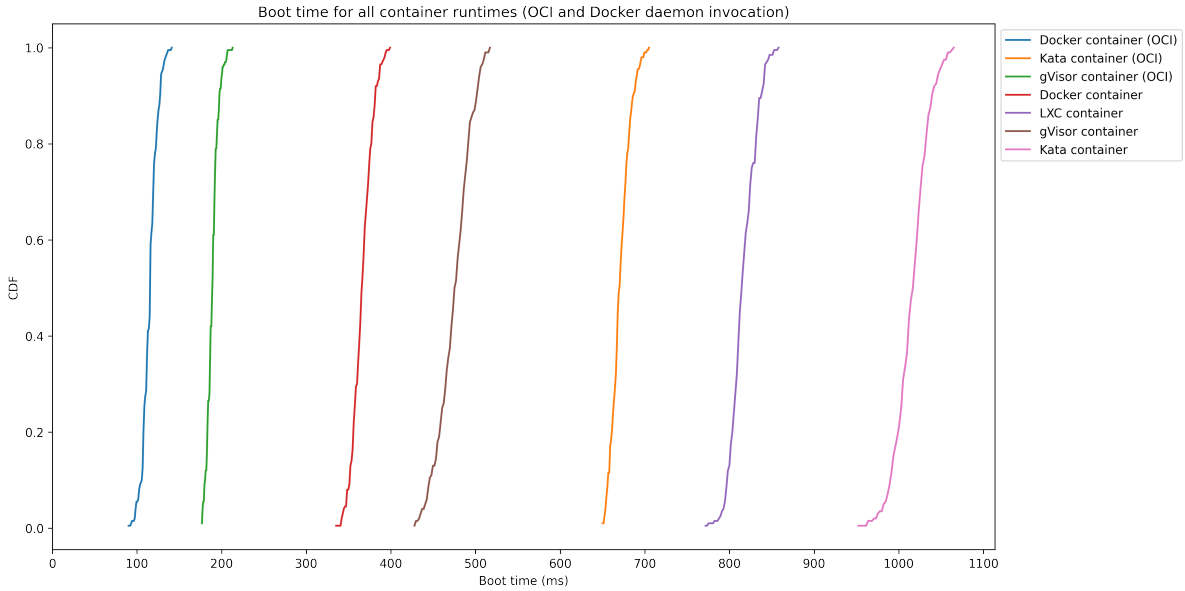

Figure 5.18: Time taken to boot container runtimes (CDF)

the containers through the Docker daemon. Then, we see Kata containers requiring a startup time of 650 milliseconds. For Kata containers startup is more complicated, since the OCI bundle root filesystem first needs to be transferred to the hypervisor, then within that hypervisor the supervising 'kata-agent' process proceeds to pivot (chroot) to that OCI bundle root filesystem, before it can run invoke any commands. Moreover, it also needs to set up other required components, such as the 9pfs (as discussed in prior Subsection 5.3.1). This all leads to the results as shown in the figure, where the Kata containers startup time is more than just the sum of the Docker and hypervisor startup time.

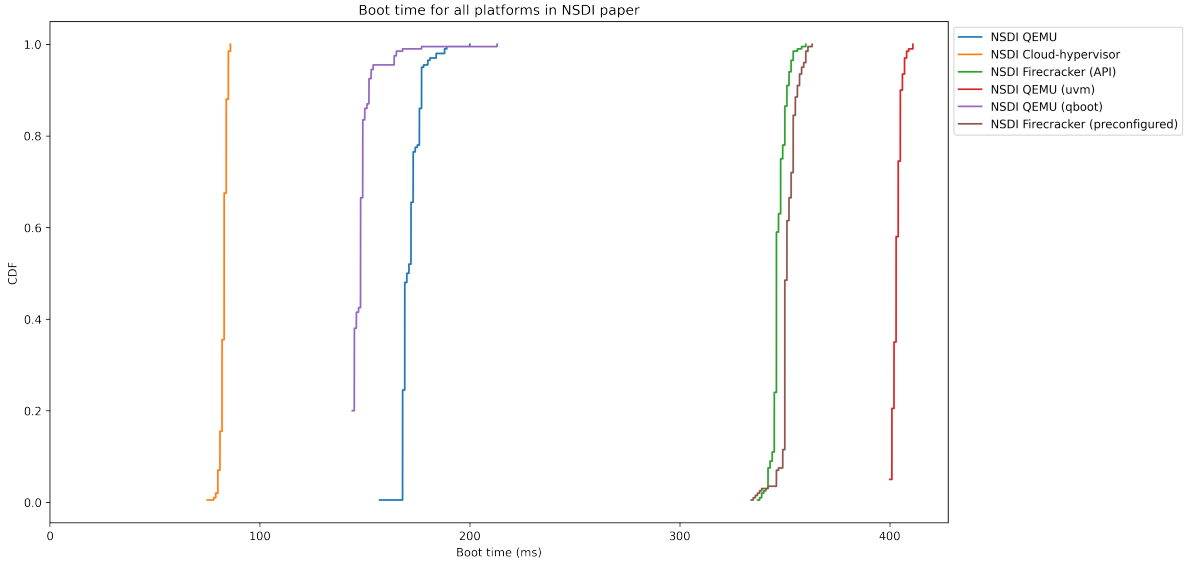

Figure 5.19: Time taken to boot hypervisors (CDF), replication of work in [3]

Figure 5.19 shows the startup time for the different hypervisors booting with the same kernel and root filesystem. As mentioned before, the `init()` system is patched to immediately quit as

soon as it starts. The fastest hypervisor, shown at left side of the Figure, is Cloud-hypervisor, significantly outperforming the other hypervisors, followed by QEMU (both plain QEMU and the QEMU minimal `qboot.bin` BIOS firmware). The slowest hypervisor is QEMU with the `µVM` device model, as inspired by Firecracker. In theory this should lead to faster startup times (with fewer devices to manage, and no BIOS at startup), but in practice for this version of QEMU it only leads to increased startup time.

At around 350 milliseconds the Firecracker hypervisor appears. This is an interesting result, since in [3] the authors conclude that Firecracker has the fastest startup time of all the hypervisors. We consider this conclusion skewed, as not the actual end-to-end (process creation to termination) time is measured, but the time taken to write to a special device during boot time, by using a patched kernel. We do not consider this an apples to apples comparison, as it does not take into account what happens before the kernel is launched, as well as anything that happens after the kernel prints this time. The true time needed to start Firecracker, in our experiments, is significantly higher than its hypervisor competitors Cloud-hypervisor and QEMU.

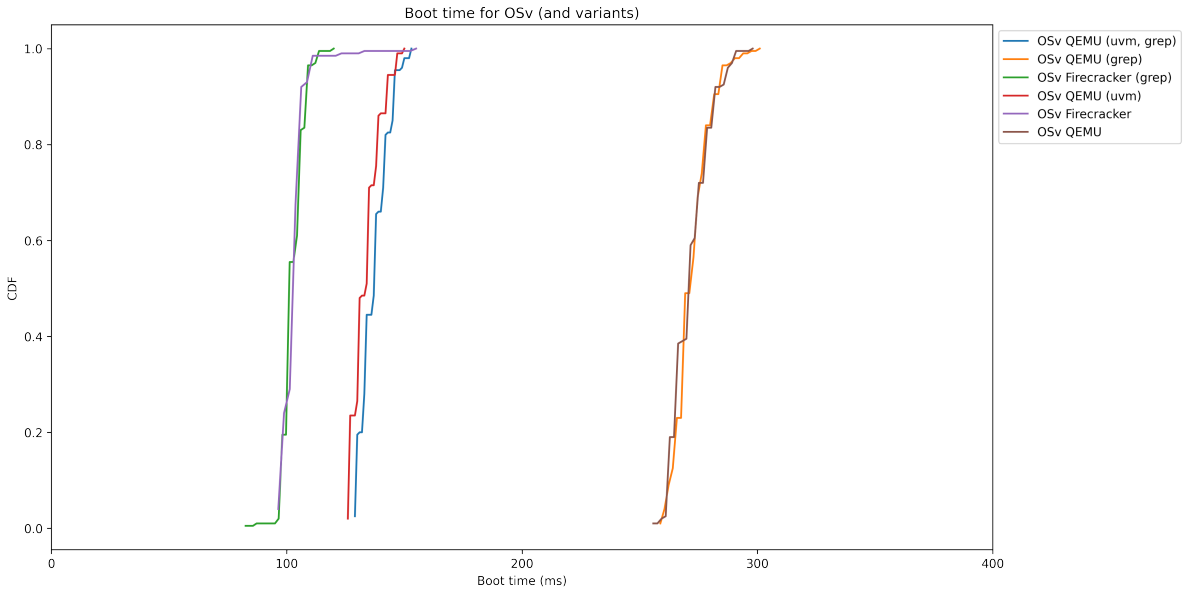

Figure 5.20: Time taken to boot OSv under several hypervisors (CDF)

The results in 5.20 indicate the OSv boot times using different hypervisors. We used two main methods two measure this startup time: end-to-end (as in our measurements above) and stopping the measurement when a specific line of text is printed to `stdout`. As we can see, those two variants of each platform setup are almost superimposed on top of one another, further strengthening the credibility of our way of measuring end-to-end. An interesting observation from these results is that the results are almost opposite of the prior hypervisor results (in Figure 5.19): Firecracker is the fastest, QEMU `µVM` ranks second, and at last place we see regular QEMU.

In conclusion, we find that most platforms are able to boot and exit within 200 milliseconds. We make the following observations:

- All containers are fast to boot, with the exception of Kata containers, which typically takes over 600 ms.

- Firecracker, despite its focus on serving the serverless computing paradigm, boots the slowest out of the three hypervisors. Cloud-hypervisor is the fastest. Boot time depends heavily on the

used machine model, as QEMU with the uvm machine model is (unexpectedly) the slowest out of all.

- Unikernels (OSv) are faster to boot than regular Linux-based images, generally as fast as containers. Booting OSv images using different hypervisors has a significant effect on boot-time.

- Measuring boot-times end-to-end (from process creation to termination) using default tools such as `grep` is as accurate as other ways (e.g. by customizing `init()`). Overhead for this measurement technique is negligibly small across the various platforms.

5.6. Real-world benchmarks

In this section we will consider benchmarking the performance of the various isolation platforms by means of real-world application workloads, in contrast to the previous micro-benchmarks that stress one specific subsystem of the platform.

5.6.1. Memcached

Memcached is a high-performance key-value store [31], often used as an additional caching layer in between e.g. a web server and a database. It can store small chunks of arbitrary data in memory and does never materialize any of its content to disk. We benchmark the performance of Memcached on each platform using the YCSB (Yahoo! Cloud serving benchmark), a popular framework for benchmarking different key-value and 'cloud' serving stores. Specifically, we use the 'workload a' preset of YCSB, a mix of 50/50 reads and writes, behaviour exhibited by e.g. a session store recording recent actions.

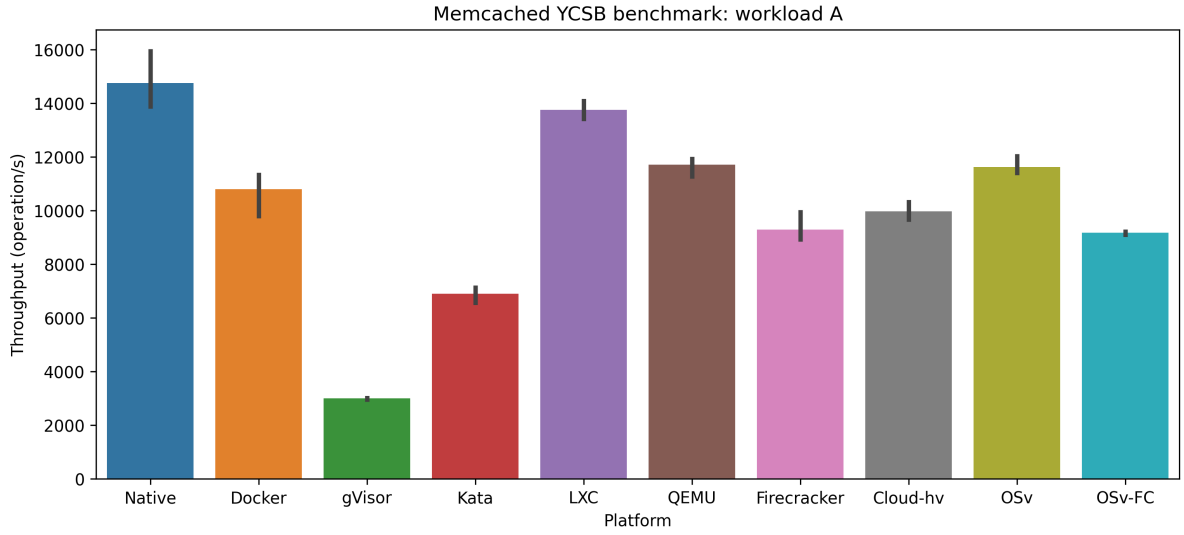

Figure 5.21: Memcached YCSB benchmark

Benchmarking using YCSB and Memcached implies stressing the memory and networking subsystems. As a reminder, the hypervisors underperformed in the memory and network microbenchmarks (the less mature the hypervisor the worse), and gVisor in particular did not fare well in the network microbenchmark. We see these prior results reflected in the Memcached benchmark: the newer hypervisors perform worse, and overall the regular containers (in particular LXC) perform very well. The result of Kata containers is surprising, since the microbenchmarks of network and memory for Kata containers would not suggest a score significantly lower than most of the other platforms. The gVisor Memcached score, although poor, can be attributed to its network perfor-

mance. In Appendix A the performance during the insertion stage for each platform is presented, but performs nearly identical to that of 'workload a' as shown in Figure 5.21.

5.6.2. Mysql

Mysql is a well-known relational database. In combination with the Sysbench [78] benchmarking tool (version 1.0.20), we stress the isolation platforms running MySQL version 5.6.45 using the `oltp_read_write` benchmark. This benchmark stresses the memory, filesystem and networking subsystems. It initially stores 1 million records into 3 tables, and then consecutively executes a `SELECT`, `UPDATE`, `DELETE` and `INSERT` SQL query. We call the combination of one of each of these queries a transaction. We perform this benchmark for every platform with an increasing amount of threads, starting at 10 up until 160 threads.

