THE UNIVERSITY
of EDINBURGH

Dependable Virtualised Systems

Jörg THALHEIM

Doctor of Philosophy

Adviser: Prof. Pramod BHATOTIA

Thesis committee chair: Prof. Björn FRANKE

Reviewers: Prof. Eduard BUGNION, Prof. Boris GROT

Institute of Computing Systems Architecture

School of Informatics

The University of Edinburgh

2022

Abstract

Virtual machines and containers are widely used in data centres and in the cloud for software deployment and management. Their popularity is based on higher capacity utilisation, lower maintenance costs, and better scalability by creating an abstraction layer on top of physical hardware. The economics and scalability of virtualised applications require that the workloads of multiple customers can run on the same hardware with low overhead without compromising security. To address this need, in this work we introduce a new set of IO middleware that allows users to run smaller containers and virtual machines and deploy them in a more secure manner.

The presented contributions can be summarised as follows:

- CNTR provides a way to extend application containers at runtime with tools deployed in a different container. In this way, you can create "slim" images that contain only the actual application, while all the tools needed for monitoring, testing, and debugging reside in a "fat" image that only needs to be deployed when needed. CNTR achieves this by creating a nested namespace in the application container that proxies files from a remote container using a FUSE filesystem.

- VMSH allows users to attach services to running virtual machines independent of the guest userspace and without any pre-installed agents. Similar to CNTR this allows developers to build more light-weight virtual machines by deploying additional services in a separate user-provided file system image on-demand. VMSH achieves this by side-loading kernel code into the guest and mounting a filesystem based on its own block device in a light-weight container without affecting the applications in the VM.

- RKT-IO leverages trusted execution environments to run workloads in containers and virtual machines to protect them from other tenants and the cloud provider on the same host, but without sacrificing on I/O performance that is usually degraded by this protection. It does so by providing a userspace network and storage I/O stack in the form of a library OS based on Linux that directly accesses the hardware from within the TEE by-passing the host kernel.

Lay summary

This thesis is about making virtualisation technologies in computing more maintainable and secure. Virtualisation logically divides physical computers into smaller virtual compartments. For computing there exists two type of compartments: containers or virtual machines. Containers and virtual machines can run users programs and operating systems safely and securely isolated from other users on the same computer. Their popularity is based on higher capacity utilisation, lower maintenance costs, and better scalability by creating an abstraction layer on top of physical hardware. The economics and scalability of virtualised applications require that the workloads of multiple customers can run on the same hardware with low overhead without compromising security. To address this need, in this work we introduce three new systems that allows users to run smaller containers and virtual machines and deploy them in a more secure manner.

The presented contributions can be summarised as follows:

CNTR provides a way to reduce the size of container images. Containers run applications in an environment that is separate from the rest of the operating system. Container images include all the files available to the applications running in the container. To allow developers and administrators to manage the applications in these environments, containers also have additional tools installed. With CNTR, users can run tools within the application container that are stored in another container. This additional container only needs to be deployed when required, keeping the original application container lean.

VMSH provides similar functionality to CNTR, but for virtual machines instead of containers. Virtual machines allow multiple operating systems to run on the same physical computer by emulating virtual hardware with a program called a hypervisor. With VMSH, virtual machines can be expanded at runtime with new programs. This allows virtual machines to be kept lightweight by only having files available to the application, while additional tools are only deployed when needed.VMSH achieves this extension in a universal way without relying on specific programs in the virtual machine, and is portable across hypervisors.

RKT-IO uses modern processor features to run applications in a trusted execution environment (called TEE) in such a way that no other applications running on the same computer can read or modify their data. This makes it possible to establish trust in hardware that has been rented from a third party (cloud computing). While previous approaches could establish similar trust, RKT-IO greatly improves performance when using network and storage hardware by accessing the hardware directly rather than relying on the operating system.

Publications

This thesis is based on the following conference papers.

1. CNTR : *Lightweight OS Containers* by Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci, In the proceedings of *USENIX ATC 2018* [251]

2. VMSH : *Hypervisor-agnostic Guest Overlays for VMs* by Jörg Thalheim, Peter Okelmann, Harshavardhan Unnibhavi, Redha Gouicem, Pramod Bhatotia, In the proceedings of *ACM EuroSys 2022* [129]

3. RKT-IO : *A Direct I/O Stack for Shielded Execution* by Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter Pietzuch, In the proceedings of *ACM EuroSys 2021* [252]

Other conference papers during my PhD

4. *Sieve : Actionable insights from monitored metrics in distributed systems* by Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer, In the proceedings of *Middleware 2017*:
`https://dl.acm.org/doi/10.1145/3135974.3135977`

5. *Speicher : Securing LSM-based Key-Value Stores using Shielded Execution* by Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani, In the proceedings of *Usenix FAST 2018*:
`https://www.usenix.org/conference/fast19/presentation/bailleu`

Table of Contents

List of Figures

Signed declaration

I declare that the thesis has been composed by myself and that the work has not been submitted for any other degree or professional qualification. I confirm that the work submitted is my own, except where work which has formed part of jointly-authored publications has been included. My contribution and those of the other authors to this work have been explicitly indicated below. I confirm that appropriate credit has been given within this thesis where reference has been made to the work of others. The work presented in § 3 was previously published in USENIX ATC 2018 as "CNTR: Lightweight OS Containers" by Jörg Thalheim, Pramod Bhatotia (phd supervisor), Pedro Fonseca, Baris Kasikci. § 4 was submitted to ACM EuroSys 2022 with the title "VMSH: Hypervisor-agnostic Guest Overlays for VMs" by the authors Jörg Thalheim, Peter Okelmann, Harshavardhan Unnibhavi, Redha Gouicem and Pramod Bhatotia. Lastly § 5 was published in under the title "RKT-IO: A Direct I/O Stack for Shielded Execution" in ACM EuroSys 2021 by Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia (phd supervisor) and Peter Pietzuch.

Jörg Thalheim

Chapter 1

Introduction

This thesis is about making virtualisation technologies in computing more reliable, maintainable and secure. Virtualisation logically divides physical resources into smaller virtual compartments. Virtualisation is a critical component in all areas of modern data centres and clouds such as storage (object stores [178], distributed block storage [7]), networking (SDN [38], SD-WAN [18]) and computing (containers, virtual machines). Virtualisation improves *efficiency* and *utilisation* by allowing service providers to overprovision physical resources by sharing them with multiple clients. Because virtualisation abstracts away from different types and generations of hardware, it allows providers to develop unified APIs for provisioning and scaling resources, which is the foundation of what we know as cloud computing. Users of cloud infrastructure can scale resources both *vertically* (i.e., by increasing the size of a virtual block device or selecting a larger compute instance) and *horizontally* (by creating more instances of the same type) as needed. This provides users and infrastructure providers with more *flexibility*. Users do not have to order hardware on-premise and thus do not need to commit financial capital, while infrastructure providers can migrate users between different physical hardware without degrading *availability* and upgrade hardware generations without breaking *compatibility*.

