Exploring the Performance-Isolation Trade-off for Isolation Mechanisms

Antonios Sklavos
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
`a.sklavos@student.vu.nl`

Matthijs Jansen
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
`m.s.jansen@vu.nl`

Animesh Trivedi
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
`a.trivedi@vu.nl`

Abstract

Isolation mechanisms can be traced to the early days of computing, where processes shared hardware resources and interacted with other processes. Gradually, the emergence of virtual machines as we know them today made it possible to provide a complete simulation of the host hardware to a virtualisation-agnostic guest operating system. Nowadays, isolation mechanisms play an important role in cloud computing's resource pooling needs, where customers expect complete isolation and maximum performance. Different isolation needs and choices have led to different implementations of mechanisms that minimise the isolation overhead and maximise performance. This survey briefly overviews commonly used mechanisms and identifies compromises made to balance performance and isolation.

1 Introduction

The concept of virtualisation can be traced to the late 1950s' where tools like Multics introduced the notion of multiprogramming [46] [44] [18]. There, a privileged kernel was responsible for managing hardware resources and restricting access to any software outside of it, essentially providing an early form of isolation [46]. Nowadays, one of the driving forces of isolation is the realisation of a fair environment among cloud computing tenants [60].

Isolation is a general term that can be broken down into the three specific notions of Security isolation, Fault isolation and Performance isolation (Table 1). **Security isolation** involves the mechanisms and policies that limit access between different guests and the host machine [54]. As such, a malicious program will be confined to its own environment and will be unable to harm the host or other guests. **Fault isolation** refers to the ability to confine the fault of a virtual machine in order to stop it from interfering with other virtual machines or the host: an (accidental) failure of a guest should not affect the whole system or other hosts. **Performance isolation** is concerned with fairly distributing the host resources to the guests so as not to benefit a single one, for example, due to guests going over their quotas [60].

Isolation Kind	**Description**	**Example**
Security Isolation	Restrict access between processes	`chroot` Unix operation
Fault Isolation	Limit the impact of failures	Using a separate kernel for every isolation unit
Performance Isolation	One's usage does not affect the performance of others	`cgroups` Unix tool

Table 1: Comparison of isolation types

Isolating processes from different users is challenging without compromising performance, and no one-solution-fits-all mechanism exists. In security-critical scenarios, full virtualisation, where processes are isolated in separate, virtualised and unmodified operating systems, can offer a solution. Performance overhead is unavoidable in this case, stemming from the abstraction of virtualisation: The hypervisor which manages the virtualised operating system is an intermediary that must handle the translation of virtualised instructions to physical ones. On the contrary, having relaxed security requirements offers the ability to limit some isolation-related overheads (for example, the need for separate operating systems for different users). Consequently, performance and isolation can be contradictory, forming a decision dilemma for isolation mechanism designers. While isolation is a qualitative measure (in contrast to quantitative performance), comparing different mechanisms and gaining insight into how they implement their notion of isolation is possible.

The challenges above lead us to explore surveys and publications related to open-source isolation mechanisms in order to answer the following research question:

RQ1: How do isolation mechanisms balance the trade-off between performance and isolation?

We find this research question valuable due to the knowledge gap it fills: it is often vague how different implementation decisions affect the trade-off. Additionally, answering the question can contribute to informed decision-making for those interested in using such tools.

Surveys are a great way to answer such questions. Although many surveys examine a combination of the isolation mechanisms presented below, advances in cloud technology and infrastructure make an updated overview necessary. In contrast to most surveys, we are interested not in a purely numerical performance difference but in the underlying reasons contributing to the different levels of isolation, which is at odds with performance.

A contribution of this survey is reaffirming the conclusion that different scenarios favour different tools. We found that ease of use is an important variable regarding isolation mechanisms. That led us to produce a performance-isolation-usability graph for each tool, which visually encapsulates this survey's findings.

The survey is organised as follows: Section 2 goes over related surveys in the literature. Section 3 explains our methodology. Section 4 presents an overview of four different categories of isolation mechanisms. In the four Sections following, we present representative examples of mechanisms of each category: Hypervisors (Section 5), Containers (Section 6), Secure Containers (Section 7) and Unikernels (Section 8). A conclusion is provided in Section 9.

2 Related Surveys.

For each of the surveys mentioned below, ranked by their respective number of citations on Google Scholar, we provide a small introduction and a comparison with the present survey. While they all touch upon isolation in virtualisation, none explicitly discusses the trade-offs of isolation and performance of all the tools we mention in Section 4.

Virtualization: Issues, security threats, and solutions. The survey by Pearce et al. [44] provides the basic concepts of virtualisation, discusses a threat model, and proposes solutions. Compared to our survey, it does not focus on a specific technology; it analyses virtualised platform threats in a general fashion. A contribution of this survey is the recognition that the implicit trust operating systems show to hardware is a vulnerability when it comes to virtualisation since the physical hardware is emulated to a degree by software.

Container Security: Issues, Challenges, and the Road Ahead. The survey by Sultan et al. [57] focuses extensively on containers and their security issues. It presents a broad view of the various mechanisms that enable the containers' operation and how different protection requirements lead to solutions.

A study of security isolation techniques. A significant contribution of this survey [54] is the classification of different isolation techniques into a hierarchy. Compared to the other related surveys, compiler and code rewriting techniques are mentioned. Additionally, the trade-offs concerning speed are discussed in a few use cases. While the survey is very extensive in the techniques it mentions, the detail of them can be considered lacking. Additionally, advances in the virtualisation field have led to the current emergence of popular tools (e.g. Firecracker (Section 5.4) and Kata Containers (Section 7.1)), which did not exist when the survey was conducted.

Virtualization in Cloud Computing: Moving from Hypervisor to Containerization—A Survey. The survey by Bhardwaj et al. [6] explores the emergence of containerisation and how it aims to be a better alternative to VMM solutions. Additionally, it compares the performance of different hypervisor and container technologies. It is important to note that there is a lack of modern isolation mechanisms specifically designed for cloud computing, such as Firecracker.

Security Issues and Challenges for Virtualization Technologies. The hypervisor-security-focused survey by Sierra-Arriaga et al. [55] provides a classification of the security issues that specifically arise from virtualisation. Performance is very briefly discussed as it is not the survey's focus.

An Exhaustive Survey on Security Concerns and Solutions at Different Components of Virtualization. With a focus on hypervisors, this survey [43] provides an overview of the various virtualisation-specific attacks and organises said attacks into a taxonomy. An extensive list of Xen (Section 5.5) and KVM (Section 5.3) recorded vulnerabilities are mentioned. Additionally, the solutions to those vulnerabilities are reviewed while discussing their limitations. The performance of isolation mechanisms is not the primary concern.

The Serverless Computing Survey: A Technical Primer for Design Architecture. The work of Li et al. [30] is an updated overview of the architecture of common virtualisation mechanisms used in cloud environments. It judges tools not only by their performance (boot times) and isolation but also by their flexibility. The survey's scope of view of isolation mechanisms is through serverless functions, which, while a big part of virtualisation, is not the only one.