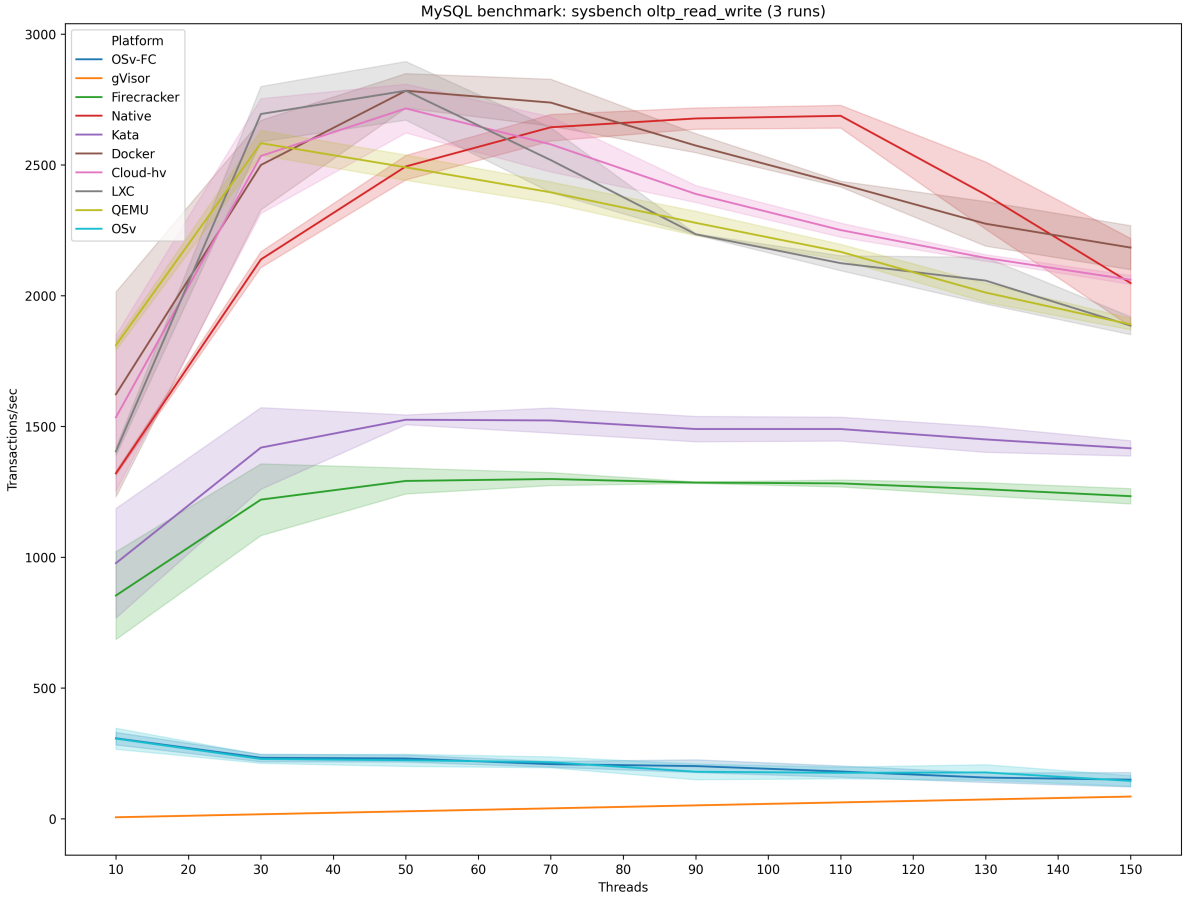

Figure 5.22: Mysql benchmark

The results of this benchmark are shown in Figure 5.22. There are numerous interesting observations to be made from these results:

- For nearly all platforms, the number of transactions per second peaks at around 50 threads, after which thread contention appears to impair overall performance. The native platform peaks at around 110 threads instead, yet does not deliver a significant performance increase over the isolation platforms.
- There are roughly 3 groups in which the platforms can be divided. OSv (and OSv-FC, superimposed on top of OSv) and gVisor severely underperform. It is likely their custom thread

allocators are to blame for this impaired performance, as these are the only two platforms that do not reuse existing mature thread implementations. The lack of any effect when varying the number of threads is also indicative of this. Moreover, as for gVisor, the high network latency as shown in Section 5.4 undoubtedly worsens performance.

- The second group consisting of gVisor and Firecracker yield performance around half of that of most other platforms. With complex real-world benchmarks like these, it remains difficult to say exactly precisely what causes this lower performance. For Firecracker, high memory latencies as demonstrated in Section 5.2 could be the root cause (as a subset of records is kept in memory during the benchmark). As for Kata containers, the relatively high I/O latency as shown in Subsection 5.3.1 could be the culprit.

- The third group, consisting of the remaining platforms, all perform alike. Due to the wide error bands (which did not shrink upon carrying out additional runs) it remains hard to tell precisely which platform performs better or worse than others.

As mentioned before, drawing strong conclusions based on data obtained through this Mysql benchmark is difficult. The wide errors bands in the plots seem inherent to the use of this particular combination (of versions of) Mysql and Sysbench, unlike what is exhibited in other work, such as [28]. Moreover, some platforms severely underperform, without being able to pin-point why this should be the case. Upon re-running this experiment, with the hope of getting more telling results, similar undecisive results were obtained.

5.7. Performance conclusions

RQ1: *Where do the new types of virtualization techniques position themselves on the spectrum of performance overhead incurred?*

- Container platforms, such as Docker and LXC, typically showcase near-native performance. Out of all the isolation platforms, containers perform the best, and typically also have a low start-up time.

- Hypervisors always impose overhead in their networking and memory subsystems. Other subsystems, such as I/O and CPU do not necessarily exhibit this overhead, although it depends on the particular hypervisor that is used. Generally speaking, the more mature the hypervisor, the lower the overhead.

- Secure containers display the weakest performance of all isolation platforms. The networking and memory subsystems perform near-native (as with hypervisors), but in particular, I/O performance suffers. The primary reason for this being the use of networked filesystems, although improvements with the likes of `virtio-fs` are promising. Moreover, this overhead could also be remedied by using networked filesystems that use local writes within the container.

- The OSv unikernel generally performs well, although its performance is hard to quantify due to instabilities and incompatibilities with the chosen benchmarks. Start-up times are comparable to containers.

The resulting spectrum of overhead imposed is graphically represented in Figure 5.23.

Besides the general statements about the isolation platforms categories, we would also like to note the following observations:

- Firecracker is not the fastest to boot in our experiments, unlike what is presented in [3].

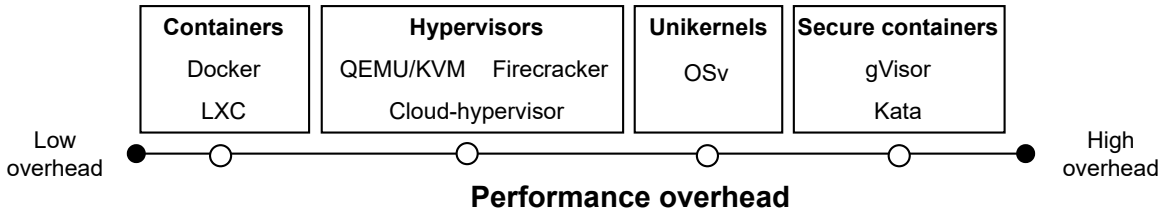

Figure 5.23: Performance overhead per isolation platform category

- The tagline of the Kata containers project "Speed of containers, security of VMs" generally does not hold in our experiments. Performance of various subsystems, with in particular I/O, is weak in comparison to hypervisors.
- I/O benchmarking in general is difficult due to the multiple layers of caching. Most of these difficulties can be overcome by manually clearing the page cache at both the native and virtualized level, although this method is not infallible (e.g. this method does not work for the gVisor platform).
- Pursuing the development of solutions and protocols that are specifically made for isolation platforms has proven to be fruitful. For example, using `virtio-fs`, which only assumes separation, not necessarily physical separation, can bring significant optimizations for all isolation platforms (`virtio-fs` initially was developed for plain QEMU only, but now also finds support in Kata containers [89] and OSv [86]).

6

Security

Another critical property of the isolation platforms we consider is security. We will approach the security of the isolation platforms in two ways, the horizontal and the vertical attack profile. The horizontal attack profile (HAP) is a quantifiable metric that indicates how wide the interface from the guest to the host is. The vertical attack profile (VAP) aims to reason about security architecture (defense-in-depth), and does not lend itself to numerical measures, but instead will be discussed in terms of commonly associated security properties of the used isolation mechanisms by considering past research and relevant exploits. The following paragraphs will explain both attack profiles in more detail.

The horizontal attack profile, a term originally coined by IBM's James Bottomley [12], is a quantitative approach to measuring security. Broadly speaking, measuring the HAP means multiplying the amount of code traversed by a certain bug density of the domain that you're measuring in. Concretely, in this text, that means measuring how many host Linux kernel functions are hit while running different workloads inside guests using the different isolation platforms. Multiplication of bug density is not needed as everything is measured within the domain. We extend the HAP metric by not only measuring how many functions are hit, but looking at which functions are hit. Functions are scored based on whether they have been part of past CVEs. As CVEs only say something about vulnerability and not necessarily exploitability, we have recreated a CVE vulnerability to exploitability model to establish this relationship. This model is referred to as the EPSS model [45]. The model enables us to look at which functions are hit and then determine their exploitability, weighing functions that are more likely to be exploited heavier than those that have a lower exploitability, or not at all, if they are not associated with any relevant CVEs.

For instance, we measure the HAP of the QEMU hypervisor with KVM acceleration enabled. We record which functions get executed by the `qemu-x86_64` process while executing using a kernel module that can trace which host kernel functions get hit during execution. We perform a workload inside the guest that is being traced. We shut down the guest, and store all of the executed kernel functions. We process this output by only keeping the unique functions. We feed these functions to our EPSS model, which outputs for each function the likelihood of exploitation. A significant portion of the functions is likely not to be associated with any CVE, and we assign a score of 0 for that function. For the remaining functions, we set the score as the likelihood as predicted by the EPSS model for that function. We sum up all of these scores, and that gives us a final score for the HAP of that platform.

In this chapter, we will first discuss the results we obtained by tracing all kernel functions that

are hit during execution. We then describe in detail how we extend the HAP, and how we have recreated the EPSS model, and the results of the extended HAP model that uses it. As the extended HAP measure describes the chance of exploitability based on the width (i.e. the sum of the score of the traced functions) of the interface between the guest and the host kernel, it does not account for the possible defense-in-depth that is employed by the isolation mechanism. We call this defense-in-depth the vertical attack profile, and we will qualitatively consider the isolation mechanisms and defense-in-depth levels that different isolation platforms offer. We do this by analyzing the commonly associated security promises these isolation mechanisms offer. We base this analysis on past research and past exploits. This analysis is provided in the remainder of this chapter.

6.1. Horizontal attack profile

The horizontal attack profile quantifies the width of the interface between the host and guest. The more (unique) calls between the guest and the host are executed, the wider the HAP is. In a regular OS the width could be measured through the amount, and perhaps specifically which, system calls are made to the kernel. This is not possible due to various approaches and implementations of isolation platforms, in which some choose to run a guest kernel, some reuse the host kernel, and some implement a kernel in user-space. Yet, with all of these platforms in our setup, there is always a host kernel present. A means to measure the HAP is thus to measure the system calls made to the host kernel during execution of the various isolation platforms. Our host kernel, the Linux kernel, provides ways to measure activity at a various levels of granularity: using the `ftrace` functionality of the kernel it is possible to not only record system calls, but also which kernel functions get executed during execution of specific (hypervisor, container, etc.) processes. As `ftrace` exposes a complex interface through its custom `tracefs` filesystem (which requires writing to specific files to perform configuration at `/sys/kernel/tracing`), we employ the more user-friendly `ftrace` frontend `trace-cmd`[1].

6.1.1. Tracing results

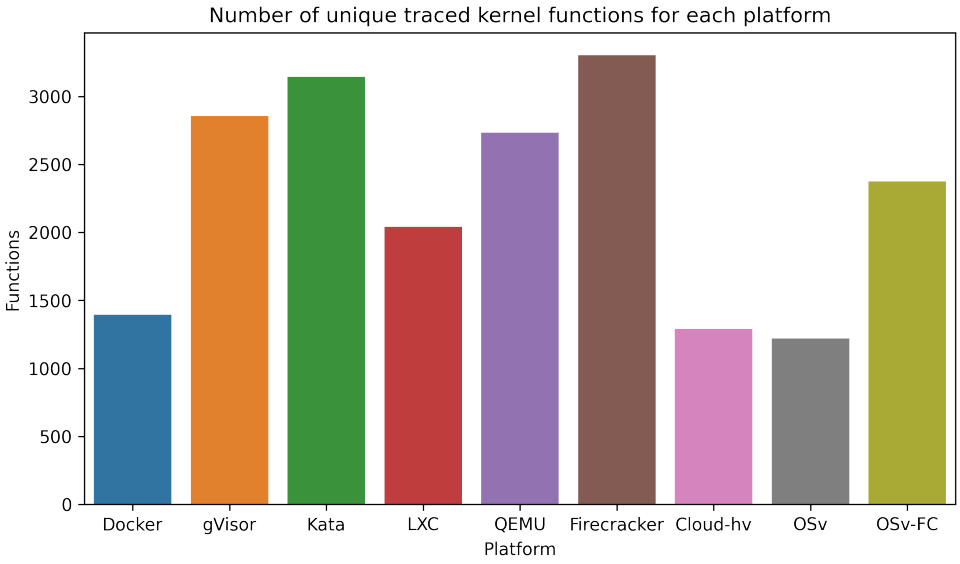

Figure 6.1: Number of unique host kernel functions executed per platform

[1] `https://trace-cmd.org/`

The total number of unique functions executed in the host kernel invoked by the isolation platforms is graphically presented in Figure 6.1. The workloads run during tracing are the CPU, memory and I/O benchmarks from the Sysbench benchmarking suite [78], the iperf3 networking benchmark [43], and simply starting the platform and shutting it down after 1 minute. A breakdown of the number of traced function on a per-workload basis is provided in Figure B.1. Note that this is the plain HAP metric, not including the extension which will be presented later in this chapter. There are quite a few remarkable observations that can be made:

- Firecracker, the isolation platforms that advertises itself as running lightweight VMs and having a minimal device model, calls into the host kernel most often of all platforms. A wider interface between the host and guest is indicative of potentially weaker security. This is quite a remarkable finding as it is the opposite of one would expect: the allegedly heavy-weight general-purpose QEMU hypervisor is expected to expose a much wider interface to the host than the highly minimalistic and specific Firecracker, but the contrary is the case.

- Cloud-hypervisor invokes very few function calls in the host kernel, which is surprising given the results of the other two hypervisors. As both the techniques as well as architecture of Cloud-hypervisor overlap with the platform with Firecracker, this could be attributed to the work-in-progress status of the project, not fully supporting all functionality that the other hypervisors do.