1.1 Trends

Virtualisation of computing resources is moving away from developers managing virtual machines and containers to providers automatically scaling applications on-demand and closer to the end-user [12, 195, 107, 117, 98, 29].

In this model developers provide immutable application images and the infrastructure automatically and reproducibly scales application instances on demand (i.e., based on the request rate in the load balancer) [66]. Instead of a

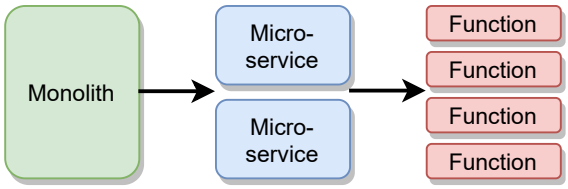


Figure 1.1: Trends in Service Architectures: In recent years we have seen a transition from Monolithic and Microservice architectures to Serverless architectures (also known as Function-as-a-Service). In this model, the developer provides application images for functions and the provider handles the provisioning of those functions.

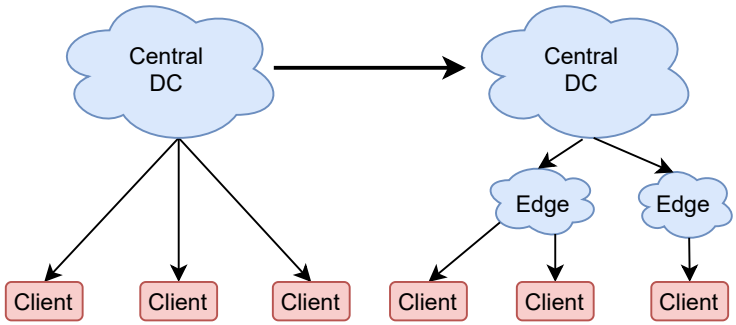


Figure 1.2: Shift to new data centre architectures: To reduce network latencies between the application and the end customer (i.e. for real-time data processing), providers are now offering edge cloud data centres that are closer to their customers.

monolithic or microservice architecture, parts or all of the functionality are shifted to serverless architectures (see also Figure 1.1). Since unused instances are shut down and the customer is only charged for the CPU seconds (instead of minutes) that the application was online, this leads to better utilisation of the infrastructure, less usage cost for the customer [27] and lower operational cost for managing the infrastructure [233].

As providers gain more information about application metrics and more control over the application life cycle, they can overprovision more customers on the same hardware and redeploy them to new machines [2] faster. Moreover, the latency caused by the distance between the data centre and the base station is becoming the new bottleneck due to new low-latency mobile networks [247] (see Figure 1.2). Therefore, to enable new applications that can benefit from these latency gains, telecom and CDN providers are offering computing resources in smaller data centres that are closer to the customer, i.e., near 5G base stations or in CDN edge nodes [55]. Again it is important that the provider can automatically provision the application depending on the proximity to the customer.

1.2 Challenges

These new environments bring new challenges for providers, developers and administrators. Application images must be kept small so that they can be distributed across the network and booted quickly[105, 189]. For this reason, management, debugging and administration services and tools are no longer included in these images. While these tools are not required for the application to run (since the provider handles deployment), in the event of errors developers and administrators still need a way to efficiently determine the root cause. Ideally developers and administrators would have the ability to attach development and management tools to different application contexts running in different virtualisation environments and debug interactively via a remote session without having to redeploy the running applications. This is becoming increasingly important as multiple layers of virtualisation become more common and it becomes difficult to track down configuration errors due to increasing complexity (e.g., containers within containers running in virtual machines connected via virtual overlay networks).

At the same time, virtualisation increases the risk of security breaches when applications run in untrusted third-party cloud environments shared with other tenants. Attackers (or even malicious cloud administrators) can compromise the application's security. Emerging edge data centres also tend to have fewer physical security measures (dedicated security personnel, multiple access-gated security doors), making them more vulnerable than traditional data centres[40]. In fact, many studies show that software bugs, configuration errors, and security vulnerabilities pose a serious threat to cloud systems, and software security is cited as a barrier to the adoption of cloud solutions [229].

1.3 Contributions

To address these challenges we are developing a new set of IO abstractions to improve the dependability (reliability, maintainability and security) of virtualised environments.

To improve *reliability* and *maintainability*, we enable developers and administrators to inspect, debug, and monitor their applications deployed in containers (CNTR in Chapter 3) or virtual machines (VMSH in Chapter 4) using tools and services from special optimised containers or filesystem images by bridging the isolation boundaries imposed by virtualisation. In terms of *security*, we provide transparent encryption and confidentiality of computations executed in containers or virtual machines by using a trusted execution environment (RKT-IO in Chapter 5).

Specifically, in this thesis we will present the design and implementation of the following three systems:

CNTR provides a way to extend application containers at runtime with tools that are deployed in another container. The observation here is that containers often contain a wide range of tools for developers and administrators that are not required for the application but are occasionally used for debugging. With CNTR, users can create "slim" images that contain only the actual application, while all the tools needed for monitoring, testing and debugging are placed in a "fat" image that only needs to be deployed when needed. CNTR achieves this by creating a nested namespace in the application container that forwards files from a remote container via a FUSE filesystem. In our extensive evaluation we show that CNTR introduces a reasonable performance overhead while reducing the image size of all official top 50 images on Docker Hub by 66.6%.

VMSH allows services to be attached to running virtual machines independently of the guest userspace and without any pre-installed agents. In other words, VMSH provides out-of-band management for virtual machines, similar to Baseboard Management Controller (BMC) for physical hardware [124, 220, 16]. Similar to CNTR, it allows developers to create lightweight virtual machines by deploying additional services on demand in a separate user-supplied filesystem image. VMSH accomplishes this by side-loading kernel code into the guest and mounting a filesystem based on its own block device in a lightweight container without affecting the applications in the VM. VMSH targets KVM directly and is therefore not tied to any particular Type-2 hypervisor. In our evaluation, we show that VMSH does not add any overhead to the application in the VM and demonstrate its usefulness through a number of implemented use cases.

RKT-IO uses trusted execution environments (i.e., SGX) to run workloads in containers and virtual machines to protect them from other tenants and the cloud provider on the same host, but without sacrificing I/O performance that is normally degraded by this protection. This is done by providing a userspace network and storage I/O stack in the form of a Linux-based library OS. RKT-IO accesses the hardware directly accesses the hardware within the TEE bypassing the host kernel by using the DPDK and SPDK frameworks. Rather than relying on interrupts, which would involve costly context switches, RKT-IO directly polls for IO events and avoids data copies by mapping device memory into userspace. To ensure data confidentiality, all I/O leaving the TEE is encrypted. The evaluation shows that this

approach is 9× faster for network operations and 7× faster for storage access when compared to other frameworks (SCONE, SGX-LKL).