A Fresh Look at the Architecture and Performance of Contemporary Isolation Platforms. The review of van Rijn et al. [59] laid the foundation for the classification of the isolation mechanisms as mentioned in section 4 of this survey. Additionally, it classifies the isolation of those mechanisms based on the metric of the Horizontal Attack Profile and extensively compares their performance. It only gives an overview of some tools and does not explain the reasons contributing to their performance and isolation.

3 Survey Methodology

In this section, we describe the survey methodology we followed. We wanted to research the performance and isolation of isolation mechanisms and searched for papers on this topic; we started with a few like [36] and [2]. We identify the following fundamental isolation mechanisms, explained in the next section, and find established systems such as QEMU (Section 5.2) and Xen (Section 5.5) for VMMs, and Docker Containers (Section 6.1) and LXC (Section 6.2) for containers; paper citations on Google Scholar and publishing organisations' reputation were taken into account in our paper selecting process.

Having decided on the tools to mention based on reputation, prominence, and influence on other tools, we studied extensively relevant papers in order to get a deep understanding of their architecture. After careful evaluation, we determined that certain papers were outdated and subsequently excluded them from our analysis.

4 Overview of Current Landscape of Isolation Mechanisms

In the next four sections, we go through the four major categories of Isolation mechanisms, as defined by [59] (summary in Table 2):

Hypervisors, which enable virtualisation. With virtualisation, virtual machine monitors (VMMs - also known as hypervisors) present a software abstraction of physical machines to virtual machines; virtual machines contain their own operating system with all dependencies needed to execute a process in an isolated execution environment.

Containers, which utilise (mostly) Linux mechanisms to sandbox processes. Containerisation (also known as OS-level virtualisation) is a technique that emerged as an alternative to virtual machine monitors; Container technologies sandbox user-level processes with techniques such as filtering system calls and limiting access to hardware and software resources.

Secure containers, which try to balance the performance of containers with the isolation of hypervisors. While virtual machines and containers (in their simplest form) lie on opposing ends of the isolation-performance spectrum, attempts to get the best of both worlds have been made by tools like Kata containers (Section 7.1) or by running containers inside virtual machines.

Unikernels, which are single-address space machine images built by compiling applications and linking them with libraries that provide OS functionality. Unikernels are optimised for single-purpose applications.

For each isolation mechanism we have identified, we go through its limitations and strong points, and additionally, we try to answer **RQ1**, namely, how it balances isolation and performance.

Category	**Approach**	**Performance - Isolation**
Hypervisors	Simulate Hardware	Simulating hardware can add performance overhead; Isolation increases.
Containers	Sandbox processes that share same kernel	Performance overhead minimised; Isolation can suffer.
Secure Containers	Isolation of crucial components through hardware mechanisms	Performance generally worse than containers; Isolation improved.
Unikernels	purpose-built machine images build with library os's	application-specific optimisations limit hypervisor overhead.

Table 2: Software Isolation Approaches

5 Hypervisors

Broadly speaking, hypervisors (also referred to as Virtual Machine Monitors - VMMs) are pieces of software (also firmware or hardware) that enable the operation of virtual machines by making the hardware resources of their hosts available to them [55]. Having privileged access to hardware resources, they act as an intermediary between the hardware and virtual machines and have the ability to enforce security and performance policies [56] [18]. The VMMs are expected to behave similarly to physical hardware [44]; they export an abstraction of the physical hardware that enables any software to run on the hardware to be also able to run on the abstraction [18]. The isolation they offer between different systems

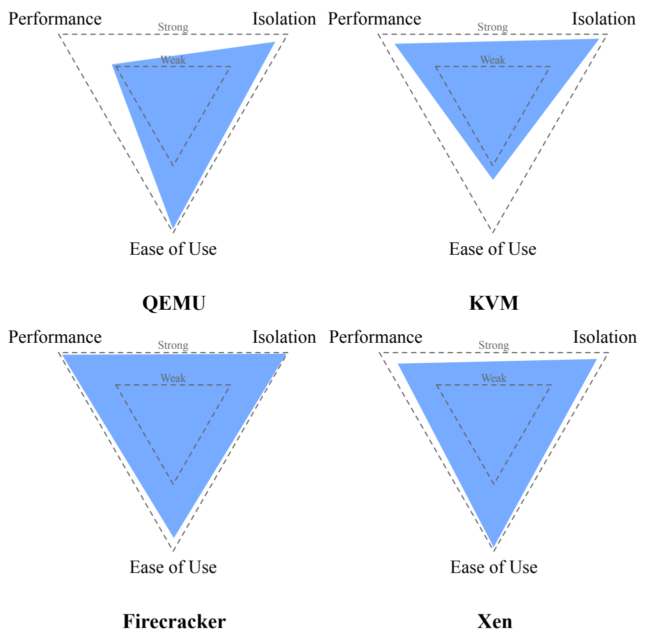

Figure 1: Qualitative Comparison of Hypervisors

Hypervisor Name	**Developer**	**Host OS**	**Notable Features**
QEMU	QEMU team	Windows, Linux, MacOS, Solaris, FreeBSD, OpenBSD, BeOS	Focus on feature completeness
KVM	Linux Kernel Community	Linux, FreeBSD, illumos	near-native performance due to hardware virtualisation extensions support
Firecracker	AWS developers	Linux	purpose-built for serverless
XEN	Linux Foundation, Intel	Linux	Para-virtualisation support

Table 3: Hypervisors Overview

is arguably their most important contribution [56]. As hypervisors are made to serve different objectives by organisations and groups with varying goals, many of their concepts can differ [56].

Hypervisors offer great convenience of use, which is a significant advantage for their users since they require no modification to the guest's VM image [22]; as a result, one can deploy the same VM image on different VMMs. They allow for a high degree of isolation between the hardware and the virtual machine, as well as between different software on the same hardware [44].

A significant disadvantage comes from the isolation itself: having each VM carry its copy of the OS and other resources limits the number of VMs one can deploy on the same system [57]. Additionally, the abstraction of the hardware that the VMM provides creates an overhead that a higher layer of software will not be able to remove [31].

Figure 1 contains one graph for each of the hypervisors mentioned below. The diagrams aim to visually compare the three tools extracted from the cited papers and offer a qualitative comparison. It is important to note that the assessments of usability and isolation in the graphs are qualitative in nature and do not involve quantitative measurements. The axis 'ease of use' represents the amount of modifications a piece of software needs to be deployed on a tool; for example, QEMU can run processes in a variety of different CPU architectures and requires no modification to a virtual machine image (for the most part). On the contrary, MirageOS (Section 8.3), a Unikernel, requires porting to software to the OCaml programming language - a great barrier to adaptation.

To conclude, Firecracker displays the best balance of Hypervisors: While it lacks device emulation and supports only (modern) Linux and OSv guests, it is optimised for performance and goes above and beyond in the domain of isolation. KVM has been known to be challenging to operate as a standalone solution, and it is common to use it in combination with other tools, like QEMU. Similar graphs are provided in each category of isolation mechanisms.

Type-1 and Type-2 Hypervisors. Historically, Hypervisors have been categorised in Type-1 (bare metal) and Type-2 (hosted) (Figure 2) [19] [44]. On the one hand, classic System VMMs (Type-1) are placed on bare metal with the utmost privileges (the guests run with fewer privileges); they are responsible for intercepting the system calls that virtual machines make concerning system access [56] [39]. On the other hand, hosted VMs (Type-2) are no more than programs installed in an operating system (host), which they use to get access to system drivers [56]; due to the additional level of abstraction [22], a performance penalty might be introduced.