- The secure containers, gVisor and Kata containers, have relatively high numbers, especially compared to the regular containers. For Kata containers this is to be expected, given that Kata containers starts its own hypervisor (albeit with a stripped down Linux kernel). The number of functions for gVisor is higher than expected; apparently, based on these results, reimplementation of a kernel in user-space does not necessarily lead to fewer calls to the host kernel. In this case it mostly provides defense-in-depth, not in width (see Section 6.2).

- OSv, in particular given the fact that it uses a hypervisor, executes host kernel functions sparingly. OSv in this sense fulfills its promise of being secure, and more importantly, from this we can conclude that *a wide HAP is not inherent to the use of a hypervisor*.

- Noteworthy is the near doubling of functions executed when OSv is run under the Firecracker hypervisor (instead of the default hypervisor QEMU). This is likely due to the experimental nature of support for the Firecracker hypervisor, similar to the reason why Cloud-hypervisor executes few functions.

- The aim of LXC to provide feature-complete system containers is apparent in the HAP results as well. The total number of functions executed is high relative to Docker containers.

These tracing results give us a foundation to build upon for extending the HAP metric in the following sections.

6.1.2. Extending the HAP

This original HAP method as shown in the previous subsection strictly describes the width of the interface, but assumes a uniform distribution of both bugs and exploits across all functions. As such, the HAP in this form fails to capture potential security risks relevant to the recorded interface. We extend the HAP measurement by associating the functions recorded with a certain *score*. This score is calculated using the EPSS model [45], a logistic regression model that takes input from an aggregation of CVEs, proof-of-concept exploits, exploit databases, and presence of relevant tags, emitting a score for any given CVE. The score is the likelihood of exploitation within the next 12 months, and

for our purposes, establishes a relationship between vulnerability and exploitability. We have recreated the EPSS model through aggregation and integration of a wealth of scraped online resources. Specifically, we use the following datasets:

1. We collect all CVEs that are associated with a Linux kernel git repository 'fix' commit. These commits indicate that a certain CVE vulnerability was fixed.

2. We collect all CVEs that have their vendor listed as Linux from `cvedetails.com`. These CVEs also have their required access (to reach the vulnerability), vulnerability type and CVSS score listed. Note that these CVEs also have a severity score associated to them, but these scores are not a good indication of likeliness of exploitation [45].

3. We collect the number of references for each CVE from the MITRE CVE database[2]. These references are web links to security bulletins, security update from vendors and bug tracking platforms describing the same CVE. The number of such references is positively correlated with the probability of being exploited [45].

4. We scrape all proof-of-concept exploits for CVEs, from various platforms. Most notably from `github.com`.

5. We scrape all exploits that can readily be abused using vulnerability exploitation frameworks. For this, we gather all the exploits present in the ExploitDB [3] and Metasploit project[4].

Then, through aggregation and integration of these online resources, we find the Linux kernel CVE fix commits, and extract the functions that are changed in these commits. This establishes a relationship between a Linux function and CVE. Naturally, there are functions that have not been any part of any publicly disclosed CVE. We set the score for these functions to 0. With that, we now have a metric for potential exploitability for every single Linux kernel function, which we can cross-link with the recorded functions during execution of the isolation platforms. Note that the Linux kernel allows for tracing at a higher resolution than functions (e.g. it supports per line tracing using the `lcov` tool, as is done in [14]), but CVEs (and as a result the EPSS model) provides us with scores at the granularity level of functions, not lines of code.

A figure outlining the pipeline of the extended HAP is given in Figure 6.2. On the left, the isolation platform its associated kernel function executions are recorded using `ftrace`. These functions are then fed to the EPSS model, in which the sigmoid function represents the logistic regression nature of the model. The scores the model outputs are summed, which gives us the final extended HAP score.

6.1.3. Extending EPSS resources

As outlined in the previous subsection, we have collected and integrated datasets from various online resources. Although this is already a wealth of information, there are certain incompatibilities between the different datasets. This is best explained using a figure:

As is seen in Figure 6.3, there is a mismatch between the CVEs listed for the Linux kernel from various resources. The kernel CVEs, in red, are strictly required for establishing the correspondence between a Linux git commit and a CVE. Thus, after integration, we end up with 1189 CVEs at the intersection of kernel CVEs and CVEdetails.com CVEs. Another mismatch becomes apparent here,

[2] `cve.mitre.org/cve`
[3] `exploit-db.com`
[4] `metasploit.com`

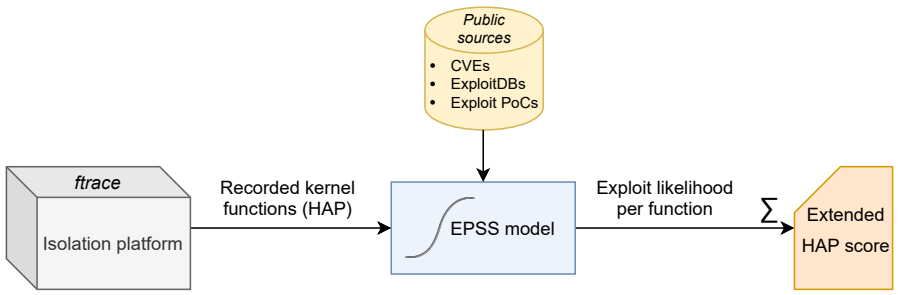

Figure 6.2: Extended HAP pipeline

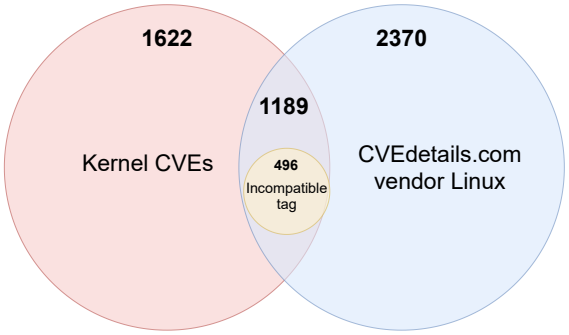

Figure 6.3: Integration of various datasets

this time between the parameter values used in the EPSS model and `cvedetails.com`. Specifically, there are 496 CVEs that do intersect, but the information as scraped lacks the information needed by the EPSS model. These CVEs do not contain information about the vulnerability type, and whether they are remote or local vulnerabilities. This is the set obtained by taking the CVEs in the red circle and subtracting the intersection of the two circles in Figure 6.3. In the coming paragraph we discuss how we attempted to impute these missing values, in order to increase the number of kernel functions that have a EPSS score associated to them. Note that this is different than the yellow circle, that indicates CVEs with tags but are not compatible with the EPSS model.

Our initial approach was to predict the vulnerability types that are missing from the CVE entries. We call this prediction variable the 'tag'. We train a text classification model based on the CVEs for which we all have training data (the blue circle in Figure 6.3). The training variable is the description of that CVE (examples of which are shown in the second column of Table 6.2), and the prediction variable is the tag. It should be noted that even within this dataset the distribution of tags that we are predicting is highly imbalanced: the distribution of the three vulnerability types present in the EPSS model is roughly 82.97% (denial of service), 10.5% (memory corruption) and 6.49% (code execution). As such, training a text classification model and predicting the tags resulted in poor results for some classes, with a recall score for the minority classes hovering around 0.65. We considered and implemented three different approaches in an attempt to tackle this problem:

1. Undersampling: we discard samples within the majority class (denial of service), and train a model given a uniform distribution of tags. However, this approach inhibits the training of a generalized model due to a lack of data. In other words, we do not have enough data to train a model using an undersampling approach.

2. Oversampling using SMOTE: first, we create feature vectors of the textual data. This transforms each sample from a text representation to a vector, in which each element represents one word in the total vocabulary of the text data. An occurrence of a word within a sample

is represented by a 1 in the corresponding position of the vector, absence is indicated by a 0. Each CVE description thus is transformed to a vector of length $vocab\_size$. We then oversample these vectors by using the SMOTE technique [18], in which we construct synthetic samples of the minority class. This technique constructs these synthetic samples by taking two existing samples of the minority class and their vector representations, and then drawing a sample in between these two points in the vector space. This sample is a new synthetic sample of the minority class. This approach however did not yield us better results. We believe the problem with this approach lies in the fact that the vectorized counts are counted using integers (i.e. samples from the vector space $\mathbb{N}^{vocab\_size}$), and the SMOTE technique creates vectors sampled from floating point space between 0 and 1 (i.e. a subset of $\mathbb{R}^{vocab\_size}$). Additional research is needed to confirm these suspicions, but in our case it is sufficient to conclude that it does not allow for a model that accurately predicts tags.

3. Oversampling using text embeddings: we create synthetic minority samples by using a pre-trained text embedding, such as the popular BERT text embedding from Google Research [85]. Feeding the text of samples within the minority class as input to such text embeddings allows for easy creation of synthetic minority samples. However, the text that describes CVEs is highly domain specific. Using a pre-trained general-purpose text embedding for this task can thus not be used to our advantage.

4. Using references: for each reference of each CVE (see third item in enumeration at the beginning of this section for an explanation of what these references are), we scrape all text from all the referenced URLs. This produces a large set of textual data pertaining to all the CVEs in the order of several gigabytes of textual data. We concatenate all the scraped text per CVE and use this data to predict the tags using a pre-trained model. This is also the method as used in the original EPSS paper [45], however, we were not able to properly reproduce their findings using this method. Most of the pages contained too much information irrelevant (i.e. noise) to the CVE to make accurate predictions.

Despite the limitations discussed above, training a model using these approaches lead to an increase of relevant statistical metrics. For example, training a Linear Support Vector Machine on CVE description data oversampled using SMOTE results in the metrics shown in Table 6.3. An increase in relevant metrics is obtained, yet there are significant limitations to this model. First of all, and perhaps most importantly, it does not generalize well. This means that on unseen data, its predictions are not accurate. This can already be inferred from the low support (the number of testing cases within that class) as shown in the rightmost column in 6.3, but is perhaps more illustrated by sampling some of its predictions, as shown in Table 6.2. The Table shows 3 samples as predicted by the model used in Table 6.3. The model, despite oversampling, exhibits a bias favoring the majority class, almost always predicting the denial of service tag. The complete table is omitted for brevity, but the problematic bias is evident in this small sample.

Having employed various techniques and methods to overcome model bias, we ultimately opted for a simpler keyword-based approach. In this approach, we carefully select specific keywords that are relevant to a tag. These tags are manually extracted from the descriptions of CVEs using ad-hoc analysis. This keyword approach results in roughly 150 additional CVEs getting tags assigned, out of the 433 potential CVEs. It is plausible that there are CVEs among the remaining 283 that should but don't get a tag assigned using the keyword-based approach. The CVEs that do get a tag assigned however have a relatively high accuracy (due to strict manual selection of keywords), unlike when employing a machine learning approach as described prior in this section.

Table 6.1: Linear SVC model results (train/test: 0.7/0.3)

CVE	**CVE Description**	**Predicted tag**
CVE-2020-25284	The rbd block device driver in drivers/block/rbd.c in the Linux kernel through 5.8.9 used incomplete permission checking for access to rbd devices, which could be leveraged by local attackers to map or unmap rbd block devices	Denial of Service
CVE-2020-10768	A flaw was found in the Linux Kernel before 5.8-rc1 in the prctl() function,where it can be used to enable indirect branch speculation after it has been disabled. This call incorrectly reports it as being 'force disabled' when it is not and opens the system to Spectre v2 attacks. The highest threat from this vulnerability is to confidentiality.	Denial of Service
CVE-2015-8941	drivers/media/platform/msm/camera_v2/isp/msm_isp_axi_util.c in the Qualcomm components in Android before 2016-08-05 on Nexus 6 and 7 (2013) devices does not properly validate array indexes, which allows attackers to gain privileges via a crafted application	Denial of Service

Table 6.2: CVEs and their predictions using a LinearSVC model trained using data oversampled using SMOTE.

	Precision	Recall	Support
Code Execution	1	0.82	11
Memory corruption	0.96	0.93	27
Denial of Service	0.96	1	73

Table 6.3: Linear SVC model results on count-vectorized CVE description text oversampled using SMOTE (train/test: 0.7/0.3)

6.1.4. Extended HAP results

The results for the extended Horizontal Attack Profile metric is shown in Figure 6.4. In this Figure, we observe that the results resemble that of the plain HAP metric, and the relative intra-group results (e.g. hypervisors) are congruent. Differences are observed when we look at groups relative to each other (inter-group). In comparison to the plain HAP metric the container platforms score lower than hypervisors. This is indicative of containers using functions that are less likely to get exploited. A graph that plots the number of vulnerable functions per traced function is shown in Figure 6.5, and illustrates a similar picture: platforms using hypervisors, on average, tend to use more vulnerable functions (as classified by occurrence in CVEs) than containers, although the differences are negligible at this point in time.

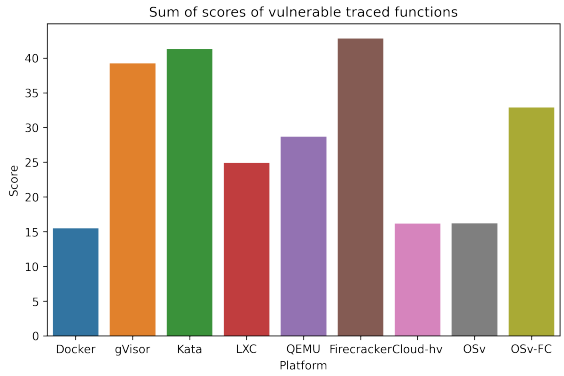

Figure 6.4: Total scores of extended HAP metric

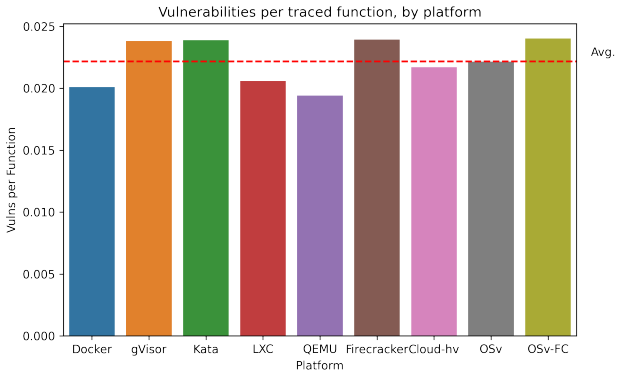

Figure 6.5: Vulnerabilities per traced function

6.1.5. Discussion of the extended HAP

There are advantages as well as some limitations to using the extended HAP metric. The first and foremost advantage is that the extended HAP establishes a relationship between vulnerability and exploitability, and therefore should result in a more realistic threat score. It weighs each function traced by its corresponding likeliness of exploitability. It does this based on historical data, which at the same time also proves to be one of the main limitations: historical data may introduce unwanted bias in the metric. Because containers are a new technology (relative to hypervisors), it might be the case that fewer relevant vulnerabilities have been discovered yet. This results in containers looking more secure when using the HAP metric as a basis, while it remains hard to tell whether this is the case in reality. If hypervisors were the newer technology of the two, it is possible that hypervisors would look safer, simply due to the reliance on the historic nature of the data the extended HAP metric is based upon. Careful selection of specific timeframes of the isolation platforms could help remedy this effect, but in practice, in our research there is little data even without filtering out any data (as extensively discussed in Subsection 6.1.3). However, one could argue that the field of vulnerability exploitation also adheres to this pragmatic nature: an isolation platform could be extremely vulnerable, but if there are no known exploits, it remains unlikely to get exploited. Furthermore, over time, as more data becomes available, stronger conclusions can be drawn from the extended HAP metric, as the recency bias naturally diminishes over time. If critical new vulnerabilities are disclosed, it can easily be added to the database of historical data, and would immediately trickle down to the extended HAP scores of the affected isolation platforms. As such, we believe our extended HAP to be valuable, and over time, can only improve.