When designing these systems, we focused on practical integration with existing systems without requiring developers to adapt to new APIs. The resulting code is publicly available including steps to reproduce the evaluation. By providing fully functional research prototypes we hope to have a long-term impact on the community and influence similar systems and standards, even outside of academia.

1.4 Outline

The rest of this thesis is divided into five parts. In chapter 2, we provide background knowledge for this dissertation. Then we introduce CNTR in Chapter 3, followed by VMSH in Chapter 4 and RKT-IO in Chapter 5 and finally a conclusion in Chapter 6.

Chapter 2

Background

In our work, we target containers and virtual machines, which we will outline in this chapter along with other relevant used technologies.

2.1 Containers

In CNTR, we provide a new approach to build slim container images. This section provides the technical background on containers that is required for Chapter 3. Containers (see Figure 2.1) are a form of process-level virtualisation enforced in the operating system, and are an important building block in the deployment of modern applications. In fact, all major cloud computing providers (e.g., Amazon [13], Google [99] and Microsoft [28]) offer Containers as a Service (CaaS) and some organisations exclusively deploy their services using containers [101]

Crucially, the kernel allows system resources to be partitioned for a particular process with very little performance overhead. The children of this process will inherit these isolation properties. Container-based virtualisation often relies on three key components: (1) the OS mechanism that enforces process-level isolation (e.g., the Linux `cgroups` (see 2.1.2) and `namespaces` (see 2.1.1)), (2) the application packaging system and runtime (2.1.3), and (3) the orchestration manager that deploys, distributes and manages containers across machines (e.g., Docker Swarm [72], Kubernetes [143]). Together, these components enable users to quickly deploy services across machines, with strong performance and security isolation guarantees, and with low-overheads.

2.1.1 Namespaces

Namespaces [163] are a feature of the Linux kernel that gives processes a separate view of what resources are available and accessible in a system. To this point there are eight

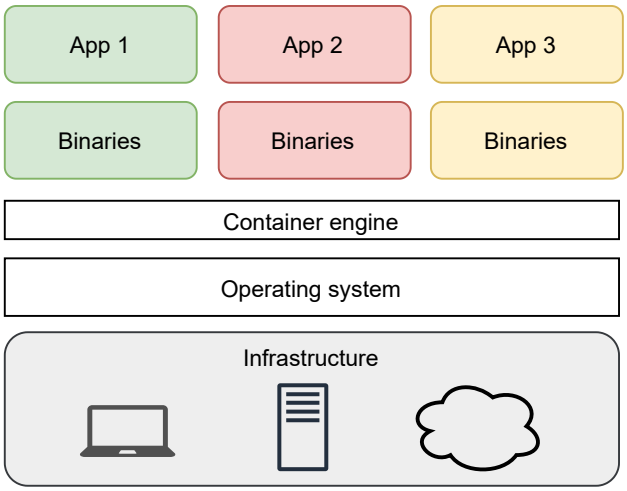


Figure 2.1: Container virtualisation allows applications to run in separate domain and filesystem. The isolation is implemented in the operating system, which is then set up by a container engine in userspace.

different types of namespaces for different types of resources (e.g., mount namespaces for filesystem mountpoints, network namespaces for network interfaces and routes or PID namespaces for process IDs). In particular, mount namespaces are used in containers to restrict access to files and mounts. They allow users to build a new filesystem hierarchy that is different from that of the host. In Chapter 3, we describe how CNTR uses a nested mount namespace to extend a "slim" container with a "fat" container at runtime. VMSH (Chapter 4) also uses mount namespaces to mount its own filesystem in a new chroot that is hidden from the guest so as not to interfere with the guest userland. In addition, both CNTR and VMSH also apply the other namespace types (i.e., network, pid or user namespaces) of the target process when they start their own processes so that they can attach to containers.

2.1.2 Cgroups

Control groups [162] (abbreviated as cgroups) allow for resource constraints , e.g., CPU time, memory usage, or I/O bandwidth limits. Most importantly, these constraints apply to a group of processes rather than a process. The latter feature is also important for process tracking: a pid cgroup allows supervisors track to all processes belonging to a container when implementing a container runtime (2.1.3). Both VMSH and CNTR recognise the cgroups used by a process and register their own process when they attach themselves.

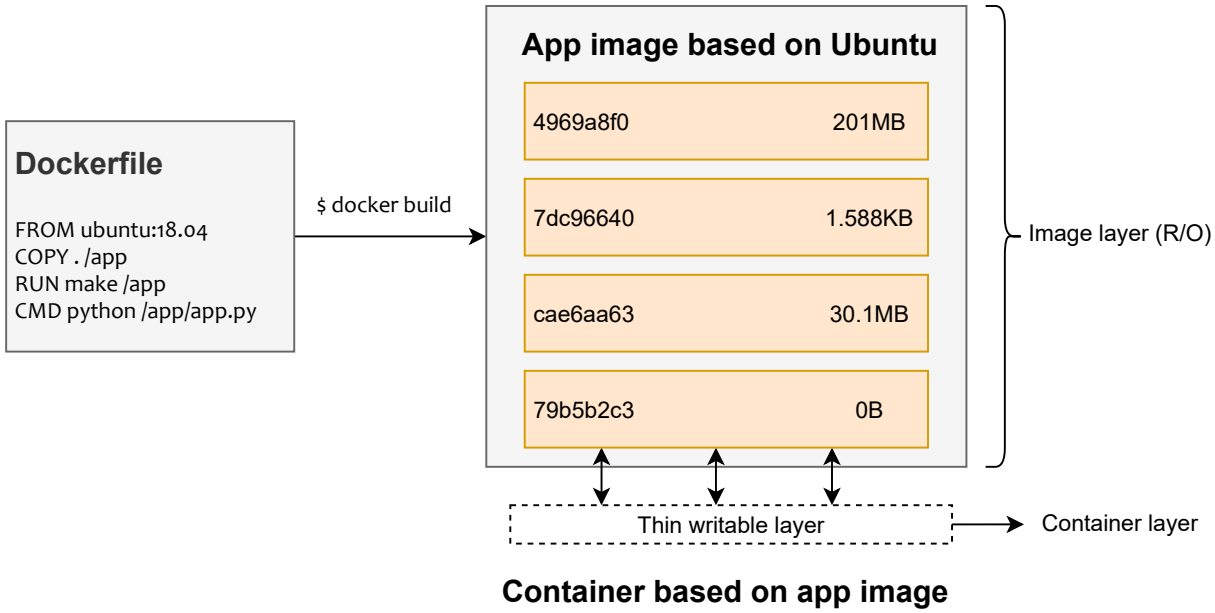


Figure 2.2: Docker builds layered filesystem images based on a Dockerfile. Each line in the Dockerfile becomes a new layer. When a container is started, Docker merges the immutable image layers and mounts a new writable layer on top.