As hypervisors have gotten more complex, the distinctions become increasingly unclear; for example, KVM (Section 5.3) can be seen as turning the Linux kernel into a Type-1 hypervisor.

Paravirtualisation. Paravirtualisation refers to a technique where the guest OS is modified, making it aware of its virtualised nature; this allows better performance by offloading tasks from the guest domain to the host domain [38] [22].

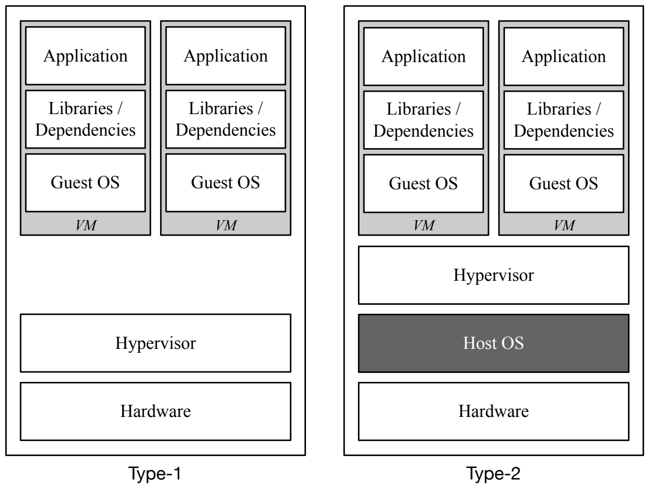

Figure 2: Type-1 and Type-2 Hypervisors Architecture

While it offers performance gains, the requirement for guest OS modification can be an obstacle. It is thus a trade-off between performance and portability [46].

5.1 A Primer on Hypervisor Attack Surface

Below we introduce some areas of Hypervisors' attack surface as found in the literature [58] [43] [45].

VM exits. Arguably, VM exits are the most significant hypervisor attack surface [58]. VM exits occur when the hypervisor stops the VM execution and handles host OS operations; different situations can lead to different VM exists. VM exists are very frequent and occur even when the VM is idle. Thus, the VM communicates with the host os indirectly through the hypervisor. Consequently, it constitutes an attack surface since it offers a window of communication between the VM and the hypervisor; A malicious VM process can exploit a hypothetical bug in how a hypervisor handles certain events.

Virtualised devices. Virtualised devices can pose a threat to virtual machine hypervisors [45] [55]. They are abstractions made to emulate physical devices and, as such (being software), can be exploited; for instance, an attacker could use a virtualised network interface card vulnerability to intercept network traffic from other VMs. Notably, device emulators, such as printers, mice, VGA Etc., are not needed in cloud environments and are usually removed to minimise vulnerabilities.

Large TCB. A large Trusted Computing Base, that is, the components critical to a machine's security, can expose a significant attack surface; the need for Hypervisors to provide some OS functionality directly influences this [2]. The presumption that the OS trusts the hardware no longer holds; the hardware is now the VMM which the OS should not fully trust [55] [44]. The break in trust creates a significant vulnerability for the OS, making the VMM a single point of failure which can lead to disastrous results [44].

5.2 QEMU

QEMU [4] is one of the most common hypervisors due to its free and open-source nature and feature completeness. It can leverage KVM (see 5.3) to enable hardware-assisted virtualisation. Additionally, it offers emulation of various CPU instruction sets and devices. Due to its flexibility and features, QEMU has been used as the foundation for other projects, like Kata Containers (Section 7.1) [2].

The abundance of emulated devices is a double-edged sword: while contributing to feature completeness, they significantly enlarge QEMU's attack surface. QEMU has been criticised for its large and complex codebase (> 1.4 MLOC), and there have been (unsuccessful) attempts by third parties to provide a stripped-down version of QEMU [2]. Contributing to its large attack surface, QEMU's large codebase further decreases its performance-isolation score.

5.3 KVM

KVM is a loadable Linux kernel module which has been part of Linux since version 2.6.20 (February 2007) [25] [49] [45]. It can utilise supported x86 processor virtualisation technologies, like direct memory access [38], to offer near-native machine performance. With KVM, Linux becomes a type-1 hypervisor, and every virtual machine is implemented as a Linux process [15].

Since I/0 emulation is notoriously underperforming [25], KVM supports paravirtualised devices using `virtio` [50]; it can also emulate I/O devices with the support of QEMU [15]. While KVM has been praised for its overall performance, it has been criticised for its configurability [15]. KVM implements isolation and security with a combination of sVirt and SELinux [49], and each VM in KVM is run with its own kernel [9]. SELinux is a Linux Security Module [64], a framework which allows the placement of 'hooks' just before access to important system calls in order to enforce security. Initially developed by the NSA, It offers Role-Based Access Control and Type Enforcement. Optionally, it supports Multilevel security [64]. `sVirt` extends SELinux's capabilities, allowing Mandatory Access Control (MAC) security to be applied to guest VMs and preventing manual labelling errors [51].

5.4 Firecracker

Firecracker is a hypervisor developed by Amazon to serve their serverless infrastructure needs. It allows the creation of microVMs with negligible CPU and minimal (9MB per VM) overhead; a full-fledged microVM can be launched in less than 125*ms* [2]. Google's crossvm (Paragraph 5.6.2) served as the initial building block of the project; KVM is also used due to its code maturity and performance.

Firecracker is optimised for serverless deployments [35]; for example, it cannot boot arbitrary kernels, it does not offer a BIOS, and it does not support VM migration [2]. Moreover, very little device emulation is provided; the only included devices are network and block devices, serial ports, and a partial PS/2 keyboard emulation. The above choices are appropriate and permissible due to Firecracker's serverless-first focus; it is actively used in production as the backbone of AWS Lambda. In a nutshell, Firecracker compromises on flexibility and feature completeness in order to focus on overhead, security and fast startup.

Security is of paramount importance for Firecracker, and extensive caution has been taken to maximise it. Mutually distrusting users deploy on AWS Lambda and expect their code and data to be inaccessible to other users. As a first example, Firecracker removes support for legacy drivers since they can pose a security vulnerability which a malicious VM could potentially leverage to gain access to the host system. Next, it eliminates potential security issues by only supporting recent kernel versions (v4.14+). Additionally, Firecracker was implemented using the (memory-safe) Rust programming language to guarantee memory safety. Its minimalist implementation can further reduce the attack surface.

To further secure the platform, known side-channel attack vulnerabilities taken advantage of by exploits such as Spectre [27] are mitigated with techniques such as disabling Hyper-Threading and disabling swap and kernel same-page merging.

The above approaches do not impact performance while significantly isolating the virtual machine from the host. Optionally, one can further isolate a Firecracker virtual machine using the provided jailer process, which applies a wrapper to Firecracker and places it into a sandboxed environment isolating it and dropping privileges.