We would also like to note a limitation that is inherent to the use of the plain HAP metric. The plain HAP metric fails to capture the defense-in-depth isolation platforms provide. For example, while Kata containers have a large HAP, they also introduce defense-in-depth, by using both names-

paces as well as a hardware-assisted isolation mechanisms. Moreover, the potential attack surface between tenants of an isolation platform is also not accounted for. The HAP measures the (horizontal) width of the attack profile, but is unable to capture these vertical, defense-in-depth, aspects of the isolation platforms. For the remainder of this chapter, starting at Section 6.2, we will look at this Vertical Attack Profile (VAP) in a qualitative manner.

6.1.6. Extended HAP conclusion

RQ2: *Is the degree of isolation offered proportional to the performance overhead imposed by an isolation platform?*

To answer **RQ2**, we first place each platform in a spectrum of extended HAP from narrow to wide, as is shown in the bottom part of Figure 6.6. Unikernels exhibit the most narrow HAP, followed by containers, hypervisors, and finally secure containers, which have the widest HAP. We then map each isolation platform category from performance overhead (grayed-out part of Figure 6.6. This Figure is also shown in 5.23) to its corresponding entry in the extended HAP spectrum. We make the following observations about this mapping:

- Secure containers, while imposing the highest performance overhead, also have the widest extended HAP.
- Unikernels show the most narrow extended HAP, while performance overhead was relatively high. This also implies that a wide HAP is not inherent to the use of a hypervisor.
- Containers and hypervisors were placed at the low overhead end of the performance spectrum, but display an extended HAP in-between of other platforms.

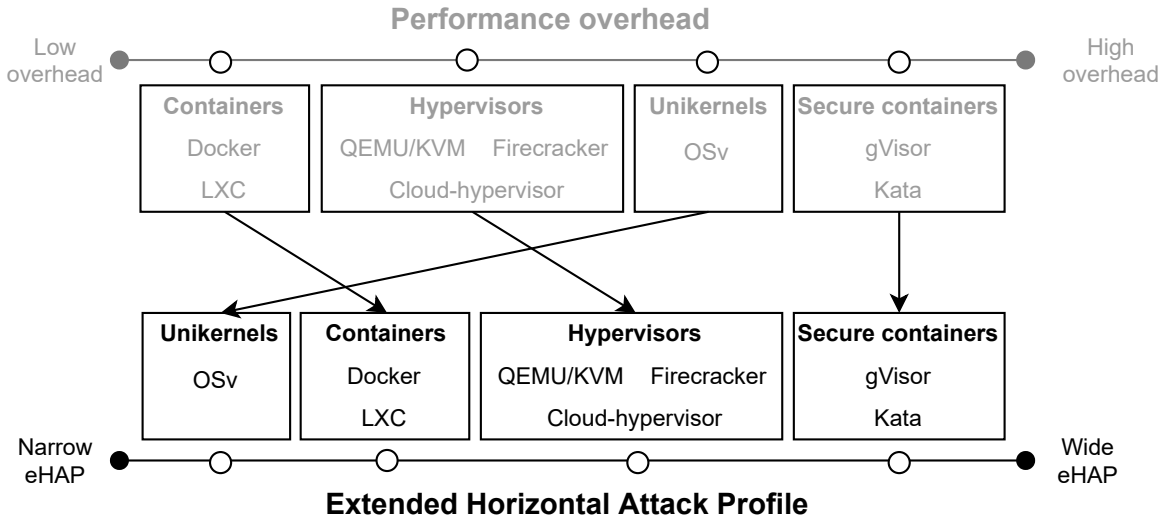

Figure 6.6: Mapping of performance overhead to extended HAP per isolation platform category. Performance overhead is displayed in gray as it is also shown in Figure 5.23

As such, we can answer **RQ2** as follows: considering the spectrum of the imposed overhead of isolation platforms, and the mapping of these platforms to the extended HAP spectrum, we do not observe any clear correlation with the Extended HAP results. Furthermore, no correlation between this and any other spectra in this thesis (the generality of virtualization in Figure 3.1 or isolation boundary location in Figure 3.2) is observed.

6.2. Vertical Attack Profile

In this section, we will describe what kind of isolation mechanisms there are (and provide CVEs for breaking these). We start each subsection with discussing the additional security measures that are typically enforced when using the platform. We continue by briefly explaining which isolation mechanism is used, and provide figures to illustrate the (location of) mechanism. Then, we discuss what different categories of vulnerabilities are typical for the platform being discussed, based on past research and CVEs. Finally, we discuss trends in research that have been proposed to increase security, and provide a conclusion.

6.2.1. Hypervisors

Before we begin categorizing the types of vulnerabilities that are typical for hypervisors, it is worth mentioning that hypervisors such as QEMU are rarely invoked directly in production, but would rather make use of a management tool like `libvirt` (and its command line front-end tool `virsh`). Such tools provide ways to readily create more secure VMs than the underlying hypervisor provides by default. In the case of `libvirt`, by default, it runs QEMU as an unprivileged user, restricts the process with SELinux, creates a mount namespace, and so on. Other hypervisor projects, like Firecracker, strongly suggest running the Firecracker binary inside an additional so-called jailer process: this jailer process adds another layer of isolation by creating a private PID and network namespace, chrooting, dropping privileges and enabling a seccomp filter (disallowing all but 24 syscalls), before it `exec()`s into the Firecracker binary.

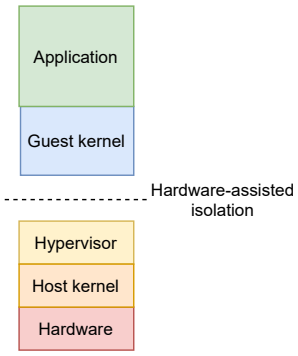

Figure 6.7: Hypervisor isolation mechanism

For explaining the first type of vulnerability category, we should recall that hypervisors (such as QEMU/KVM) typically ensure isolation through hardware-assisted virtualization. Non-virtualizable instructions can only be executed in ring -1, and as such the CPU enforces a way that only allows the hypervisor to run such instructions. Privileged instructions cause a trap and will be routed by the CPU to a handler in ring 0 code, at which the guest kernel is running. For paravirtualized devices running inside the guest, hypercalls are the interface to the hypervisor. Therefore, both privileged instruction handlers and hypercalls potentially provide the VM tenant with a way to escalate privileges over the resources of the host or cause a denial of service. An example of a vulnerability found within a privileged instruction handler is CVE-2020-25085, in which the instruction handler accounts for a 12-bit argument but the user can provide up to 16-bits, resulting in an out-of-bounds write that can cause denial of service. An example of an issue with a hypercall is CVE-2016-5412. In this CVE, if the paravirtualized guest makes a `H_CEDE` hypercall while the CPU is in a transactional state, the host CPU may get stuck in an infinite loop, causing a denial of service.

Another potential category of vulnerabilities in hypervisors are found in the implementation of the virtualized hardware infrastructure of VMs itself, rather than in the interface between the host

and guest. An extensive characterization and subcategorization of these sources is provided in [66], outlining 6 different subcategories. Examples include implementation flaws in virtual CPUs (CVE-2010-4525), in the software MMU (CVE-2010-029), interrupt and timer mechanisms (CVE-2010-030) and even in a floppy disk controller (CVE-2015-3456). It is important to highlight that this category is inherent as well as unique to hypervisor isolation platforms that employ virtualized hardware. In a breakdown of known vulnerabilities for the hypervisor platforms Xen and KVM, CVEs of this category respectively account for 54.3% and 84.2% of all known CVEs. This finding strengthens the motivation and need for hypervisors (such as Firecracker) to keep the virtualized device model minimal.

An interesting trend in research for more secure (cloud) virtualization is the idea of getting rid of the hypervisor altogether. For example, [50] and [80] propose the NoHype architecture: remove the hypervisor layer and instead use existing techniques and use the naturally arising partitioning in hardware for isolation. Concretely, the authors propose (1) using only one VM per core (and instance of naturally arising partitioning), (2) have the existing hardware MMU handle memory partitioning, and (3) employing a modified IOMMU that presents itself as N separate devices, each of which is responsible for a memory range with the width of the total physical range divided by N. The authors thus advocate the further use of hardware to enforce isolation, to the extent where a hypervisor is rendered redundant. In [92], the authors argue that virtualization mechanisms are not a requirement for a strong isolation boundary, and that other isolation platforms like containers can provide the same degree of isolation. This can be accomplished through a process that the authors refer to as 'microkernelfication', in which functionality traditionally seen as kernel functionality is moved to user-space daemons, and the existing kernel acts as a mere message-passer (this architecture is resemblant of a microkernel, hence the name microkernelfication).

Despite the large attack surface that hypervisors appear to expose (both in HAP as well as CVEs), as well as publications proponing alternative architectures to get rid of hypervisors altogether, hypervisors continue to be a popular isolation platform. There are many potential reasons why this is the case, and it remains hard to say which are the true reasons, but the maturity of the underlying technology (e.g. KVM and hardware-assisted virtualization) as well as compatibility with existing software likely play a large role. Orchestration platforms such as `libvirt` can also help remedy some vulnerabilities, for example by applying namespaces and seccomp policies.

6.2.2. Containers

In this subsection, we first discuss the Docker ecosystem as a whole, excluding the actual isolation mechanism used in containers. We discuss the additional security measures taken by Docker and also discuss a new attack surface as introduced by this ecosystem. We then categorize the various types of vulnerabilities that are typical for the container isolation mechanisms that Docker and LXC use. The vertical attack profile of the secure containers gVisor and Kata containers are discussed in later subsections (Subsection 6.2.4 and 6.2.5, respectively).

Docker, as well as the secure containers gVisor and Kata containers, are used in conjunction with the Docker CLI (e.g. through `docker run`). Running containers with the Docker CLI implies usage of the Docker daemon `dockerd`. This daemon runs as root. By default, the daemon creates containers using Docker's `runc`, but Kata and gVisor containers can be started by setting the `--runtime` flag accordingly. Because the isolation platforms are typically invoked through the Docker daemon, it is relevant to discuss the additional security measures it provides (even more so considering that [55] found that these additional security measures play a more important role in preventing privilege escalation than the base isolation mechanisms namespaces and cgroups do).

By default, Docker starts containers with a restricted set of Linux capabilities. Linux capabilities break up the privileges traditionally seen as superuser privileges into distinct units called capabilities, that can be enabled on a per-thread basis [13]. Examples of such capabilities are `CAP_SYS_BOOT` which determines whether a thread can reboot the system, and `CAP_DAC_OVERRIDE` determines whether the thread can bypass file read, write and execute permissions checks. By default Docker enables 14 out of 41 Linux capabilities. Another security measure that is enabled by default is a seccomp filter. The seccomp filter blocks any non-whitelisted system calls, and by default blocks 44 out of more than 300 calls. Note that even though it filters 44 system calls, there is significant overlap between the seccomp filter and linux capabilities, meaning a significant set of system calls is already gated by the allowed linux capabilities. Other measures that Docker provides to harden the containers are not enabled by default. Most notably this includes AppArmor, a Linux kernel module that administrates which files can be accessed by an application. AppArmor profiles exist for both the containers itself as well as `dockerd`, but are not applied by default.

Docker Hub is an online repository that lets developers easily download and upload Docker images, and is tightly integrated with the Docker CLI (through e.g. `docker pull` and `docker push`). Although convenient, this integration also exposes an additional attack surface. Downloaded images can be uploaded by anyone, and typically do not undergo the same strict security scrutiny as Linux distributions (that are used in VMs) do. New software solutions for analyzing Docker images have been developed, such as Snyk [90], which is by also integrated into Docker Hub itself [40]), but can never guarantee absence of vulnerabilities. Prior research highlights the potential severity of this new attack surface: the authors of [76] find that both official images (which are curated by the company behind Docker, Docker Inc.) and community images average over 70 vulnerabilities per image, even when only the most recent version of an image were analyzed. When all versions where analyzed, the average image contained over 180 vulnerabilities. Such vulnerabilities are also automatically propagated from parent to child images with the Dockerfile `FROM` statement. Moreover, it is also common practice to download dependencies from other third-party resources. These dependencies are typically downloaded upon every commit (through a CI/CD pipeline), meaning that if an adversary gains control over any of these third-party resources it would be trivial to get code running on production machines. This type of supply chain attack was successfully executed in [21], in which malicious code ended up on production servers within just over five minutes.

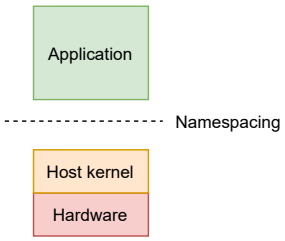

Figure 6.8: Container isolation mechanism

Docker uses two features of the running host Linux kernel to establish an isolation boundary: cgroups and namespaces. Breaking out of the namespace can potentially do a lot of harm, as Docker processes run as root. An escape from the container thus results in the adversary controlling a process running as root directly on the host. CVEs underlining this exact problem are numerous (e.g. CVE-2014-6407, CVE-2014-9357, CVE-2015-3627, CVE-2019-5736). Escape from containers is possible by modification of shared memory (as discussed in [47] and shown in CVE-2016-5195), but more often by exploiting weaknesses in the filesystem, such as through symbolic links (e.g. CVE-

2018-15664, CVE-2018-15664, CVE-2015-3627, CVE-2014-6407). LXC roughly shares its VAP (isolation mechanism and typical vulnerabilities) with Docker, as it also builds upon namespaces and cgroups. For example, CVE-2015-1331 also describes an attack through symbolic links for LXC.