2.1.3 Container runtimes

Users typically do not interact directly with namespaces or cgroups but instead use a userspace container runtime. The runtime sets up the container environment (i.e., storage, network, namespaces, and cgroups). Applications in different containers are isolated and have all their resources included in their own filesystem tree. Popular container runtimes include Docker [69], systemd-nspawn [173] and LXC [271]. Depending on the implementation, container runtimes also take care of packaging applications and services, uploading them to a central registry and then deploying them to the target system. To share storage space between different applications, some container engines also use overlay- or snapshot-capable filesystems (see Figure 2.2) to share common files and base images between different containers on the same host. We discuss this approach in more detail in § 3.2.2 and evaluate its effectiveness in § 3.5.3.

2.1.4 Orchestration manager

Another important factor for the popularity of containers is their orchestration managers such as Kubernetes [143] or Nomad [190]. Orchestration managers distribute containers to different hosts according to their resource needs. They provide overlay networks and load balancers to automate the scaling of applications

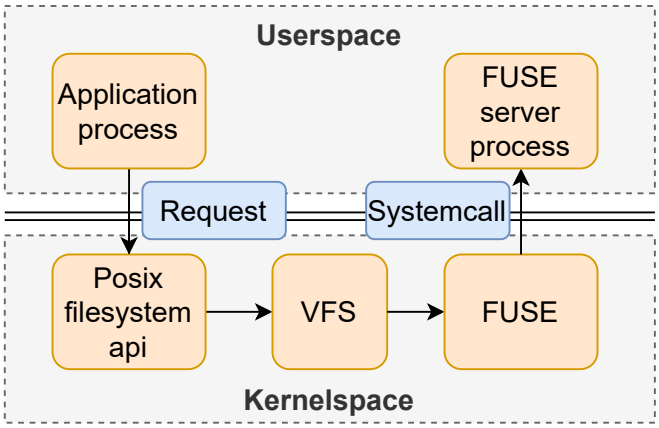


Figure 2.3: FUSE request flow: When an application access a FUSE filesystem, the kernel will forward the request to the responsible FUSE filesystem server running in userspace

in a distributed environment. They can also ensure that workload is scheduled on other nodes when hosts are unavailable.

2.2 Fuse

In addition to namespaces, CNTR (Chapter 3) also implements a FUSE filesystem. FUSE[90] is a cross-platform protocol for userspace filesystems. FUSE consists of a generic FUSE kernel driver that acts as a client and a user-level FUSE server that implements the filesystem. The kernel driver is a proxy between processes accessing the filesystem via Linux VFS and the FUSE server running in userspace (see Figure 2.3). The FUSE server communicates with kernel space using a request/response protocol. CNTR uses FUSE to implement a filesystem proxy that can make files from a "fat container" available in a "slim container".

2.3 Hardware virtualisation

In contrast to CNTR, VMSH aims at hardware-assisted virtualisation instead of process-level virtualisation. In this model, a hypervisor (see Figure 2.4) sets up a virtual machine (VM) and emulates devices for an operating system running inside the VM, called a guest. Hypervisors can run directly on the physical hardware (also called Type 1 hypervisors) or on top of an operating system (Type 2). Hardware-assisted virtualisation leverages the capabilities of host processors to enable efficient full virtualisation. In full virtualisation, the entire hardware environment is emulated to enable the execution of an unmodified guest OS that uses

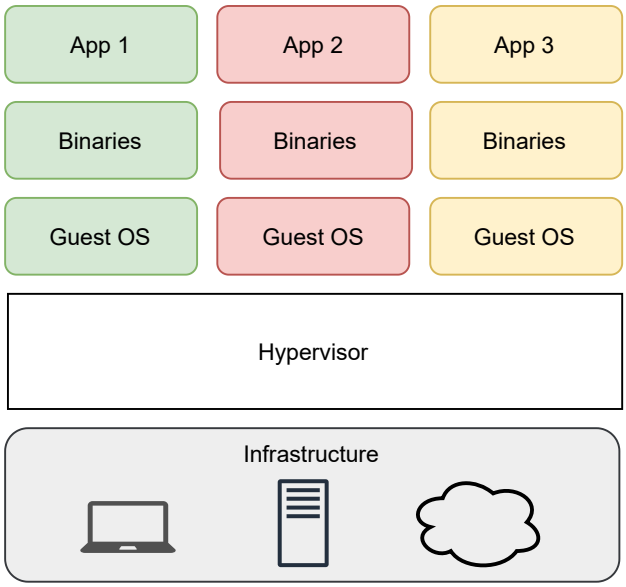


Figure 2.4: Hypervisors separate applications by emulating hardware for each virtual machine. Each virtual machine has its own guest operating system.

the same instruction set as the host machine [33]. Even though container virtualisation imposes less overhead than virtual machines [235], most cloud providers still prefer virtual machines for security reasons [2].

The reason is that the trusted computing base (the part of the system that must be trusted to be correct) of a hypervisor is smaller in terms of code size and easier to verify. For virtual machines, the interface is mainly devices, whereas for container virtualisation the more complex syscall API of the operating system and connected subsystems must be trusted. In summary, containers are useful for lightweight performance isolation and co-locating applications with incompatible dependencies in the same OS, but providers still use virtual machines to securely separate workloads from different tenants on the same physical hardware.

In the following sections, we provide background on our targeted hypervisor interface KVM and the virtual device emulation standard VirtIO both of which we use for the VMSH project in Chapter 4.

2.3.1 Kernel-based virtual machine (KVM)

Kernel-based Virtual Machine (KVM [136]) is a kernel API for Linux (also FreeBSD and Illumos) that provides an abstraction layer on top of the hardware-based virtualisation capabilities of various CPU architectures. KVM belongs to the Type-1 hypervisor class (since it runs on bare-metal hardware) and is a widely used virtualisation API used by major cloud providers [239, 97, 115, 196].

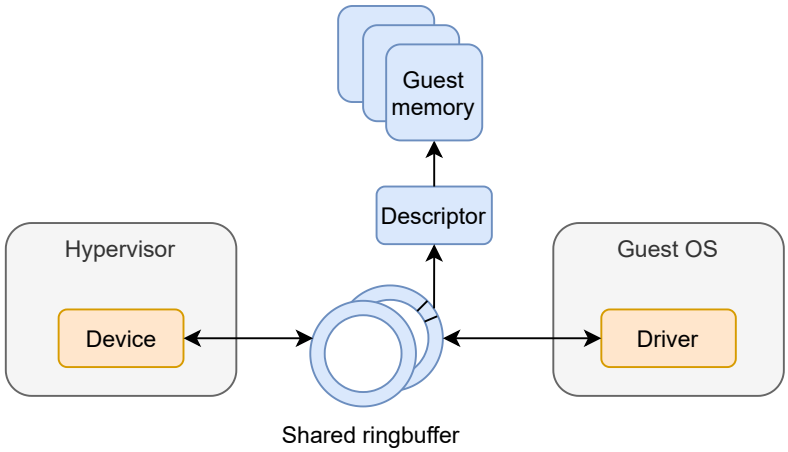


Figure 2.5: Virtio devices are implemented in the hypervisor and are advertised to the guest OS through discovery mechanism such as PCI or MMIO. The guest OS driver can communicate with the hypervisor devices by writing/reading request and response messages in a shared ring buffer queue (also called virtqueues). The shared ring buffer contains descriptors that reference data in the guest's memory.