5.5 XEN

Xen is a very popular isolation mechanism with widespread production use; it can operate in full virtualisation mode, paravaritualisation or other variations [36] [14] [9] [67]. Xen has kept the Application Binary Interface unaltered to allow guest applications to run without changes [46]. Here we focus on its paravirtualised capabilities, which were the focus of its very influential introduction paper [3]; Full virtualisation mode on Xen uses Qemu to emulate devices [67].

Paravirtualisation refers to an abstraction that allows improved performance (compared to full virtualisation) at the cost of having to modify the guest OS image (albeit not the guest applications) [3]. Paravirtualisation does not require CPU-supported virtualisation extensions, making it possible to run on hardware that does not support virtualisation [67]. Support for full virtualisation was never part of the x86 architectural design (see contradiction), which requires overhead-inducing handling and the addition of extra complexity [3].

Security and isolation. Xen version 4.5 introduced Virtual Machine Introspection (VMI), a technique that allows for monitoring the activity of a VM by an external tool with minimal overhead [18]. More specifically, this allows the VMM to act as an intrusion detection system while evading attacks targeting those systems; it uses hardware-level state and interrupts (can be observed from outside of VM) to interpret them into os-level semantics [18].

Arguably, when multiple VMs are operating in a machine, it is not enough to just fairly allocate resources between VMs, since device drivers' consumption from the hypervisor at the request of VMs can break performance guarantees. The work of [21] introduces a toolchain (metrics aggregator, scheduler and control mechanism) that enables the monitoring of total resource usage of VMs and, thus, enforcement of resource isolation guarantees.

Performance. One study found that the Unikernel benefits come at zero to little cost to performance in all cases [33]. A different study showed that Unikernels are an option for replacing containers by placing language runtimes into a hypervisor [34]

5.6 Notable mentions

The following two isolation mechanisms came up during our research. However, their trade-off between isolation and performance could not be sufficiently appraised due to very limited published research based on them.

5.6.1 Cloud Hypervisor

Cloud Hypervisor is a Virtual Machine Monitor optimised for cloud workloads [23]. Being implemented in Rust, it extends its capabilities to address the broader requirements of cloud environments. One of its notable advantages over Firecracker is the expanded support for device emulation, enabling seamless integration with a broader range of hardware components. Additionally, in contrast to Firecracker, it can be run on Windows.

5.6.2 crosvm

crosvm is a KVM-based virtual machine monitor developed by Google. It is primarily utilised as the primary virtual machine monitor of ChromeOS. It served as the foundation of Firecracker after vigorous refactoring; for example, device emulation was restricted [2].

Tool Name	Developer	Host OS	Notable Features
Docker	Docker, Inc.	Windows, Linux, MacOS	Robust ecosystem
LXC	Virtuozzo, IBM, Google and individuals	Linux	Focus on system containers that provide a VM-like environment
OpenVZ	Virtuozzo, OpenVZ community	Linux	guests individually implement filesystem and network functionality

Table 4: Containerisation Solutions Overview

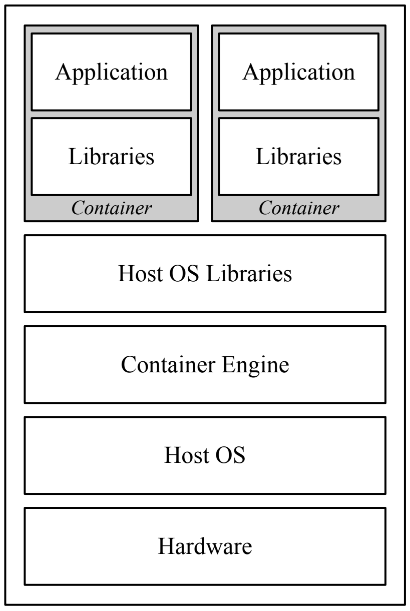

Figure 3: Container Architecture

6 Containers

Containers allow programs to run in isolated environments, sharing the host operating system kernel, filesystem and resources [57] [37] [38]. They originally came into view to replace VMs as faster alternatives [57].

Containers use a combination of Linux kernel extensions and other tools to confine processes into their own execution environment. Most notably, kernel `namespaces` can be used to limit the resources (e.g. process ids, file names) that a process has access to. `cgroups` is responsible for limiting the resource usage (e.g. CPU, disk access) of a process. `chroot`, is a Linux tool that restricts a process' access to a specific directory. In fact, `chroot`, as well as FreeBSD `jails`, can be regarded as early forms of containerisation. Figure 3 shows the general architecture of Containers.

Containers can offer advantages in the performance domain: network, disk, computing, and memory performance overhead can be zero to little [66] [40]. In fact, their overhead can be so small that the performance can be comparable with the equivalent bare-metal one [6]. This is a direct consequence of the shared kernel: the syscall execution path is shortened [37]. Due to their more negligible overhead compared to VMMs (albeit not in every case [31]), they allow for increased density - the amount of isolation unit instances that a machine can host, as well as smaller disk images [40]. Moreover, an advantage of containers over other isolation methods is the ease of deploying different versions of the same applications they provide [37]. Finally, containers provide efficient resource use by avoiding code duplication [37].

Containers do not come without disadvantages. Isolation is typically limited (compared to Hypervisors 5); for example, if a container stresses the kernel with system calls, it will not be able to handle system calls from other containers (performance isolation). A malicious container can take advantage of this by causing a denial-of-service attack [36] (security isolation); Overprovisioning can lead to DoS attacks also [44]. Due to its shared nature, a compromised kernel affects all containers [38]. Additionally, they are generally stateless (no direct memory sharing) [53]; clearly, the goal of isolation is incompatible with the concept of sharing memory. In their case, isolation is understood to be lacking enough that deploying a container inside a hypervisor has been common practice [36] [38] [8].

Containers suffer from slow cold start times (for example, when scaling) [53], which can be avoided by recycling old containers, further sacrificing isolation. Notably, it has been argued that the only way to guarantee container threat containment is to use full virtualisation instead (Section 5) [39]. Non-namespace-aware system calls originating in containers are a significant susceptibility for containers since they can expose sensitive information [52]. While tools like Docker (Section 6.1) and LXC (Section 6.2) are versatile and easy to use, that comes at the cost of decreased security [40], demonstrated by side-channel attacks like Spectre [30].

Comparing the two tools (Figure 4), they display similar performance overall (LXC boot-up time is inferior [59]), but docker shows increased attack surface due to its enlarged toolchain [10]; docker is supported on all major platforms (in contrast to LXC) and is well documented and supported.