As a response to the recurrence of filesystem based vulnerabilities, the authors of [52] present a safer shared-filesystem isolation mechanism for containers. The main idea is that only read operations are directly permitted to the host, and all other I/O operation are handled by a deprivileged filesystem. This deprivileged filesystem utilizes a low-level block interface. This reduces the attack surface, while complex host abstractions such as symbolic links remain available for the guest through ahead-of-time preparation of scanning the host filesystem metadata at container creation time. Container startup overhead the ahead-of-time preparation technique imposes remains limited (120 milliseconds on a cold start) due to the copy-on-write nature of the deprivileged filesystem. A second proposed solution to improve I/O subsystem isolation is outlined by Kappes et al. in [48]. In this work, the authors introduce the libservices framework, which acts as an abstraction layer between the container workload and host kernel. This abstraction layer contains both the application (container workload) and system services in user-space (e.g. a filesystem in user-space through FUSE). In other words, it moves parts of the host kernel functionality up to the container, with an increase in isolation as a result.

Breaking cgroups, although harmful, could at most lead to oversubscribing physical resources of the host (which can affect other tenants running on the same host). In [33], the authors show that exploiting the cgroup mechanisms through deliberately spawning new child processes can lead to performance degradation for other tenants. In extreme cases, the adversary tenant was able to oversubscribe resources up to 200x, while neighboring tenants would experience system performance degradation of up to 95%.

Due to the massive popularity of containerization in industry, many additional secure container frameworks and methods have been proposed. For example, the authors of [8] design and implement a container framework, SCONE, that uses Intel's Software Guard Extensions (SGX) for a secure container environment. Intel SGX provides applications a trusted execution environment, in which confidentiality and integrity are guaranteed even in the presence of a malicious host (e.g. malicious host kernel) [42]. Another example that can potentially increase the security of containers are orchestration frameworks, such as the popular Kubernetes framework. An orchestration framework may strengthen the isolation boundary by adding additional layers of abstraction, for example:

- Every container runs inside an isolated Kubernetes pod. This adds an additional layer of indirection. Pods also allow for the declaration of network policies, which limits the allowed ways of communication between pods.
- Kubernetes clusters may be deployed on a stripped-down and hardened OS specifically built for running Kubernetes (e.g. Rancher OS[5], VMWare Photon[6] and CoreOS[7])
- Kubernetes allows for the encryption of data at rest through the use of secrets.

However, it is beyond the scope of this thesis to enumerate, discuss, and test all of these new platforms (see Chapter 8 for related work in this space). In conclusion, the isolation boundary of plain containers (without orchestration framework) remains relatively thin, as it is only guarded by the implementation of namespaces and cgroups in the host kernel. The fact that improper validation

[5] `https://rancher.com/products/rancher`
[6] `http://vmware.github.io/photon`
[7] `https://coreos.com`

of symbolic links have often provided a means to escape a Docker or LXC container testifies this statement. Docker image repositories (Docker Hub) provide a convenient but vulnerable additional attack surface.

6.2.3. Unikernel

A unikernel shares much of the VAP characteristics with the hypervisors, since the unikernel image is run by a hypervisor. Indeed, the vertical attack profile of hypervisors and unikernels are the same, since they use the same isolation mechanisms. The difference is the level of the interface that the application communicates through: a small unikernel, instead of a full-fledged Linux kernel. The authors of [92] generalize this observation into the following statement: "the level of interface between the guest and host is not fundamentally tied to the actual isolation mechanism". As such, the difference in security between unikernels and hypervisors running ordinary operating systems lies solely in the width of the horizontal attack profile (as a result of a difference in level of interface), not in the vertical attack profile.

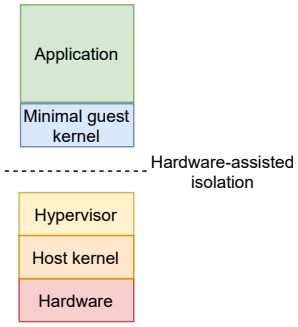

Figure 6.9: Unikernel isolation mechanism

Work on further increasing the security of unikernels has been carried out nonetheless. In [75], the authors propose and implement a unikernel that make use of the Intel SGX security enclave, similar to how SCONE uses Intel SGX for securing containers as discussed in Subsection 6.2.2.

6.2.4. gVisor

gVisor redirects requests that require higher privilege from the host kernel to its user-space Sentry process. The Sentry process offloads its I/O request to the Gofer process. Both of these processes are restricted by a namespace, and the interface between the Sentry/Gofer process and the host kernel is guarded by a seccomp filter. The Sentry thus offers an abstraction layer in between the host kernel and the containers, theoretically making it harder to reach and attack the host kernel directly. This opens up the possibility for the Sentry process itself being attacked directly by the tenant, and CVEs such as CVE-2018-16359 and CVE-2018-19333 showcase this. As gVisor implemented its complex user-space kernel from scratch, and it being a relatively new project, it is not unthinkable that more significant vulnerabilities will surface in the near future. However, gVisor remains arguably more secure than the Docker default container runtime, as the Sentry runs in the unprivileged user-space (instead of kernel-space) and is restricted by a seccomp filter. The authors of [93] also draw this conclusion.

6.2.5. Kata containers

The vertical attack profile of Kata containers should and does look similar to the vertical attack profile of Docker stacked on top of the profile of the hypervisors. This means that Kata containers use both namespaces and hardware assisted virtualization as its isolation mechanisms, leading to an

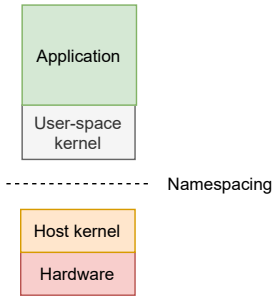

Figure 6.10: gVisor isolation mechanism

increase in complexity and thus an increase in size of the trusted computing base, also resulting in a visible widening of the HAP. All published CVEs at the time of writing affect Kata containers' first line of defense, namespaces: incorrectly (un)mounting (CVE-2020-2026) and improper file permissions (CVE-2020-2023, CVE-2020-2025, CVE-2020-2024,CVE-2020-28914). This type of vulnerability is reminiscent of the Docker vulnerabilities discussed in Subsection 6.2.2 and seems to be inherent to the use of namespaces as an isolation mechanism.

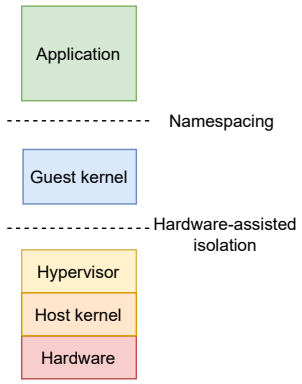

Figure 6.11: Kata isolation mechanism

6.3. Security conclusions and limitations

In this section, we presented research into various aspects pertaining to the security and degree of isolation offered by the isolation platforms. With these results, we can answer **RQ3**: *Does the extended HAP metric accurately quantify the degree of isolation?*

Answering **RQ3** requires us to first answer the question of whether the extended HAP significantly improves upon the regular HAP metric. As discussed in Subsection 6.1.4 and 6.1.5, we believe that the extended HAP provides significant improvements over the regular HAP, although it also exhibits some limitations:

- The extended HAP addresses the lack of sensitivity to functions that have proven to be relatively prone to exploits. This results in a more accurate quantification of the degree of isolation.

- The differences in values for the isolation platforms at this point in time are small, but will increase as more historical data and vulnerabilities are disclosed. Severe vulnerabilities and exploits will directly result in the extended HAP score of affected platforms.

- The EPSS model introduces a recency bias, where the newer platform has less public CVEs and exploits available, skewing the distribution of the EPSS model in their favor. We argue, however, that this bias is also present in the field of vulnerability exploitation itself, and, as such, the established relationship closely approximates the true likelihood of exploitation.
- The EPSS model is limited by the amount of information it is fed during the training of the model. This implies a reliance on the availability of public vulnerability and exploit datasets, which are, in our experience, not always exhaustive.

As such, we conclude that the extended HAP improves upon the regular HAP. As a follow-up question, we discuss whether the regular HAP in itself is a good metric for measuring the degree of isolation that an isolation platform offers. This question remains hard to answer, as we cannot compare it to any other quantitative measure. Instead, we can only compare it to the qualitative security research in this chapter, which discussed the various types of vulnerabilities as well as recent advances in defenses. We observe the following limitations with the plain HAP metric:

- Defense in depth is not captured by the HAP metric. This is particularly apparent for the secure containers category, platforms of which typically have a very wide HAP, while their *raison d'être* is to be more secure than regular containers. The HAP metric not being able to capture defense in depth as such lacks a critical component in the overall assessment of the degree of isolation of an isolation platform.
- Inter-guest communication is not captured with the HAP.
- The HAP is not sensitive to parts of the codebase that exhibit vulnerabilities more frequently than average, such as filesystem sharing in containers. The extended HAP metric addresses this point specifically.

In conclusion, we answer **RQ3** as follows: The extended HAP metric quantifies the degree of isolation more accurately than the plain HAP by not assuming a uniform distribution of vulnerabilities. The plain HAP is, as far as we are aware, the only quantitative measure of security. Although we do observe limitations to the use of this metric (e.g. not taking into account the defense in depth), we believe it to be a valuable metric for approximating the degree of isolation between the host and guest.

7

Conclusion and future work

Virtualization is the underlying technology that powers cloud infrastructure as we know it today. Virtualization enabled logical partitioning of computing resources, rather than being dictated by their physical reality. The use of cloud computing resources is exceedingly popular, with public clouds diversifying their offerings to serve the newly arising needs of customers. This increase in both mainstream adoption and variety of offerings underline the importance of several key properties of virtualization platforms. In particular, we consider the performance overhead and the degree of isolation offered of utmost importance. Prior research typically delved deep into either of these two properties, and are generally limited to a small set of examined platforms.

This thesis attempts to bridge this gap. It addresses both the performance and security aspects of various isolation platforms, as well as any potential correlations between the two. We provide a survey of the architecture and defense in depth measures as employed by the wide-ranging set of isolation platforms. Furthermore, we carry out an extensive collection of experiments, quantifying both the security and performance offered by the isolation platforms. These experiments include typical micro-benchmarks and real-world benchmarks, as well as a novel way to quantitatively measure the degree of isolation.

7.1. Conclusion

In this section, we reiterate the research questions and provide a brief summary of the conclusions reached for each of them.

RQ1: *Where do the new types of virtualization techniques position themselves on the spectrum of performance overhead incurred?*
A wide array of experiments was conducted to stress and subsequently characterize the performance of the various types of isolation platforms. We found that container platforms have the best, near-native, performance. Hypervisors exhibit significant differences amongst one another, but on I/O and CPU bound tasks typically perform on-par with native. Networking and memory always exhibit overhead. Secure containers particularly suffer from overhead in the I/O subsystem, but promising alternatives are being developed. Finally, unikernels perform well, but their performance is hard to characterize due to the various incompatibilities with benchmarking software. Start-up time is generally the lowest for containers, whereas for the hypervisors it is highly dependent on the used machine model.

RQ2: *Is the degree of isolation offered proportional to the performance overhead imposed by an isolation platform?*

To answer this question, we first established the spectrum of HAP on which we place the isolation platforms, from narrow to wide. We assume this to be indicative of the degree of isolation (also see **RQ3**). Our experiments showed that unikernels exhibit the narrowest HAP, followed by containers, hypervisors, and finally secure containers, which have the widest HAP. We then map the platforms from the performance overhead spectrum to the HAP spectrum. With this mapping, we do not observe a clear correlation between the degree of isolation of an isolation performance and its corresponding performance overhead. No other correlation between the degree of isolation and the spectrums of the generality of virtualization and isolation boundary location was observed.

RQ3: *Does the extended HAP metric accurately quantify the degree of isolation?*
The plain HAP metric exhibits limitations that we address with the extended HAP. Particularly, the lack of sensitivity to parts of the codebase that are more prone to vulnerabilities (based on historical data) was addressed. To that end, we constructed the extended HAP metric to quantify the likelihood of exploitability for an isolation platform, given the invoked host kernel functions. We found that the extended HAP scores, although numerical differences in comparison to the plain HAP at this time remain small, prove to be a useful metric for measuring the degree of isolation. Moreover, disclosure of new vulnerabilities and exploits can trivially be incorporated into the extended HAP metric. As such, the divergence of plain and extended HAP scores will increase over time, in which the extended HAP provides increasingly accurate measurements. We also observe limitations to the use of the extended HAP metric. First, the plain HAP, which the extended HAP builds upon, fails to capture defense in depth. Second, it remains hard to verify to what extent the degree of isolation is captured by the extended HAP, as there is no other quantitative measure to benchmark against.

7.2. Future work

This thesis has revealed numerous interesting insights that could serve as a basis for new research paths. Additional research could build on top of the work presented here, either to extend the scope of the study and provide a deeper analysis of specific causes of overhead, or to address limitations observed in the isolation platforms themselves. Although incomplete, the following list enumerates topics and ideas we believe to be worth pursuing in the future:

1. The platforms discussed and analyzed in this thesis, although wide-ranging, are not exhaustive. For example, the inclusion of a Type-1 hypervisor (e.g. Xen [9]) could provide additional insights into the characteristics of hypervisor-based platforms in general. Moreover, an extremely lean implementation of containers (such as `systemd-nspawn` [79]) could also provide new insights into the performance overhead that is minimal for a container implementation. Analysis of platforms that take a different approach altogether to isolation could prove an interesting path to pursue. For example, the SCONE platform [8] uses a hardware-assisted secure enclave to provide isolation.

2. Some results warrant extra research to draw stronger conclusions. Most notably the MySQL benchmarks provided unexpected results given the results in prior work [28]. The role of thread schedulers seems to be particularly weighty in determining performance. An interesting follow-up project could carry out a measurement study of the importance of thread schedulers on various subsystems and real-world benchmarks within isolation platforms.

3. Although the HAP and extended HAP are revolutionary in that they bring a quantitative metric to capture the degree of isolation, finding a way to incorporate the defense in depth measure taken by isolation platforms would likely bring substantial improvements to the accuracy of the metric. For example, secure containers typically employ additional layers of indirection to

increase security. Not only would incorporating this allows for a more accurate representation of the degree of isolation, it would also be able to serve as a benchmark against the the HAP and extended HAP metrics.

4. We have observed significant differences between the network latency and throughput as offered by the isolation platforms. Considering the importance of the networking subsystem, it could prove to be fruitful to take one step back and look at the performance characteristics and limitations of the general methods to enable network virtualization (for example, TAP devices and virtual ethernet cables). Moreover, additional research into drivers that take advantage of the specific context of isolation platforms, as such drivers, like virtio-fs, boast significant speedups.