KVM requires a Type-2 hypervisor running in userspace to run the guest OS using the KVM API. KVM hypervisors include QEMU [212], Firecracker [2], Cloud Hypervisor [54] and crosvm [95]. The hypervisor sets up the initial CPU and memory and emulates the devices (*e.g.*, block, console, NICs). VMSH attaches to VMs based on KVM.

2.3.2 VirtIO

Emulating physical hardware is slow and causes significant overheads compared to the native execution on real hardware.

Therefore, most hypervisors rely on paravirtualisation for devices where the software interface is similar but not identical to its hardware equivalent to improve performance and simplify the interface between the hypervisor and the guest. A widely used standard for paravirtualisation is called VirtIO.

VirtIO defines a common interface for VM-optimised device emulation (network devices, block devices, etc.) [261, 225]. Most hypervisors implement VirtIO devices and their guest drivers exist for all major OSes. Depending on the device type, VirtIO specifies a set of consumer/producer virtqueues in shared memory (see Figure 2.5). Virtqueues are used by the device in the hypervisor and the driver in the guest to exchange data. VirtIO has two main transport mechanisms based on either memory mapped IO (MMIO) or on the PCI standard. In VMSH, we implement the MMIO

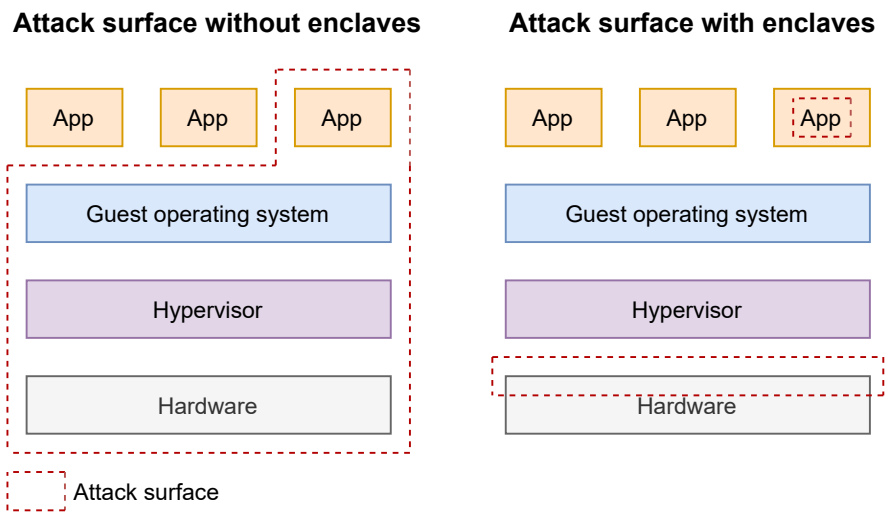


Figure 2.6: SGX enclaves can significantly reduce the attack surface of applications running inside virtual machines. The cloud provider, operating system and hypervisor no longer need to be trusted.

variant, which is widely used, especially in microVMs [214, 2, 84, 132].

2.4 Trusted execution environments

A trusted execution environment (TEE) is a protected area in the CPU which guarantees untempered execution of code and keeps data confidential. Most popular CPU vendors have some form of TEEs (e.g., Intel: SGX [122], AMD: Secure Memory Encryption (SME [65]), RISC-V: Keystone [222] or ARM's Realm [24]). In the context of virtualisation, TEEs are interesting because they allow protecting data and computation from other tenants on the same machine and, in some cases, even from the cloud provider itself. To be of practical use in a cloud environment, TEEs must also support remote attestation so that it can remotely measure the hardware to determine if it is a genuine and secure device and is not emulated by a malicious actor. Early TEEs, such as ARM trustzones, required the physical provisioning of keys to establish remote attestation. While this works well for embedded devices where the device manufacturer manages the firmware and updates for the device, in cloud computing it is unlikely that the user deploying their application will ever see the hardware running their code. This trust bootstrapping problem was solved when Intel introduced SGX, which we discuss in more detail in the next section.

2.4.1 SGX

We use SGX [122] in RKT-IO (Chapter 5) to protect applications running in containers/virtual machines in the cloud. However, the concept we use should be transferable to similar TEE implementations on other architectures, e.g., RISC-V's Keystone [222], ARM's Realm [24] or AMD's SME [65]. SGX is a feature in modern Intel CPUs (starting with the Skylake generation in 2015) that allows computation to be performed securely and confidentially on hardware owned by an untrusted third-party [62].

SGX-enabled Intel processors come with a key provisioned securely in the chip. A user can use an Intel-powered remote attestation service to ensure that their application runs unmodified in a protected environment called *Enclave* [246]. Although previous technologies such as *TPMs* [257] (Trusted Platform Module) and *Intel TXT* [118] already offered this feature for the entire operating system and virtual machines, respectively, SGX allows enclaves to be run for single processes [62]. This significantly reduces the size of the TCB compared to previous approaches (see Figure 2.6). Other processes or the operating system cannot access the *Enclave* data because the hardware transparently encrypts all memory for the application in a special area called EPC (Enclave Page Cache). In older SGX generations the EPC was limited to 128 MB (later increased to 256 MB) of memory and required expensive swapping involving the operating system for memory pages that exceeded this limit. In the third generation of Intel SGX, released in 2021, this limit was increased to 1 TB [273], which imposes significantly less overhead for real-world applications.

To perform I/O, the application still relies on the untrusted operating system by executing system calls. For I/O-heavy applications, this can become a bottleneck since system calls also involve expensive context switches between the SGX enclave and the OS. To reduce the overhead, previous SGX frameworks [25, 197, 208] have introduced *switchless* designs that offload system calls to unprotected I/O threads outside and communicate with these threads via shared memory queues. However, RKT-IO communicates directly with hardware using userspace direct I/O frameworks to bypass the kernel, which we describe in more detail in the next section.