6.1 Docker containers

Docker is a prevalent containerisation platform. It uses the `runc` container engine [8]. Docker is notable for its ease of

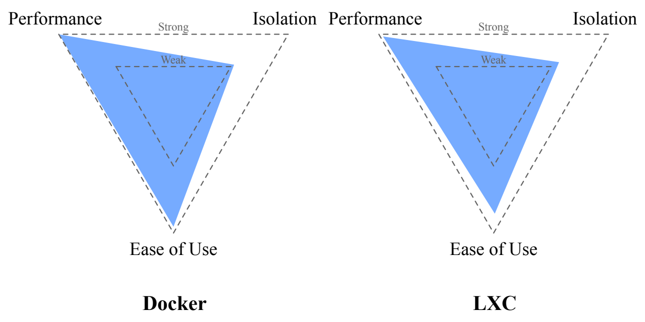

Figure 4: Qualitative comparison of Containers

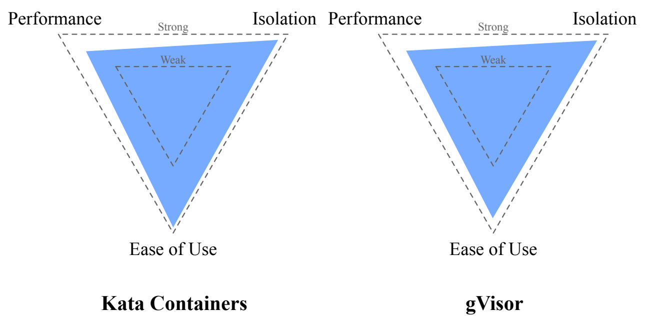

Figure 5: Qualitative comparison of Secure Containers

use and near-zero performance overhead, which directly contributed to its popularity [40] [59] [37]. Early versions of docker built upon the LXC (Section 6.2) container runtime [5]. While Docker is marketed as a microservices orchestration platform, many developers use it to boost their productivity [37].

The containerised applications share the host OS kernel, which is responsible for providing isolation with tools like namespaces and cgroups [37] [57]. In addition to the above, kernel `Capabilities` are used, which allows limited access to objects managed by the kernel and `Apparmor`, which limits the resources available to programs. Moreover, a small portion of system calls is blacklisted using a `seccomp` profile [57].

While Docker (and containerisation platforms in general) display minuscule overhead, this usually comes at the cost of isolation. First and foremost, the shared kernel becomes a single point of failure: a potential kernel security breach can affect the total amount of containers deployed on the host [38]. Also, since docker containers run in user-space (privilege ring 3 [1]), any other application could potentially disrupt them [38]. Notably, the whole Docker toolchain constitutes its attack surface [10], for example, by the widespread practice of using external dependencies (docker images) which end up in production code [37].

6.2 LXC

Linux Containers (LXC) is a low-level container runtime that utilises Linux Kernel features like namespaces, cgroups and capabilities to containerise applications [8] [16] [15] [32]. Its main difference with Docker (Section 6.1) is its ability to simulate a standard Linux environment closely [59]. As a result, it makes it easy to create multiple execution environments within a single OS [48].

Linux containers suffer from weak isolation: applications are inherently more isolated when deployed on VMs (Section 5) compared to having to share the same kernel. This can lead to scenarios where an application leaks sensitive information through system calls [15].

6.3 OpenVZ

OpenVZ is another commonly used containerisation technology similar to LXC.

Virtual Private Servers (VPS) serve as the underlying abstraction of isolation. OpenVZ uses user-level tools and a modified kernel to operate [9]. The modified kernel traps system calls and rewrites the results [12]. It differs from LXC and Docker containers since the guests individually implement filesystem and network functionality, among others [54]. OpenVZ enforces performance isolation by assigning an I/O priority in its I/O scheduler [9].

7 Secure Containers

Secure containers (also mentioned as sandboxed container technologies) are security-oriented containerisation platforms. They aim to balance performance and isolation using kernel features like `namespaces` and leveraging hardware-based isolation [59]. Even though Secure containers advertise container-like performance, in most cases, they perform worse than containers and VMMs - especially regarding I/O operations [59].

We mention Kata containers (Section 7.1) and gVisor (Section 7.2). Comparing the two (Figure 5), Kata containers blend in the container ecosystem by design, getting an increased ease of use rating.

7.1 Kata containers

Kata Containers is a tool that aims to combine the benefits of VMs and containers in a single solution [47] [16] [24]. Each workload is run in a container and further isolated in its separate VM (and thus separate kernel - the containers' 6 largest attack surface) [47] [61]. Kata containers is essentially a container-optimised VM [61]. It is the combined result of Intel Clear Containers with Hyper runV.

Its runtime, `kata-runtime`, supports the industry-standard OCI container format allowing it to work seamlessly with Docker [16]. It isolates network, I/O and memory due to

Tool Name	Developer	Host OS	Notable Features
Kata Containers	OpenStack Foundation community	Linux	containers wrapped in lightweight vms
gVisor	Google	Linux	custom user-space kernel that intercepts the containerised applications' system calls

Table 5: Secure Containers Overview

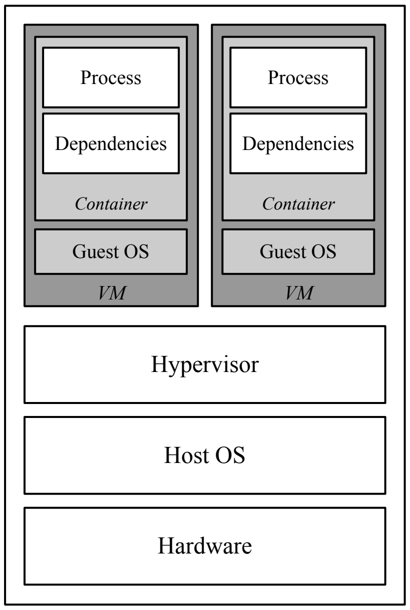

Figure 6: Kata Containers Architecture

workloads running in separate kernels while utilising CPU VT extensions for isolation. The most fundamental way it balances performance with isolation is by using a lightweight VMM by default (QEMU/KVM). Additionally, it employs an optimised hardware passthrough implementation allowing the VMs to communicate efficiently with devices [47].

Kata Containers has been criticised in the literature. More specifically, its assumption that virtual machines are secure enough and one only needs to worry about their performance is questionable [46]. Additionally, in practice, overloaded containers stress `kata-runtime` significantly [61]. Finally, the isolation Kata containers offer in practice has been criticised as weaker than Google gVisor (Section 7.2) [61].

7.2 Google gVisor

Google gvisor is a user-space kernel; it intercepts the containerised applications' system calls in order to isolate them further (essentially providing some OS functionality in user-space) [16] [8] [2] [20]. Since it intercepts application system calls, there is no need for a hypervisor; thus, it is a different approach from the ones mentioned above [16]. Each container is run with its own kernel (`Sentry`), which implements a large amount of the available system calls [16] [8]. Like Kata Containers 7.1, its runtime (`runsc`) is OCI-compliant allowing for easy Docker 6.1 integration [68].

Compared to Docker's runtime engine (`runc`), the `runsc` engine is more isolated: the application inside the container has no direct access to the host kernel but only to Sentry [8]. Consequently, gVisor achieves better security; When a Sentry breach occurs, an attacker can only access a user-space process (whose possible system calls are limited) [68].

The approach of Google gVisor does not come without disadvantages. Applications that use system calls that Sentry has not implemented cannot run on the platform [16]. Having Sentry provide some OS functionality on user-space makes for a more complex design than standard containers while still depending on the same kernel functionality they depend on [8].

Performance-wise, Google gVisor has been shown to lack significantly compared to traditional containers; simple system calls are more than twice as slow [68]. Moreover, I/O on an external tmpfs can take 216 times longer [68].

To conclude, gVisor improves the isolation of containers by significantly decreasing the host system's and the containers' shared resources and by managing the container resources dynamically using the Gofer and Sentry modules [61]. The increased isolation comes with a significant performance overhead [61]. Thus, once more, containerisation comes at a high cost [68].