8

Related work

Due to the popularity of cloud computing, and thus isolation platforms, the field has garnered significant attention from academia. There are various ways in which this attention manifests itself: from proposed improvements on existing systems (e.g. [52] proposes an improvement for container storage), to completely revising the methods of virtualization (e.g. [53] moves the virtualization layer up to the application level, and [5] proposes a new interfacing mechanisms between hypervisor host and guest). This increased attention combined with the relative ease of creating such platforms through powerful general-purpose libraries and modules like `rust-vmm` and `KVM`, lead to the development of a diverse landscape of isolation platforms and techniques. We will first discuss literature surveys that takes one step back, and compare such platforms. Then, we look at related work that examine only one platform. Finally, we present and discuss related work pertaining to the security of these platforms.

8.1. Performance comparisons

One important survey that provides a comprehensive performance comparison of virtual machines and containers on Linux is [28]. In this quantitative evaluation, the Docker container platform and QEMU/KVM are compared along various subsystem axes. In general, the results indicate that Docker attained or outperformed QEMU/KVM in every test. The CPU and memory subsystems performed nearly equal across the two examined platforms. In contrast, the I/O subsystem of Docker performed on-par with a non-isolated system, but QEMU/KVM significantly underperformed in this regard. The throughput of sequential I/O remained as high as native, albeit with a significantly higher standard deviation, while small (4kB) write/reads performance is half of that of Docker and the native platform. The survey concludes with the observation that a comparison between these two platforms can only get closer, as Docker imposed close to no overhead from its outset, while QEMU/KVM over time has minimized its overhead. The authors also observe that the convenience, faster deployment, elasticity and performance of Docker proves to be a compelling alternative to QEMU/KVM and hypervisors in general.

In similar vein, [34] presents a quantitative comparison of the OSv unikernel vs. Docker in a microservices context (entailing benchmarks mostly pertaining to REST services). This study concludes that unikernels outperform Docker, which should be attributed primarily by the kernel-intensive networking nature of REST services. By the transitive property, considering the previous paragraph, unikernels should thus outperform native platforms in this context, as also shown in [67]. Both works note a few caveats however, namely that the OSv unikernel has a higher memory

footprint (due to the addition of the minimal yet additional guest kernel) and are generally considerably less user-friendly in their current state. Mavridis et al. in [61] performs a quantitative comparison but in the context of software-defined systems and cloud computing, and reaches the same conclusion.

8.2. Isolation platforms

There are numerous published works that study one isolation platform in detail, rather than comparing them to other platforms. This results in work that can focus on (potential) shortcomings by carrying out specific benchmarks. On example is [93]. In this work, the gVisor platform is studied, and properties that might be impaired due to its specific architecture, are benchmarked. Besides typical performance benchmarks (e.g. memory allocation and network throughput), system call overhead and file opening performance are also benchmarked. The results show that system call overhead is severe, particularly when it cannot be processed solely by the Sentry process (i.e. I/O system calls). Furthermore, weak I/O performance is exacerbated due to excessive mode and process switching, and the 9P filesystem. The authors propose and benchmark a setup where a `tmpfs` filesystem is used instead, and conclude that it is a good, performant, alternative for stateless workloads.

In [3], the authors of Firecracker discuss and benchmark their own isolation platform. This work begins with discussing that with the rise of the serverless computing paradigm also implies a growing need for an isolation platform that promises both low overhead and a strong isolation boundary. Three options are discussed: containers, language-specific isolation (e.g. the Java Virtual Machine) and virtualization. The authors argue that, despite its shortcomings, virtualization benefits outweigh its downsides due to the presence of a guest kernel. This guest kernel, according to the authors, eliminate the need to "trade off between kernel features and security", as the threat model is independent of the feature-completeness of the guest kernel. Besides advocating the use of virtualization as key isolation mechanism, it also suggest running the hypervisor process to a constrained (jailed) environment. Finally, the paper concludes with a short performance evaluation that considers start-up time, memory overhead and I/O performance. The findings of the start-up times are replicated in this work (in Section 5.5), memory overhead is found to be the lowest of all hypervisors (with Cloud-hypervisor coming in at a close second place), and I/O, broadly speaking, underperforms due to the lack of a flush-to-disk block I/O implementation. Another (unpublished) work, [46], examines the internals of the Firecracker architecture in-depth, but does not include a performance evaluation.

The original OSv [51] paper presents the implementation design and a brief evaluation of the platform. As the architecture and implementation design have already been discussed in Chapter 4, we focus on the performance evaluation aspect of the work here. Concretely, the authors look at Memcached, SPECjvm2008 and network throughput performance. The authors note a 22%, 0.5% and 25% speedup, respectively. The authors also present a thread context switch benchmark, to showcase the efficiency of the custom thread scheduler. In this benchmark, two threads colocated on the same processor alternately wake each other up, and the average time per iteration of this process is measured. OSv, on average, takes 328 ns to wake up both threads, whereas native Linux takes around 905 ns.

Another approach to lightweight VMs is proposed and implemented in [58]. In this work, the authors introduce the LightVM isolation platform, which implements a redesign of the Xen hypervisor to provide lightweight virtualization. The primary value proposition of this redesign is that, unlike

plain Xen, does not require a message-passing interface between the front-end and back-end of paravirtualized drivers, but instead employs shared memory between the host and guest. This in turn leads to a decrease in performance overhead (i.e. lightweight), while still retaining the high degree of isolation as offered by hardware-assisted virtualization. This work does not focus on traditional performance benchmarks, but instead benchmarks metrics related to the creation of guests (such as guest instantiation time, memory and CPU usage on the host). In these benchmarks, the authors show that the LightVM platform boasts a near-constant creation and boot time regardless of the number of running VMs, but, as mentioned above, does not discuss traditional benchmark performance.

Naturally, considering the maturity and vastness of the field of software-based isolation and virtualization, there are significantly more texts that present or discuss a single isolation platform. However, some papers present isolation platforms that, over the course of time, have evolved into platforms serving substantially different needs than at the time of publication. For example, the original publication on the QEMU hypervisor [11] primarily focuses on emulation, rather than virtualization.

8.3. Security

For any isolation platform, security is of utmost importance. Naturally, a lot of work in the field of isolation platform security has been carried out, from container security (e.g. [16] [47] [60] [55]), to hypervisor security (e.g. [91] [77] [20]). The majority of publications focus on discussing one platform, and as such, have been discussed in Chapter 6 already. Instead, in this section, we discuss existing surveys on general categories of isolation platforms (e.g. containers).

The authors of [10] construct a taxonomy of container defenses, and places existing container defense frameworks into these categories. The taxonomy consists of just 3 types of container defenses:

- Configuration based defense: Configuration set and enforced by the host kernel disallows certain behavior. For example, a whitelist of allowed network interfaces to be used, or directories to be mounted. An example of a container defense framework of this type is AppArmor.

- Code-based defense: From the host, code is pushed to the container to enforce additional security measures. Pushing the code is typically made possible through the use of the eBPF Linux kernel functionality [1]. Seccomp-BPF is a defense framework in this category, which enforces a policy on allowed system calls within the container itself (based on the pushed eBPF code).

- Rule-based defense: From the container, string rules are pushed to the host. The host is responsible for enforcing these rules through handlers in the kernel. An example of this defense type is OpenBSD's `pledge()` system call [72], reducing the abilities of a process (e.g. allowing changes to routing tables, or interacting with file descriptors), and can never regain them.

The paper concludes that the use of any type of container defense mentioned above can potentially significantly improve security, but due to the diversity regarding the objectives, architectures and implementation details, it remains challenging to optimally make use of such defenses. Potential remedies outlined are tailoring general defense techniques (e.g. AppArmor) to containers specifically (akin to [56]), or hybrid abstractions that take advantage of both Lightweight VMs and containers (e.g. gVisor).

In [22], a comprehensive overview of various types of isolation platforms is presented. This research includes Type-1 and Type-2 hypervisors, containers, and unikernels. For each of these platforms, the authors qualitatively assess the likeliness of exposure to various vulnerabilities, as well as the relationship between vulnerabilities and attacks. Based on this data, the authors propose three improvements to increase the degree of isolation offered by the isolation platforms. First, the authors plead for integration of security mechanisms at design time, as their research has shown insufficiencies of architectural countermeasures in hypervisors. Second, attack surface of hypervisors should continue to get minimized (for example by simplification of device model, as the Firecracker hypervisor does). Finally, security mechanisms (e.g. seccomp profiles) should be actively managed to combat the ever-evolving arsenal of exploits. The authors also note that unikernels in particular offer promising potential for reducing attack surface through 1) simplified architecture and 2) integration of security mechanisms at image build time. In similar research, albeit on a smaller scale, [73] suggest that the biggest threat to hypervisor-based platforms is an escape from the VM, and should also be mitigated through simplification of architecture of hypervisors.

In [25], a broad qualitative comparison between full-system virtualization, containers and unikernels is given. A trade-off is implicitly presented in this work, in which one the one hand running a full OS within a VM for one application is prohibitively inefficient, but containers do not offer the strong isolation boundary that virtualization platforms do. The author considers the use of unikernels an adequate solution, although imperfect due to the lack of privilege levels. An optimal solution, instead merely being adequate, should combine the best features of the 3 discussed types of isolation platforms. New solutions should be assessed on these criteria through the use of several metrics: monolithic vs. microkernel architectural differences, the use of privilege levels, and attack surface measurements. A study that employs this last metric to assess three different types of isolation platforms (a hypervisor, container and secure container) can be found in [14], which shows that platforms that make extensive use of code outside the kernel for OS functionality still execute a higher relative amount of code in the host kernel.

Although the works above present an accurate and up-to-date overview of the security of isolation platforms, none of them provide quantitative data. As discussed in Section 6.1, we attempt to measure the degree of isolation through execution tracing. This method was initially outlined in [12], and eventually lead to a similar measurement approach in [14]. The latter work captures the executed host kernel code for the Docker, gVisor and Firecracker platforms. Broadly speaking, based on the data obtained in their experiments, Firecracker reduced the frequency of invocation of kernel code, but increased the amount of executed kernel code. gVisor invoked a large amount of duplicated host kernel code indirectly through its user-space Sentry process. Finally, Docker containers executed the least amount of host kernel code of the three. These findings wholly align with our measurements and conclusions in Section 6.1.

A

Additional performance results

In this chapter we provide additional results that were obtained through benchmarking, that were omitted for either clarity. Section A.1 contains the results of the other available STREAM benchmarks Add, Scale, and Triad. Section A.2 provides figures that display the 50th and 90th percentile using the same data as used in the main text for the 90th percentile. Finally, Section A.3 and in particular Figure A.6 shows the number of insertion operations executed for each platform, which exhibits behavior nearly identical to that of the actual benchmark in the main text.