2.5 Direct I/O frameworks

Direct I/O frameworks bypass parts of the operating system or the entire operating system to gain faster access to the underlying hardware (see Figure 2.7). They are particularly popular in networking applications (e.g., netmap [223], PF_RING [110],

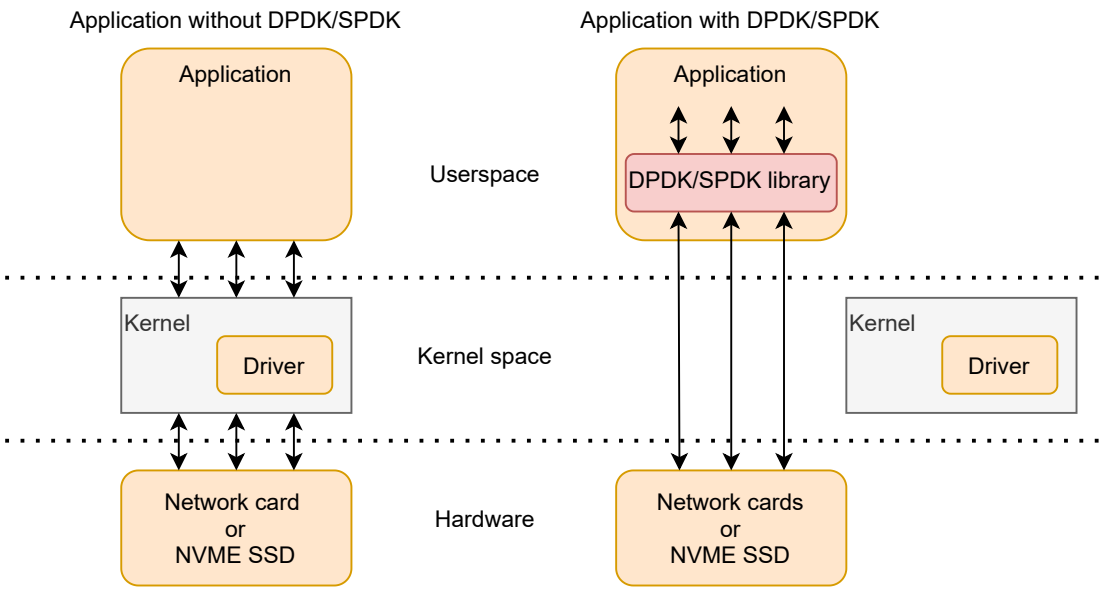


Figure 2.7: Direct I/O frameworks like DPDK/SPDK provide direct access to the underlying networking and storage hardware from within the application. This allows high-performance applications to exchange data with the devices faster by reducing the need for context switches, data copies, and allocations.

DPDK [77]), where a large amount of data packets must be processed in software (e.g., 100Gbit/s network cards), but are also on the rise for storage-oriented application with the introduction of fast NVME devices. Commonly, devices use interrupts to notify the CPU of data availability. Interrupts cause a context switch to the operating system and a jump to the registered interrupt handler. This is efficient for CPU-bound applications because the OS is invoked only when needed. However, for I/O-intensive applications, this can be costly and lead to a problem known as *Interrupt storm*, where the software becomes unable to make progress due to the high number of interrupts. Although, this problem has been addressed with the introduction of hybrid-polling mode drivers [79] and interrupt rate limiting in hardware, data copies between buffers in userspace and hardware buffers and dynamic memory allocations in the kernel [223] required for the POSIX API remain an issue. Frameworks such as Intel's DPDK and SPDK or Solarflare's Onload bypass the kernel by mapping device memory buffers and registers into the application's address space. Instead of relying on interrupts that can only be received by an operating system in privileged mode, they poll device registers for updates. The observation here is that for I/O-heavy applications, new data is always available and therefore interrupts are not needed as hardware queues need to be processed frequently. By accessing I/O memory directly, the application does not need to copy

data and memory management can be optimised, e.g., by pre-allocating all required buffers.

In RKT-IO we use DPDK and SPDK, which we will explain in the next section.

2.5.1 DPDK

The Data Plan Development Kit (DPDK [77]) is a cross-platform library for Linux, Windows and FreeBSD that also abstracts from different network hardware and provides a userspace interface for writing network applications. DPDK is mainly optimised for processing data packets and does not provide a high-level TCP/IP stack or socket abstraction.

2.5.2 SPDK

The Storage Performance Development Kit (SPDK [121]) builds on top of DPDK, but instead provides access to NVME devices. NVME is a standardised interface for non-volatile memory over PCIe or the network. SPDK provides a low-level block device interface to NVME as well as a block allocator called Blobstore [43] that can allocate objects on a device. SPDK also provides a very simple, non-POSIX filesystem called Blobfs [44], which is based on Blobstore.

Both DPDK and SPDK require developers to port their applications to their respective APIs, as they do not provide a common POSIX API. Therefore, to simplify the porting of legacy applications we ported a library OS in RKT-IO to DPDK/SPDK to provide applications with a network and filesystem stack, which we explain in the next section.

2.6 Library OS

Unlike monolithic kernels or microkernels, library operating systems (libOSes, also known as unikernels [21]) reside in the same CPU protection ring and address space as the application. Due to the lack of memory protection, libOSes can often host only a single process or program. On the upside, system calls are cheaper compared to general-purpose OSes because system calls do not require a CPU context switch. Instead of the context switch, the application makes a function call to the linked library operating system code. Since the I/O abstraction (i.e., filesystem or network stack) is in the same address space as the application, the hardware can be accessed more directly and with fewer additional data copies.

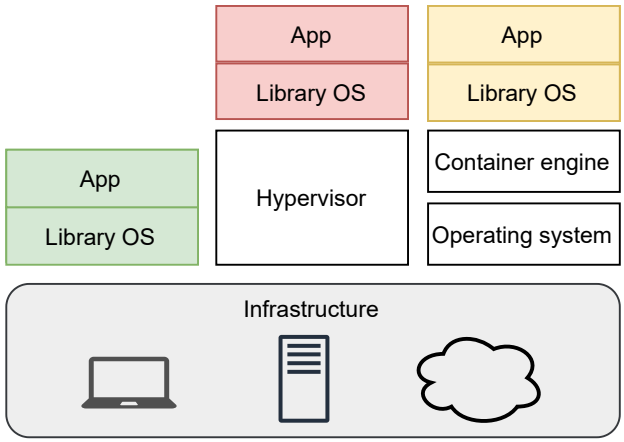


Figure 2.8: Library operating systems come in many forms. Some run directly on hardware, some run only in virtual machines, and others may run as processes in userspace. What they have in common is that the operating system functionality is linked against the application as a library.