8 Unikernels - Paravirtualisation

Unikernels are single-purpose programs built by compiling the entire software stack into an image, linking libraries that provide functionality which the OS would traditionally provide. The produced images are single-address space applications that can be run by a hypervisor or directly on hardware (Figure 7) [37]. Unikernels contain only the operating system and application code necessary to run a specific application [36]. Due to their small performance overhead and footprint, Unikernels can be deployed to various domains, from edge microservices to desktop applications [41].

Comparing the 3 Unikernels we mention below (Figure 8), OSv (Section 8.1) gets assigned the best ease of use score without requiring modifying virtual machine images. MirageOS (Section 8.3) requires porting software to the OCaml programming language, making it troublesome to put into

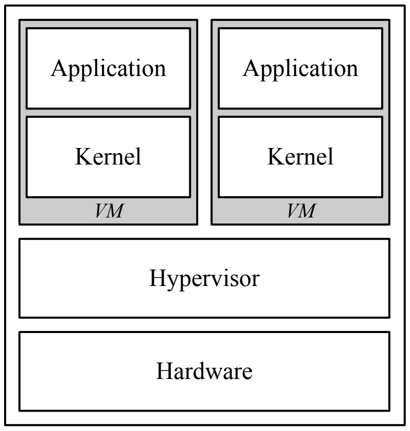

Figure 7: Unikernel Architecture

Tool Name	Developer	Focus
OSv	Cloudius Systems	cloud computing-focused Unikernel
IncludeOS	IncludeOS AS	IoT-focused Unikernel for C++ services
MirageOS	open-source community	type-safe Xen images

Table 6: Unikernels Overview

service.

Unikernel Architecture. Here, we briefly describe the architecture of Unikernels. First, the configuration of the image is part of the compilation, reducing overhead since options are programmable and thus can be type-checked and statically analysed [33]. Second, the architecture allows for system-wide optimisations since linked libraries provide the os functionality; the linker has knowledge of which libraries must be included and can provide further optimisations. Last, Unikernels rely on the hypervisor as the sole means of isolation since they are tailored to specific applications. They utilise protocol libraries to enable applications to trust external entities.

Unikernel Security. To ensure security, due to their design, unikernels allow the following optimisations for security: They use sealing to ensure that the binaries only contain code available during compilation, making them less vulnerable to code injection attacks [33]. They employ address space randomisation at compile time to make it more difficult for attackers to predict the location of critical system components. The interface has fewer Linux system calls, thus reducing the attack surface [62]. Having a very lightweight design itself minimises the attack surface exposed.

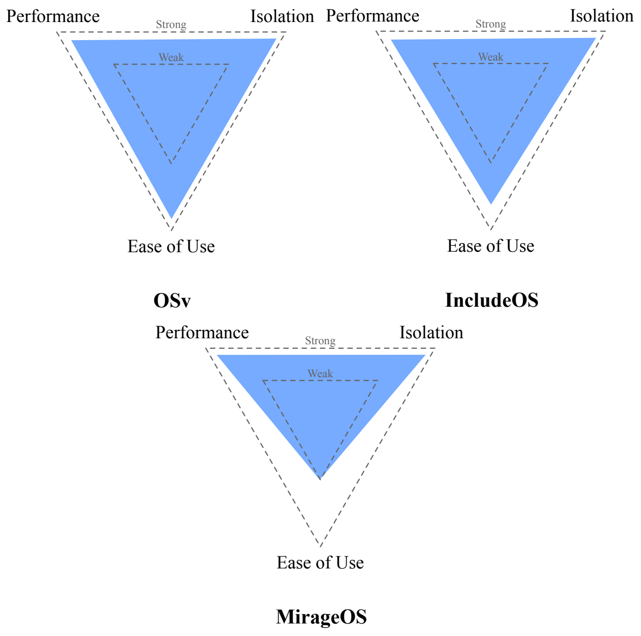

Figure 8: Qualitative comparison of Unikernels

Unikernel Performance. One study found that the Unikernel benefits come at zero to little cost to performance in all cases [33]. A different study showed that Unikernels are an option for replacing containers by placing language runtimes into a hypervisor [34].

Unikernels Critique. The current tooling available for virtual machines is unsuitable for building lightweight and responsive cloud services. It must be adapted for unikernels or lightweight VMs to address memory density, performance, and startup time [36]. Furthermore, They are usually tied to a specific implementation language which can be limiting in many scenarios. Porting an application to a Unikernel model can be unwieldy, contributing to the Unikernels' slow adoption rate [41].

8.1 OSv

OSv is an open-source cloud computing-focused Unikernel intended to run under a hypervisor [40] [42] [26] [65].

Its motivation comes from the observation that features traditionally provided by operating systems, like hardware abstraction, are already provided by cloud hypervisors; avoiding providing the same functionality in an operating system can avoid some functionality overlap and, thus, code duplication [26]. Taking direct influence from the theoretical background of Exokernel (Section 8.4), it takes the role of a library operating system while the hypervisor takes the role of the exokernel [26].

A significant advantage of OSv, compared to other tools like MirageOS (Section 8.3), is its ability to run existing Linux applications [26]. Another advantage is its support of several hypervisors rather than targeting a single one; for example, MirageOS exclusively targets Xen (Section 5.5). Moreover, under specific circumstances, one study found it to perform better than Docker (Section 6.1) and KVM (Section 5.3) [65].

8.2 IncludeOS

IncludeOS is a cloud and IoT-focused Unikernel for C++ services. It advertises a minimal footprint and is independent of virtualisation platform [7].

Similar to most Unikernels, thanks to using statically linked libraries, IncludeOS achieves true minimality: only things needed are included. For example, a hello world IncludeOS application run on QEMU has less than 1/3 of the memory footprint of an equivalent Java program executed in the JVM [7].

IncludeOS achieves isolation and performance mainly from its zero-overhead principle; it compromises on features and focuses on specialisations. This limits its usability. For example, Linux IRQ (interrupt requests) and Programmable interrupt timers are entirely absent [7]. Its non-preemptive kernel reduces the usability of the Unikernel by making it unfit for scenarios with real-time requirements. Another way it approaches security and isolation is by using only one network driver (virtio), which reduces the attack surface it exposes.

8.3 MirageOS

MirageOS produces unikernels by compiling OCaml code into Xen Images [34]. It uses OCaml for the following two reasons [33]:

- It provides concise syntax, which reduces the attack surface.
- Since some Xen components are implemented in OCaml, integration is easier.

One of the ways MirageOS approaches security isolation is by reducing source-level backward compatibility [33]. Additionally, the choice of type-safe OCaml prevents memory overflows in I/O handling [33].

MirageOS has been shown to have little to zero penalties in performance in all cases [33]. While providing type safety and zero overhead, its single-language focus is a significant obstacle to porting legacy code, directly impacting the tools' potential adoption.

8.4 Notable mention : Exokernel

The work of MIT Parallel and Distributed Operating Systems group, Exokernel, has had a significant influence on minimalistic kernel designs [36]. It takes an opinionated approach to kernel design, where the OS abstracts as few resources as possible, forcing application-level resource management. This comes from the observation that the lower the level of a primitive, the more efficiently it can be implemented [13]. Consequently, it provides applications with the ability to manage physical resources directly. [36] More specifically, the kernel exports the hardware resources through an interface, which library operating systems can interact with and create system objects [13].