A.1. STREAM benchmark

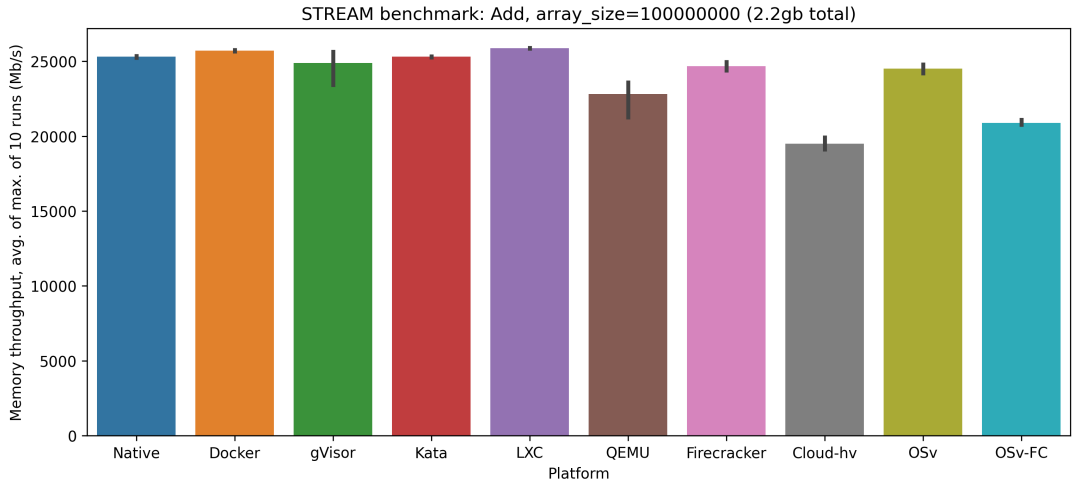

Figure A.1: Memory add throughput STREAM benchmark

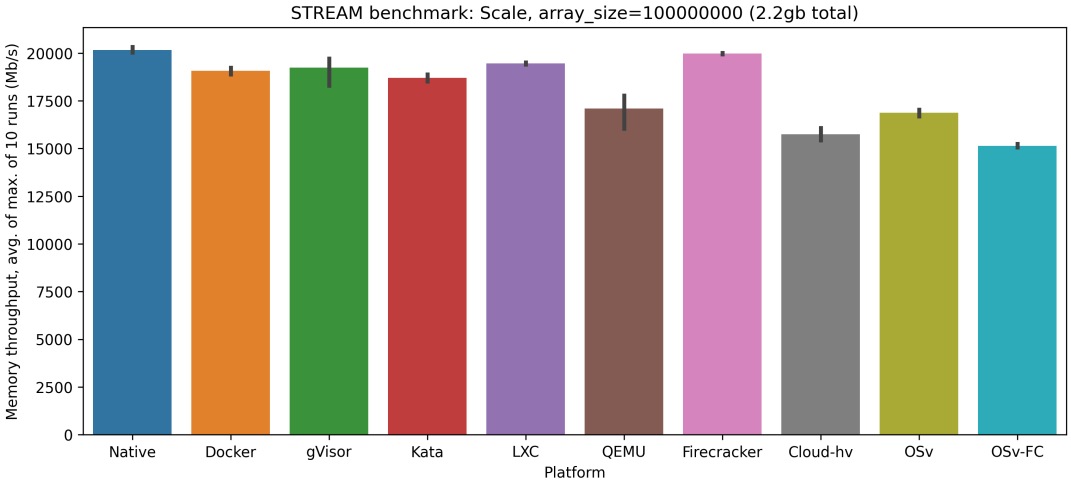

Figure A.2: Memory scale throughput STREAM benchmark

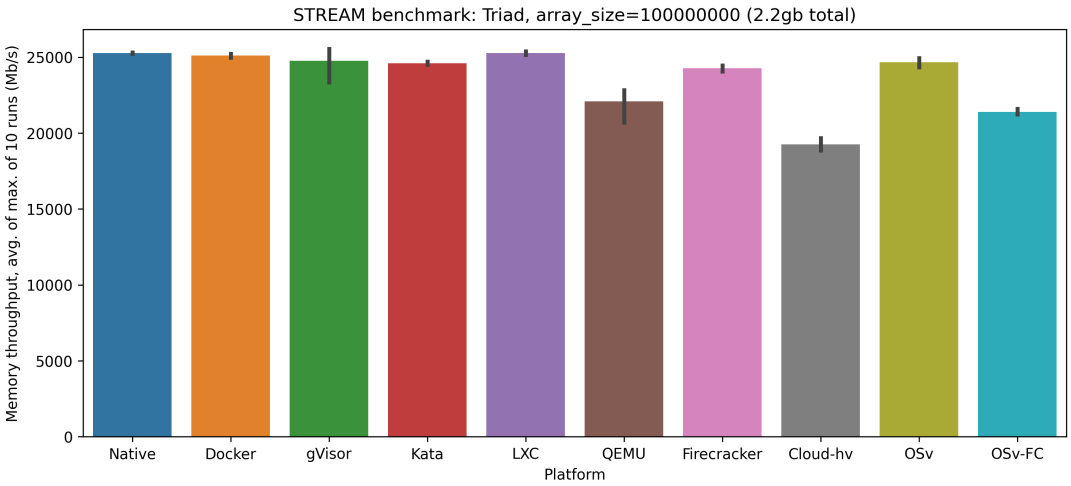

Figure A.3: Memory triad throughput STREAM benchmark

A.2. Netperf benchmark

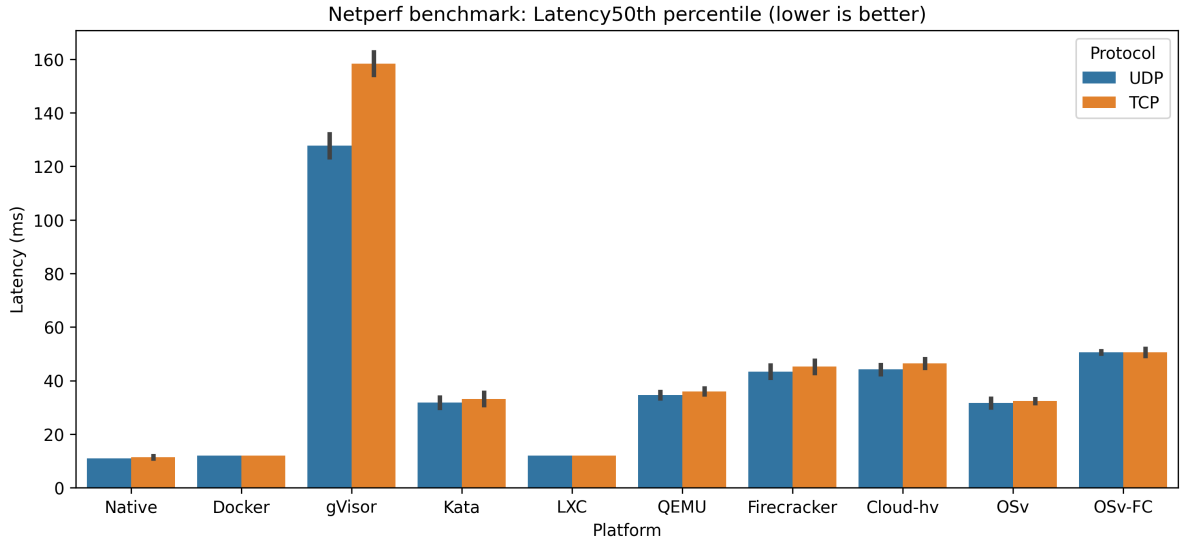

Figure A.4: Netperf network latency benchmark (50th percentile)

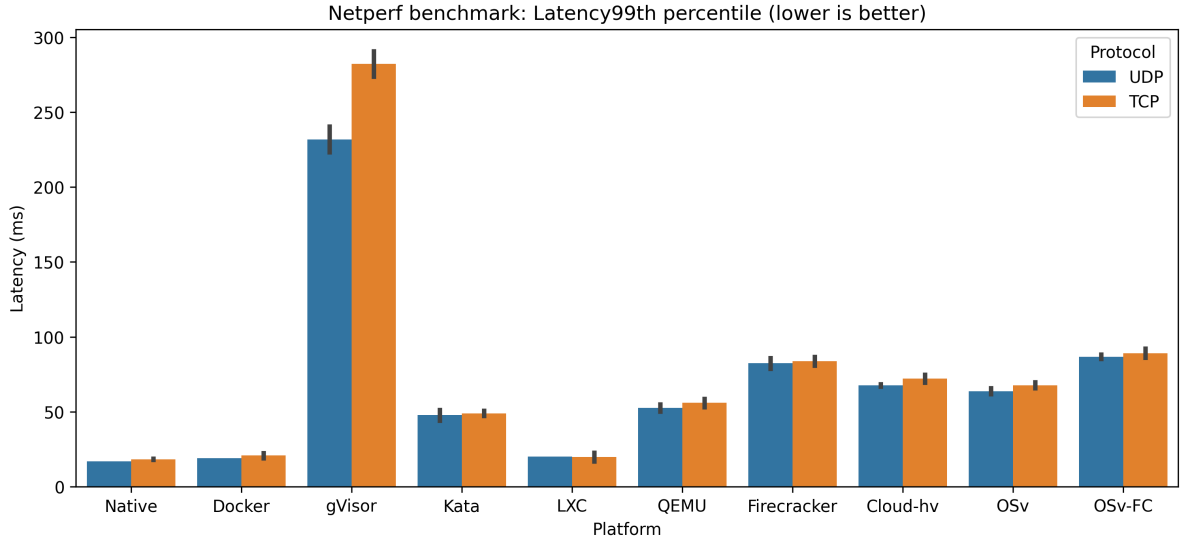

Figure A.5: Netperf network latency benchmark (99th percentile)

A.3. Memcached benchmark

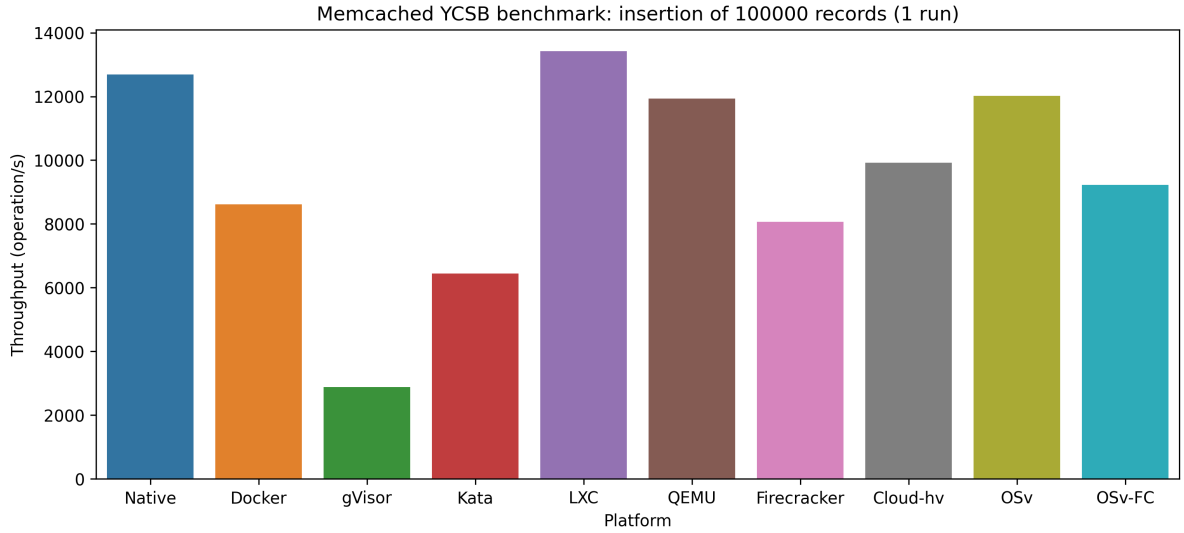

Figure A.6: Memcached YCSB benchmark, insertion stage (100000 records)

B

Additional tracing results

This chapter presents figures pertaining to the tracing of platforms as explained in Chapter 6 but were omitted in the main text for the sake of clarity.

A detailed breakdown of traced functions on a per-workload basis is given in Figure B.1.

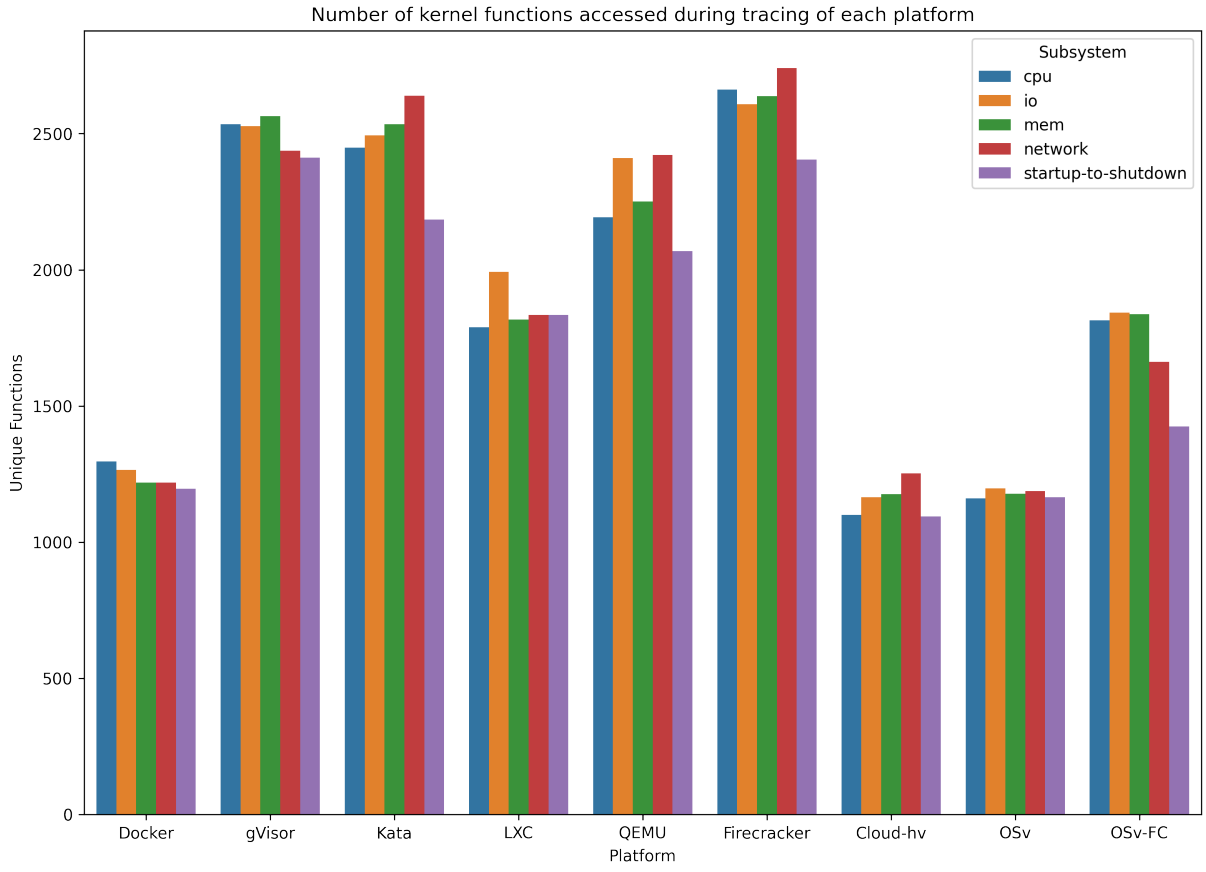

Figure B.1: Number of unique traced kernel function for each platform per workload.

A figure illustrating the number of traced kernel functions over time is given in Figure B.2. In this Figure, we see the number of new unique traced functions while starting a Firecracker hypervisor guest, waiting for 60 seconds, and then initiate a shutdown. This Figure indicates that not

necessarily all host kernel functions are executed at startup, but rather only occur after a period of idleness.

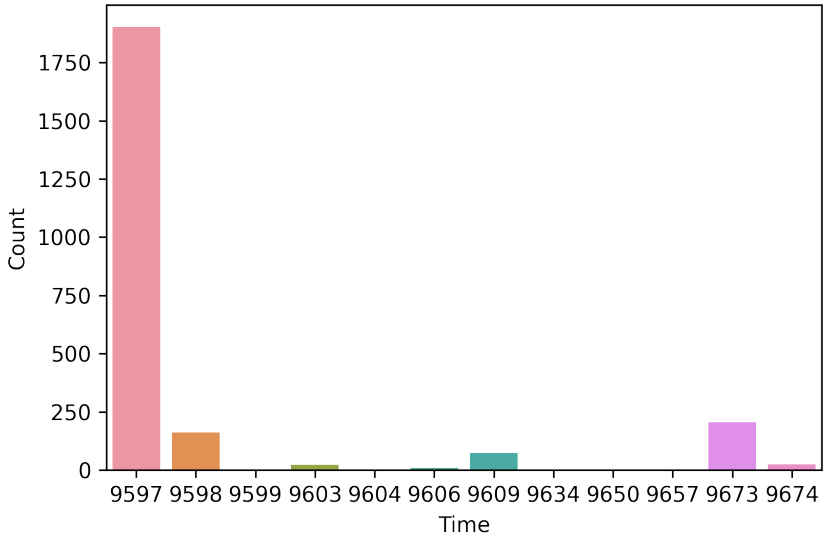

Figure B.2: Number of traced kernel functions over time in 1 second intervals. The vertical axis indicates the number of kernel functions that it has not seen during tracing of this process yet. Note that the horizontal axis, that indicates time in seconds (in this case offset by the boot time of the host machine), is not continuous.

Bibliography

[1] *A thorough introduction to eBPF – LWN*. https://lwn.net/Articles/740157/. [Online; accessed 25-January-2020]. 2017.

[2] Keith Adams and Ole Agesen. "A comparison of software and hardware techniques for x86 virtualization". In: *ACM Sigplan Notices* 41.11 (2006), pp. 2–13.

[3] Alexandru Agache et al. "Firecracker: Lightweight Virtualization for Serverless Applications". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 2020, pp. 419–434.