2.6.1 SGX-LKL

RKT-IO takes advantage of this to improve I/O performance in trusted execution environments by bypassing the host kernel and using the filesystem and network stack from SGX-LKL [208] (see Figure 2.9). The Linux Kernel Library (LKL) [192] is an architectural no-MMU port of the Linux kernel that brings kernel functionality to userspace in the form of a libOS. Since LKL is based on Linux, it provides high compatibility with POSIX APIs and relies on well-tested code. To run LKL within the trusted execution environment, SGX-LKL provides a small memory and threading abstraction. This abstraction allocates memory in the encrypted enclave memory regions. LKL's kernel threads and application threads are scheduled by the userland scheduler to allow faster context switching without leaving the enclave. By including a port of the Musl libc [185], SGX-LKL provides a fully binary-compatible interface for applications. Instead of interacting with the untrusted host OS, its libc invokes LKL for system calls. SGX-LKL performs I/O by using virtio-based block and network devices. These devices are emulated by untrusted threads running outside the protected enclave. The filesystem and network stack of SGX-LKL interacts with these devices and transparently encrypt application data leaving the enclave (i.e., with full disk encryption and VPN encryption). Device operations are then mapped to a TAP interface for the network device and a file for the block device in the host OS. This separation greatly improves the security of applications that are not designed to run on an untrusted host OS, since they do not directly use the host's syscall API.

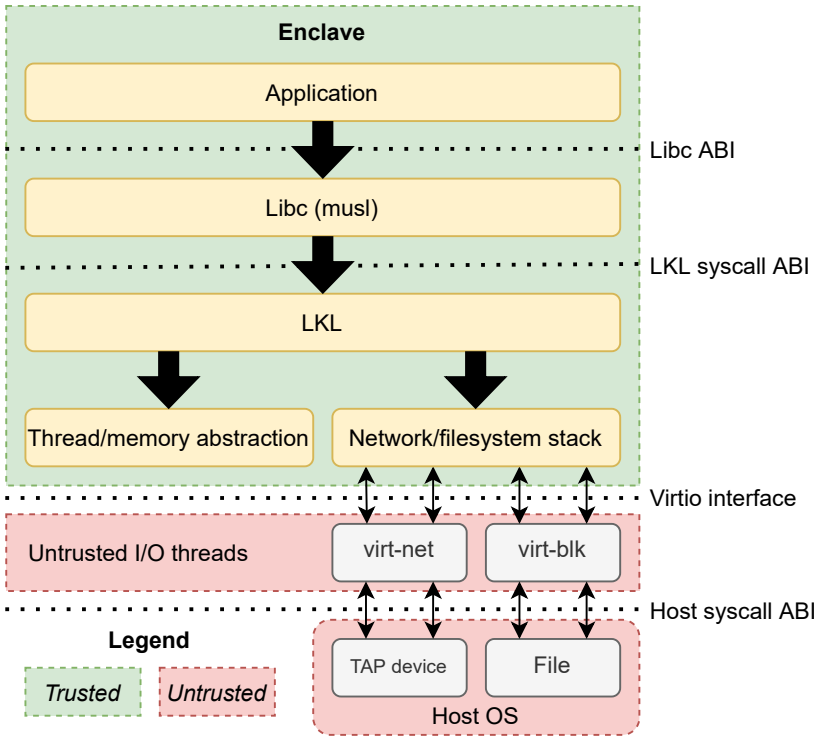


Figure 2.9: Architectural overview over SGX-LKL.

Having provided the necessary background information, we would like to present our three contributions to improving the dependability of virtualised systems.

Chapter 3

CNTR : Lightweight OS Containers

Container-based virtualisation has become the de-facto standard for deploying applications in data centres. However, deployed containers frequently include a wide-range of tools (e.g., debuggers) that are not required for applications in the common use-case, but they are included for rare occasions such as in-production debugging. As a consequence, containers are significantly larger than necessary for the common case, thus increasing the build and deployment time.

CNTR[1] provides the *performance* benefits of lightweight containers and the *functionality* of large containers by splitting the traditional container image into two parts: the "fat" image — containing the tools, and the "slim" image — containing the main application. At run-time, CNTR allows the user to efficiently deploy the "slim" image and then expand it with additional tools, when and if necessary, by dynamically attaching the "fat" image.

To achieve this, CNTR transparently combines the two container images using a new nested namespace, without any modification to the application, the container manager, or the operating system. We have implemented CNTR in Rust, using FUSE, and incorporated a range of optimizations. CNTR supports the full Linux filesystem API, and it is compatible with all container implementations (i.e., Docker, rkt, LXC, systemd-nspawn). Through extensive evaluation, we show that CNTR incurs reasonable performance overhead while reducing, on average, by 66.6% the image size of the Top-50 images available on Docker Hub.

[1] Read it as "center".

3.1 Introduction

Containers offer an appealing, lightweight alternative to VM-based virtualisation (e.g., KVM, VMware, Xen) that relies on process-based virtualisation. Linux, for instance, provides the `cgroups` and `namespaces` mechanisms that enable strong performance and security isolation between containers [153]. Lightweight virtualisation is fundamental to achieve high efficiency in virtualised datacenters and enables important use-cases, namely just-in-time deployment of applications. Moreover, containers significantly reduce operational costs through higher consolidation density and power minimization, especially in multi-tenant environments. Because of all these advantages, it is no surprise that containers have seen wide-spread adoption by industry, in many cases replacing altogether traditional virtualisation solutions [101].

Despite being lightweight, deployed containers often include a wide-range of tools such as shells, editors, coreutils, and package managers. These additional tools are usually not required for the application's core function — the common operational use-case — but they are included for management, manual inspection, profiling, and debugging purposes [253]. In practice, this significantly *increases container size* and, in turn, translates into slower container deployment and inefficient datacenter resource usage (network bandwidth, CPU, RAM and disk). Furthermore, larger images degrade container deployment time [78, 22]. For instance, previous work reported that downloading container images account for 92% of the deployment time [78]. Moreover, a larger code base directly affects the reliability of applications in datacenters [41].

Given the impact of using large containers, users are discouraged from including additional tools that would otherwise simplify the process of debugging, deploying, and managing containers. To mitigate this problem, Docker has recently adopted smaller run-times but, unfortunately, these efforts come at the expense of compatibility problems and have limited benefits [73].

To quantify the practical impact of additional tools on the container image size, we employed Docker Slim [71] on the 50 most popular container images available on the Docker Hub repository [70]. Docker Slim uses a combination of static and dynamic analyses to generate smaller-sized container images, in which, only files needed by the core application are included in the final image. The results of this experiment (see Figure 3.6) are encouraging: we observe that by excluding unnecessary files from typical containers it is possible to reduce the container size, on average, by 66.6%. Similarly, others have found that a only small subset (6.4%) of the container images is read in the common case [105].

CNTR addresses this problem[2] by building lightweight containers that still remain fully functional, even in uncommon use-cases (e.g., debugging and profiling). CNTR enables users to deploy the application and its dependencies, while the additional tools required for other use-cases are supported by expanding the container "on-demand", during runtime (Figure 3.1 (a)). More specifically, CNTR splits the traditional container image into two parts: the "fat" image containing the rarely used tools and the "slim" image containing the core application and its dependencies.