9 Conclusion

Compared to containers, virtual machines do not share the host kernel [38]; Consequently, VMs are generally considered more isolated and thus more secure [39]. Virtual machines, compared to containers, offer slower boot times [39]. Due to the isolation of VMMs, the overhead when sharing data between the guest and the host (or other guests) is generally noticeable [15] [38], especially when several virtual machines are vying for the same resources [22]. However, the application type can significantly influence the overhead of each hypervisor [22].

While containers rose to prevalence as an alternative to hypervisors, their isolation has been deemed unfit in many scenarios, leading to the common practice of deploying containers inside virtual machines. As documented, surveys have shown that many companies see the security of containers as an obstacle to adoption [57]. At first sight, the solution of blending containers and VMMs promises the best of both worlds: Full isolation and zero performance overhead. Kata Containers (Section 7.1) builds on this principle. This approach has been studied by [8], [38]. The practice above has been criticised: performance impact is noteworthy in both containers and VMs, mainly in the case of I/O operations [15]; combining the above does not necessarily improve performance while adding the overhead related to hypervisors.

To conclude, no perfect isolation mechanism exists for all scenarios; Expert tuning is needed to minimise the main culprit of performance: I/O operations [15]. Moreover, more often than not, one has to take into account the variables of ease of use and configurability of each tool [9]; for example, KVM is notoriously hard to configure [15]. Advanced tools like Firecracker (Section 5.4) and Kata containers (Section 7.1) are opinionated and make compromises to favour specific use cases. QEMU (Section 5.2), due to its supported hardware architectures and available virtualised devices, is one of the most feature-complete isolation mechanism available but suffers from a large attack surface. In the case of isolation mechanisms optimisation, *there is no such thing as a free lunch* [63].

9.1 Future Direction

WebAssembly is a bytecode format initially designed to support near-native speed in applications running on web pages. A virtual stack machine reads and executes the instructions in a memory-safe environment. Code from different, even unsafe, programming languages such as C++ and Rust can be compiled to WebAssembly and run in its sandboxed environment. WebAssembly has gotten traction in the context of isolation mechanisms; for example, Docker has started offering beta support for it [11]. Moreover, it has been adapted for usage on the Edge [17].

WebAssembly has been criticised extensively, most notably by Lehmann et al., [29] for bringing vulnerabilities considered solved by natively compiled programs to the foreground. For example, buffer overflow prevention mechanisms like stack canaries are not utilised.

In their paper *Will Serverless End the Dominance of Linux in the Cloud?* [28], Koller et al. make the observation that the Linux kernel is making every effort to stay on top of isolation techniques leading to an excessively complex database; its abstractions are currently ill-suited for the serverless paradigm, where the execution unit is not a traditional process. They raise the argument for reconsidering the dominance of Linux in the cloud and considering replacing it with Unikernels (Section 8), which can offer OS functionality through library operating systems.

References

[1] Aalam, Z., Kumar, V., and Gour, S. A review paper on hypervisor and virtual machine security. In *Journal of Physics: Conference Series* (2021), vol. 1950, IOP Publishing, p. 012027.

[2] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. Firecracker: Lightweight virtualization for serverless applications. In *NSDI* (2020), vol. 20, pp. 419–434.

[3] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. *ACM SIGOPS operating systems review 37*, 5 (2003), 164–177.

[4] Bellard, F. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track* (2005), vol. 41, California, USA, p. 46.

[5] Bernstein, D. Containers and cloud: From lxc to docker to kubernetes. *IEEE cloud computing 1*, 3 (2014), 81–84.

[6] Bhardwaj, A., and Krishna, C. R. Virtualization in cloud computing: Moving from hypervisor to containerization—a survey. *Arabian Journal for Science and Engineering 46*, 9 (2021), 8585–8601.

[7] Bratterud, A., Walla, A.-A., Haugerud, H., Engelstad, P. E., and Begnum, K. Includeos: A minimal, resource efficient unikernel for cloud services. In *2015 IEEE 7th international conference on cloud computing technology and science (cloudcom)* (2015), IEEE, pp. 250–257.

[8] Caraza-Harter, T., and Swift, M. M. Blending containers and virtual machines: a study of firecracker and gvisor. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2020), pp. 101–113.

[9] Che, J., Shi, C., Yu, Y., and Lin, W. A synthetical performance evaluation of openvz, xen and kvm. In *2010 IEEE Asia-Pacific Services Computing Conference* (2010), IEEE, pp. 587–594.

[10] Combe, T., Martin, A., and Di Pietro, R. To docker or not to docker: A security perspective. *IEEE Cloud Computing 3*, 5 (2016), 54–62.

[11] Docker. Docker+wasm (beta). https://docs.docker.com/desktop/wasm/.

[12] Emeneker, W., and Stanzione, D. Hpc cluster readiness of xen and user mode linux. In *2006 IEEE International Conference on Cluster Computing* (2006), IEEE, pp. 1–8.

[13] Engler, D. R., Kaashoek, M. F., and O'Toole Jr, J. Exokernel: An operating system architecture for application-level resource management. *ACM SIGOPS Operating Systems Review 29*, 5 (1995), 251–266.

[14] Fayyad-Kazan, H., Perneel, L., and Timmerman, M. Full and para-virtualization with xen: a performance comparison. *Journal of Emerging Trends in Computing and Information Sciences 4*, 9 (2013), 719–727.

[15] Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)* (2015), IEEE, pp. 171–172.

[16] Flauzac, O., Mauhourat, F., and Nolot, F. A review of native container security for running applications. *Procedia Computer Science 175* (2020), 157–164.

[17] Gadepalli, P. K., McBride, S., Peach, G., Cherkasova, L., and Parmer, G. Sledge: a serverless-first, light-weight wasm runtime for the edge. In *Proceedings of the 21st International Middleware Conference* (2020), pp. 265–279.

[18] Garfinkel, T., Rosenblum, M., et al. A virtual machine introspection based architecture for intrusion detection. In *Ndss* (2003), vol. 3, San Diega, CA, pp. 191–206.

[19] Goldberg, R. P. Architectural principles for virtual computer systems. Tech. rep., HARVARD UNIV CAMBRIDGE MA DIV OF ENGINEERING AND APPLIED PHYSICS, 1973.

[20] Google. What is gvisor? https://gvisor.dev/docs/.

[21] Gupta, D., Cherkasova, L., Gardner, R., and Vahdat, A. Enforcing performance isolation across virtual machines in xen. In *Middleware 2006: ACM/IFIP/USENIX 7th International Middleware Conference, Melbourne, Australia, November 27-December 1, 2006. Proceedings 7* (2006), Springer, pp. 342–362.

[22] Hwang, J., Zeng, S., y Wu, F., and Wood, T. A component-based performance comparison of four hypervisors. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)* (2013), IEEE, pp. 269–276.

[23] Hypervisor, C. Documentation. https://www.cloudhypervisor.org/docs/prologue/introduction/.

[24] Intel. Kata containers. https://www.intel.com/content/www/us/en/developer/articles/technical/kata-containers.html.