[4] Filipe de Aguiar Geissler, Fernanda Lima Kastensmidt, and José Eduardo Pereira Souza. "Soft error injection methodology based on QEMU software platform". In: *2014 15th Latin American Test Workshop-LATW*. IEEE. 2014, pp. 1–5.

[5] Nadav Amit and Michael Wei. "The design and implementation of hyperupcalls". In: *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 2018, pp. 97–112.

[6] Andrea Arcangeli, Izik Eidus, and Chris Wright. "Increasing memory density by using KSM". In: *Proceedings of the linux symposium*. Citeseer. 2009, pp. 19–28.

[7] *Architecture of the Kernel-based Virtual Machine (KVM)*. http://www.linux-kongress.org/2010/slides/KVM-Architecture-LK2010.pdf. [Online; accessed 2-December-2020]. 2020.

[8] Sergei Arnautov et al. "{SCONE}: Secure linux containers with intel {SGX}". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 689–703.

[9] Paul Barham et al. "Xen and the art of virtualization". In: *ACM SIGOPS operating systems review* 37.5 (2003), pp. 164–177.

[10] Maxime Bélair, Sylvie Laniepce, and Jean-Marc Menaud. "Leveraging kernel security mechanisms to improve container security: a survey". In: *Proceedings of the 14th International Conference on Availability, Reliability and Security*. 2019, pp. 1–6.

[11] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX Annual Technical Conference, FREENIX Track*. Vol. 41. 2005, p. 46.

[12] James Bottomley. "Exploring New Frontiers in Container Technology". In: *Linux Plumbers Conference* (2018).

[13] *capabilities(7) - Linux manual page*. https://man7.org/linux/man-pages/man7/capabilities.7.html. [Online; accessed 15-Dec-2020]. 2020.

[14] Tyler Caraza-Harter and Michael M Swift. "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*. 2020, pp. 101–113.

[15] Neal Cardwell et al. "BBR: Congestion-based congestion control". In: *Queue* 14.5 (2016), pp. 20–53.

[16] Emiliano Casalicchio and Stefano Iannucci. "The state-of-the-art in container technologies: Application, orchestration and security". In: *Concurrency and Computation: Practice and Experience* 32.17 (2020), e5668.

[17] *cgroups(7) - Linux manual page*. https://man7.org/linux/man-pages/man7/cgroups.7.html. [Online; accessed 27-November-2020]. 2020.

[18] Nitesh V Chawla et al. "SMOTE: synthetic minority over-sampling technique". In: *Journal of artificial intelligence research* 16 (2002), pp. 321–357.

[19] Yuchung Cheng et al. "RACK: a time-based fast loss detection algorithm for TCP". In: *draft-ietf-tcpm-rack-01. txt* (2016).

[20] Yeongpil Cho et al. "Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices". In: *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*. 2016, pp. 565–578.

[21] Theo Combe, Antony Martin, and Roberto Di Pietro. "To docker or not to docker: A security perspective". In: *IEEE Cloud Computing* 3.5 (2016), pp. 54–62.

[22] Maxime Compastié et al. "From virtualization security issues to cloud protection opportunities: An in-depth analysis of system virtualization models". In: *Computers & Security* 97 (2020), p. 101905.

[23] *Container Networking Interface Github repository*. https://github.com/containernetworking/cni. [Online; accessed 27-November-2020]. 2020.

[24] *Containerd Runtime Interface V2*. https://github.com/containerd/containerd/tree/master/runtime/v2. [Online; accessed 10-Dec-2020]. 2020.

[25] Michael J De Lucia. *A survey on security isolation of virtualization, containers, and unikernels*. Tech. rep. US Army Research Laboratory Aberdeen Proving Ground United States, 2017.

[26] *Docker overview – Docker architecture*. https://docs.docker.com/get-started/overview/. [Online; accessed 30-November-2020]. 2020.

[27] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. "Exokernel: An operating system architecture for application-level resource management". In: *ACM SIGOPS Operating Systems Review* 29.5 (1995), pp. 251–266.

[28] Wes Felter et al. "An updated performance comparison of virtual machines and linux containers". In: *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE. 2015, pp. 171–172.

[29] *ffmpeg - A complete, cross-platform solution to record, convert and stream audio and video*. https://ffmpeg.org/. [Online; accessed 2-December-2020]. 2020.

[30] *Firecracker homepage*. https://firecracker-microvm.github.io/. [Online; accessed 27-November-2020]. 2020.

[31] Brad Fitzpatrick. "Distributed caching with memcached". In: *Linux journal* 124 (2004).

[32] *Flexible I/O tester - fio*. https://git.kernel.dk/?p=fio.git;a=summary. [Online; accessed 12-January-2021]. 2021.

[33] Xing Gao et al. "Houdini's Escape: Breaking the Resource Rein of Linux Control Groups". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 1073–1086.

[34] Tom Goethals et al. "Unikernels vs containers: An in-depth benchmarking study in the context of microservice applications". In: *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*. IEEE. 2018, pp. 1–8.

[35] Christophe Guillon. "Program instrumentation with qemu". In: *1st International QEMU Users' Forum*. Vol. 1. Citeseer. 2011, pp. 15–18.

[36] *gVisor homepage*. https://gvisor.dev/. [Online; accessed 23-December-2020]. 2020.

[37] *gVisor platform documentation*. https://gvisor.dev/docs/architecture_guide/platforms/. [Online; accessed 7-December-2020]. 2020.

[38] Stefan Hajnoczi. *QEMU/KVM Architecure*. http://blog.vmsplice.net/2011/03/qemu-internals-overall-architecture-and.html. [Online; accessed 4-December-2020]. 2011.

[39] Stefan Hajnoczi. *QEMU/KVM Architecure - 2015 edition*. https://vmsplice.net/~stefan/qemu-kvm-architecture-2015.pdf. [Online; accessed 4-December-2020]. 2015.

[40] *Hub Vulnerability Scanning – Docker documentation*. https://docs.docker.com/docker-hub/vulnerability-scanning/. [Online; accessed 16-December-2020]. 2020.

[41] Intel Intel. "and IA-32 architectures software developer's manual". In: *Volume 3A: System Programming Guide, Part* 1.64 (64), p. 64.

[42] Intel Intel. *Software guard extensions programming reference, revision 2*. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf. 2014.

[43] *iperf3 networking benchmark tool homepage*. https://iperf.fr/. [Online; accessed 24-December-2020]. 2020.

[44] Gorka Irazoqui et al. "Wait a minute! A fast, Cross-VM attack on AES". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2014, pp. 299–319.

[45] Jay Jacobs et al. "Exploit Prediction Scoring System (EPSS)". In: *BlackHat USA '19* (2019).

[46] Madhur Jain. "Study of Firecracker MicroVM". In: *arXiv preprint arXiv:2005.12821* (2020).

[47] Zhiqiang Jian and Long Chen. "A defense method against docker escape attack". In: *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy*. 2017, pp. 142–146.

[48] Giorgos Kappes and Stergios V Anastasiadis. "Libservices: dynamic storage provisioning for multitenant I/O isolation". In: *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*. 2020, pp. 33–41.

[49] Paul A Karger et al. "A VMM Security Kernel for the VAX Architecture." In: *IEEE Symposium on Security and Privacy*. Citeseer. 1990, pp. 2–19.

[50] Eric Keller et al. "NoHype: virtualized cloud infrastructure without the virtualization". In: *Proceedings of the 37th annual international symposium on Computer architecture*. 2010, pp. 350–361.

[51] Avi Kivity et al. "OSv—optimizing the operating system for virtual machines". In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014, pp. 61–72.

[52] Ricardo Koller and Dan Williams. "An ounce of prevention is worth a pound of cure: ahead-of-time preparation for safe high-level container interfaces". In: *11th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. 2019.

[53] James Larisch, James Mickens, and Eddie Kohler. "Alto: lightweight vms using virtualization-aware managed runtimes". In: *Proceedings of the 15th International Conference on Managed Languages & Runtimes*. 2018, pp. 1–7.

[54] Ian M. Leslie et al. "The design and implementation of an operating system to support distributed multimedia applications". In: *IEEE journal on selected areas in communications* 14.7 (1996), pp. 1280–1297.

[55] Xin Lin et al. "A measurement study on linux container security: Attacks and countermeasures". In: *Proceedings of the 34th Annual Computer Security Applications Conference*. 2018, pp. 418–429.

[56] Nuno Lopes et al. “Container Hardening Through Automated Seccomp Profiling”. In: *Proceedings of the 2020 6th International Workshop on Container Technologies and Container Clouds*. 2020, pp. 31–36.

[57] *LXC Network configuration*. `https://linuxcontainers.org/lxd/docs/master/networks`. [Online; accessed 12-January-2021]. 2021.

[58] Filipe Manco et al. “My VM is Lighter (and Safer) than your Container”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 218–233.

[59] David Marshall. “Understanding full virtualization, paravirtualization, and hardware assist”. In: *VMWare White Paper* 17 (2007), p. 725.

[60] Antony Martin et al. “Docker ecosystem–vulnerability analysis”. In: *Computer Communications* 122 (2018), pp. 30–43.

[61] Ilias Mavridis and Helen Karatza. “Lightweight Virtualization Approaches for Software-Defined Systems and Cloud Computing: An Evaluation of Unikernels and Containers”. In: *2019 Sixth International Conference on Software Defined Systems (SDS)*. IEEE. 2019, pp. 171–178.

[62] John D McCalpin. “Stream benchmark”. In: *www.cs.virginia.edu/stream/ref.html* 22 (1995).

[63] *Micro-optimizing KVM VM-exits*. `https://people.redhat.com/~aarcange/slides/2019-KVM-monolithic.pdf`. [Online; accessed 1-December-2020]. 2019.

[64] *namespaces(7) - Linux manual page*. `https://man7.org/linux/man-pages/man7/namespaces.7.html`. [Online; accessed 27-November-2020]. 2020.

[65] *OSv Github repository*. `https://github.com/cloudius-systems/osv`. [Online; accessed 27-November-2020]. 2020.

[66] Diego Perez-Botero, Jakub Szefer, and Ruby B Lee. “Characterizing hypervisor vulnerabilities in cloud computing servers”. In: *Proceedings of the 2013 international workshop on Security in cloud computing*. 2013, pp. 3–10.

[67] Max Plauth, Lena Feinbube, and Andreas Polze. “A performance survey of lightweight virtualization techniques”. In: *European Conference on Service-Oriented and Cloud Computing*. Springer. 2017, pp. 34–48.

[68] Gerald J Popek and Robert P Goldberg. “Formal requirements for virtualizable third generation architectures”. In: *Communications of the ACM* 17.7 (1974), pp. 412–421.

[69] Gerald J Popek and Charles S Kline. “The pdp-11 virtual machine architecture: A case study”. In: *Proceedings of the fifth ACM symposium on Operating systems principles*. 1975, pp. 97–105.

[70] Daniel Price and Andrew Tucker. “Solaris Zones: Operating System Support for Consolidating Commercial Workloads.” In: *LISA*. Vol. 4. 2004, pp. 241–254.

[71] *QEMU 1.0 release*. `https://lwn.net/Articles/470341/`. [Online; accessed 1-December-2020]. 2020.

[72] Theo de Raadt. *Pledge: a new mitigation mechanism*. 2015.

[73] Andrew R Riddle and Soon M Chung. “A survey on the security of hypervisors in cloud computing”. In: *2015 IEEE 35th International Conference on Distributed Computing Systems Workshops*. IEEE. 2015, pp. 100–104.

[74] Davood Ghatreh Samani et al. “The Art of CPU-Pinning: Evaluating and Improving the Performance of Virtualization and Containerization Platforms”. In: *arXiv preprint arXiv:2006.02055* (2020).

[75] Ioannis Sfyrakis and Thomas Groß. “UniGuard: Protecting Unikernels using Intel SGX”. In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2018, pp. 99–105.

[76] Rui Shu, Xiaohui Gu, and William Enck. “A study of security vulnerabilities on docker hub”. In: *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. 2017, pp. 269–280.

[77] Udo Steinberg and Bernhard Kauer. “NOVA: A microhypervisor-based secure virtualization architecture”. In: *Proceedings of the 5th European conference on Computer systems*. 2010, pp. 209–222.

[78] *Sysbench Github repository*. `https://github.com/akopytov/sysbench`. [Online; accessed 24-December-2020]. 2020.

[79] *systemd.nspawn(5) - Linux manual page*. `https://www.man7.org/linux/man-pages/man5/systemd.nspawn.5.html`. [Online; accessed 08-February-2021]. 2021.

[80] Jakub Szefer et al. “Eliminating the hypervisor attack surface for a more secure cloud”. In: *Proceedings of the 18th ACM conference on Computer and communications security*. 2011, pp. 401–412.

[81] OASIS Virtual I/O Device (VIRTIO) TC. *Virtual I/O Device (VIRTIO) specification*. `http://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.pdf`. [Online; accessed 4-December-2020]. 2019.

[82] *tini: A tiny but valid ‘init‘ for containers - Github repository*. `https://github.com/krallin/tini`. [Online; accessed 04-February-2021]. 2021.

[83] *Tinymembench Github repository*. `https://github.com/ssvb/tinymembench`. [Online; accessed 2-December-2020]. 2020.

[84] *ttRPC: GRPC for low-memory environments - Github repository*. `https://github.com/containerd/ttrpc`. [Online; accessed 23-December-2020]. 2020.

[85] Iulia Turc et al. “Well-Read Students Learn Better: On the Importance of Pre-training Compact Models”. In: *arXiv preprint arXiv:1908.08962v2* (2019).

[86] *Using virtio-fs on a unikernel*. `https://www.qemu.org/2020/11/04/osv-virtio-fs/`. [Online; accessed 01-February-2021]. 2021.

[87] Jacobsen V. and Felderman R. “Speeding up networking”. In: *Linux Conference Australia* (2006).

[88] Eric Van Hensbergen and Ron Minnich. “Grave Robbers from Outer Space: Using 9P2000 Under Linux.” In: *USENIX Annual Technical Conference, FREENIX Track*. 2005, pp. 83–94.

[89] *Virtio-fs for Kata Containers storage*. `https://vmsplice.net/~stefan/stefanha-virtio-fs-kata.pdf`. [Online; accessed 01-February-2021]. 2021.

[90] *Vulnerability scanning for Docker local images – Docker documentation*. `https://docs.docker.com/engine/scan/`. [Online; accessed 16-December-2020]. 2020.

[91] Zhi Wang and Xuxian Jiang. “Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity”. In: *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 380–395.

[92] Dan Williams, Ricardo Koller, and Brandon Lum. “Say goodbye to virtualization for a safer cloud”. In: *10th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. 2018.

[93] Ethan G Young et al. “The true cost of containing: A gVisor case study”. In: *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. 2019.