[25] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium* (2007), vol. 1, Dttawa, Dntorio, Canada, pp. 225–230.

[26] Kivity, A., Laor, D., Costa, G., Enberg, P., Har'El, N., Marti, D., and Zolotarov, V. Osv—optimizing the operating system for virtual machines. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)* (2014), pp. 61–72.

[27] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM 63*, 7 (2020), 93–101.

[28] Koller, R., and Williams, D. Will serverless end the dominance of linux in the cloud? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (2017), pp. 169–173.

[29] Lehmann, D., Kinder, J., and Pradel, M. Everything old is new again: Binary security of webassembly. In *Proceedings of the 29th USENIX Conference on Security Symposium* (2020), pp. 217–234.

[30] Li, Z., Guo, L., Cheng, J., Chen, Q., He, B., and Guo, M. The serverless computing survey: A technical primer for design architecture. *ACM Computing Surveys (CSUR) 54*, 10s (2022), 1–34.

[31] Li, Z., Kihl, M., Lu, Q., and Andersson, J. A. Performance overhead comparison between hypervisor and container based virtualization. In *2017 IEEE 31st International Conference on advanced information networking and applications (AINA)* (2017), IEEE, pp. 955–962.

[32] LXC. Linux containers security. https://linuxcontainers.org/lxc/security/.

[33] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Computer Architecture News 41*, 1 (2013), 461–472.

[34] Madhavapeddy, A., and Scott, D. J. Unikernels: the rise of the virtual library operating system. *Communications of the ACM 57*, 1 (2014), 61–69.

[35] Mampage, A., Karunasekera, S., and Buyya, R. A holistic view on resource management in serverless computing environments: Taxonomy and future directions. *ACM Computing Surveys (CSUR) 54*, 11s (2022), 1–36.

[36] Manco, F., Lupu, C., Schmidt, F., Mendes, J., Kuenzer, S., Sati, S., Yasukata, K., Raiciu, C., and Huici, F. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 218–233.

[37] Martin, A., Raponi, S., Combe, T., and Di Pietro, R. Docker ecosystem–vulnerability analysis. *Computer Communications 122* (2018), 30–43.

[38] Mavridis, I., and Karatza, H. Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing. *Future Generation Computer Systems 94* (2019), 674–696.

[39] Mieden, P., and Partarrieu, P. Performance analysis of kvm-based microvms orchestrated by firecracker and qemu. Tech. rep., Technical Report. University of Amsterdam, 2019.

[40] Morabito, R., Kjällman, J., and Komu, M. Hypervisors vs. lightweight virtualization: a performance comparison. In *2015 IEEE International Conference on cloud engineering* (2015), IEEE, pp. 386–393.

[41] Olivier, P., Chiba, D., Lankes, S., Min, C., and Ravindran, B. A binary-compatible unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2019), pp. 59–73.

[42] OSv. Design. http://osv.io/design.

[43] Patil, R., and Modi, C. An exhaustive survey on security concerns and solutions at different components of virtualization. *ACM Computing Surveys (CSUR) 52*, 1 (2019), 1–38.

[44] Pearce, M., Zeadally, S., and Hunt, R. Virtualization: Issues, security threats, and solutions. *ACM Computing Surveys (CSUR) 45*, 2 (2013), 1–39.

[45] Perez-Botero, D., Szefer, J., and Lee, R. B. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Proceedings of the 2013 international workshop on Security in cloud computing* (2013), pp. 3–10.

[46] Randal, A. The ideal versus the real: Revisiting the history of virtual machines and containers. *ACM Computing Surveys (CSUR) 53*, 1 (2020), 1–31.

[47] Randazzo, A., and Tinnirello, I. Kata containers: An emerging architecture for enabling mec services in fast and secure way. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)* (2019), IEEE, pp. 209–214.

[48] Rathore, M. S., Hidell, M., and Sjödin, P. Kvm vs. lxc: comparing performance and isolation of hardware-assisted virtual routers. *American Journal of Networks and Communications 2*, 4 (2013), 88–96.

[49] RedHat. What is kvm? https://www.redhat.com/en/topics/virtualization/what-is-KVM.

[50] Russell, R. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review 42*, 5 (2008), 95–103.

[51] SELinux. svirt overview. https://selinuxproject.org/page/SVirt.

[52] Shen, Z., Sun, Z., Sela, G.-E., Bagdasaryan, E., Delimitrou, C., Van Renesse, R., and Weatherspoon, H. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (2019), pp. 121–135.

[53] Shillaker, S., and Pietzuch, P. Faasm: Lightweight isolation for efficient stateful serverless computing. *arXiv preprint arXiv:2002.09344* (2020).

[54] Shu, R., Wang, P., Gorski III, S. A., Andow, B., Nadkarni, A., Deshotels, L., Gionta, J., Enck, W., and Gu, X. A study of security isolation techniques. *ACM Computing Surveys (CSUR) 49*, 3 (2016), 1–37.

[55] Sierra-Arriaga, F., Branco, R., and Lee, B. Security issues and challenges for virtualization technologies. *ACM Computing Surveys (CSUR) 53*, 2 (2020), 1–37.

[56] Smith, J. E., and Nair, R. The architecture of virtual machines. *Computer 38*, 5 (2005), 32–38.

[57] Sultan, S., Ahmad, I., and Dimitriou, T. Container security: Issues, challenges, and the road ahead. *IEEE access 7* (2019), 52976–52996.

[58] Szefer, J., Keller, E., Lee, R. B., and Rexford, J. 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.

[59] van Rijn, V., and Rellermeyer, J. S. A fresh look at the architecture and performance of contemporary isolation platforms. In *Proceedings of the 22nd International Middleware Conference* (2021), pp. 323–335.

[60] Volpert, S., Erb, B., Eisenhart, G., Seybold, D., Wesner, S., and Domaschka, J. A methodology and framework to determine the isolation capabilities of virtualisation technologies. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering* (2023), pp. 149–160.

[61] Wang, X., Du, J., and Liu, H. Performance and isolation analysis of runc, gvisor and kata containers runtimes. *Cluster Computing 25*, 2 (2022), 1497–1513.

[62] Williams, D., Koller, R., Lucina, M., and Prakash, N. Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing* (2018), pp. 199–211.

[63] Wolpert, M. No free lunch theorem. https://en.wikipedia.org/wiki/No_free_lunch_theorem.

[64] Wright, C., Cowan, C., Smalley, S., Morris, J., and Kroah-Hartman, G. Linux security modules: General security support for the linux kernel. In *11th USENIX Security Symposium (USENIX Security 02)* (2002).

[65] Xavier, B., Ferreto, T., and Jersak, L. Time provisioning evaluation of kvm, docker and unikernels in a cloud platform. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)* (2016), IEEE, pp. 277–280.

[66] Xavier, M. G., Neves, M. V., Rossi, F. D., Ferreto, T. C., Lange, T., and De Rose, C. A. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (2013), IEEE, pp. 233–240.

[67] Xen. Xen project software overview. `https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview`.

[68] Young, E. G., Zhu, P., Caraza-Harter, T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. The true cost of containing: A gvisor case study. In *HotCloud* (2019).