During runtime, CNTR allows the user of a container to efficiently deploy the "slim" image and then expand it with additional tools, when and if necessary, by dynamically attaching the "fat" image. As an alternative to using a "fat" image, CNTR allows tools from the *host* to run inside the container. The design of CNTR simultaneously preserves the performance benefits of lightweight containers and provides support for additional functionality required by different application workflows.

The key idea behind our approach is to create a new *nested namespace* inside the application container (i.e., "slim container"), which provides access to the resources in the "fat" container, or the host, through a FUSE filesystem interface. CNTR uses the FUSE system to combine the filesystems of two images without any modification to the application, the container implementation, or the operating system. CNTR selectively redirects the filesystem requests between the mount namespace of the container (i.e., what applications within the container observe and access) and the "fat" container image or the host, based on the filesystem request path. Importantly, CNTR supports the full Linux filesystem API and all container implementations (i.e., Docker, rkt, LXC, systemd-nspawn).

We evaluat CNTR across three key dimensions: (1) *functional completeness* – CNTR passes 90 out of 94 (95.74%) `xfstests` filesystem regression tests [82] supporting applications such as SQLite, Postgres, and Apache; (2) *performance* – CNTR incurs reasonable overheads for the `Phoronix` filesystem benchmark suite [203], and the proposed optimizations significantly improve the overall performance; and lastly, (3) *effectiveness* – CNTR's approach on average results in a 66.6% reduction of image size for the Top-50 images available on Docker hub [70]. We make the CNTR implementation along with the experimental setup [56] publicly available.

[2]Note that Docker Slim [71] does not solve the problem; it simply identifies the files not required by the application, and excludes them from the container, but it does not allow users to access those files at run-time.

3.2 Motivation

3.2.1 Container-based virtualisation

Containers as introduced in Section 2.1 consist of a lightweight, process-level form of virtualisation that is widely used and has become a cornerstone technology for datacenters and cloud computing providers. Unlike VM-based virtualisation, containers do not include a guest kernel and thus have often smaller memory footprint than traditional VMs. Containers have important advantages over VMs for both users and data centres:

1. **Faster deployment.** Containers are transferred and deployed faster from the registry [22].
2. **Lower resource usage.** Containers consume fewer resources and incur less performance overhead [236].
3. **Lower build times.** Containers with fewer binaries and data can be rebuilt faster [253].

Unfortunately, containers in practice are still unnecessarily large because users are forced to decide which auxiliary tools (e.g. debugging, profiling, etc.) should be included in containers at *packaging-time*. In essence, users are currently forced to strike a balance between lightweight containers and functional containers, and end up with containers that are neither as light nor as functional as desirable.

3.2.2 Traditional approaches to minimize containers

The container-size problem has been a significant source of concern to users and developers. Unfortunately, existing solutions are neither practical nor efficient.

An approach that has gained traction, and has been adopted by Docker, consists of packing containers using smaller base distributions when building the container runtime. For instance, most of Docker's containers are now based on the Alpine Linux distribution [73], resulting in smaller containers than traditional distributions. Alpine Linux uses the `musl` library, instead of `libc`, and includes `busybox`, instead of `coreutils` — these differences enable a smaller container runtime but at the expense of compatibility problems caused by runtime differences. Further, the set of tools included is still restricted and fundamentally does not help users when less common auxiliary tools are required (e.g., custom debugging tools).

The second approach to reduce the size of containers relies on union filesystems (e.g., UnionFS [215]). Docker, for instance, enables users to create their containers on

top of commonly-used base images. Because such base images are expected to be shared across different containers (and already deployed in the machines), deploying the container only requires sending the diff between the base image and the final image. However, in practice, users still end up with multiple base images due to the use of different base image distributions across different containers.

Another approach that has been proposed relies on the use of *unikernels* [157, 155], a single-address-space image constructed from a *library OS* [230, 36, 259]. By removing layers of abstraction (e.g., processes) from the OS, the unikernel approach can be leveraged to build very small virtual machines—this technique has been considered as containerization because of its low overhead, even though it relies on VM-based virtualisation. However, unikernels require additional auxiliary tools to be statically linked into the application image; thus, it leads to the same problem.

3.2.3 Background: container internals

The Linux operating system achieves isolation through an abstraction called namespaces (see 2.1.1). During the container startup, by default, namespaces of the host are unshared. Hence, processes inside the container only see files from their filesystem image (see Figure 3.1 (a)) or additional volumes, that have been statically added during setup. New mounts on the host are not propagated to the container since by default, the container runtime will mount all mount points as private.

3.2.4 Use-cases of CNTR

We envision three major use cases for CNTR that cover three different debugging/management scenarios:

Container to container debugging in production CNTR enables the isolation of debugging and administration tools in *debugging containers* and allows application containers to use debugging containers on-demand. Consequently, application containers become leaner, and the isolation of debugging/administration tools from applications allows users to have a more consistent debugging experience. Rather than relying on disparate tools in different containers, CNTR allows using a single debugging container to serve many application containers. Containers are usually built and deployed to provide a single service. Ideally, they should only contain files that are needed to run the service to minimize the complexity and the need for (security) updates as well as to save disk space. However, oftentimes, developers will require debugging and adminsitration tools to be included in containers for troubleshooting and management purposes (e.g., gdb, strace, tcpdump, curl, ...).

These tools are not necessarily required for the actual service, but they are heavily used in practice.

Host to container debugging CNTR allows developers to use the debugging environments (e.g., IDEs) in their host machines to debug containers that do not have these environments installed. These IDEs can sometimes take several gigabytes of disk space and might be not even compatible with the distribution of the container image is based on. Another benefit of using CNTR in this context is that development environments and settings can be also efficiently shared across different containers.

Container to host administration and debugging Container-oriented Linux distributions such as CoreOS [61] or RancherOS [217] do not provide a package manager and users need to extend these systems by installing containers even for basic system services. CNTR allows a user of a privileged container to access the root filesystem of the host operating system. Consequently, administrators can keep tools installed in a debug container while keeping the host operating system's filesystem lean.

3.3 Design

In this section, we present the detailed design of CNTR.

3.3.1 System overview

Design goals CNTR has the following design goals:

- *Generality:* CNTR should support a wide-range of workflows for seamless management and problem diagnosis (e.g., debugging, tracing, profiling).
- *Transparency:* CNTR should support these workflows without modifying the application, the container manager, or the operating system. Further, we want to be compatible with all container implementations.
- *Efficiency:* Lastly, CNTR should incur low performance overheads with the split-container approach.

Basic design CNTR is composed of two main components (see Figure 3.1 (a)): a nested namespace, and the CNTRFS filesystem. In particular, CNTR combines slim and fat containers by creating a new *nested namespace* to merge the namespaces of two containers (see Figure 3.1 (b)). The nested namespace allows CNTR to selectively

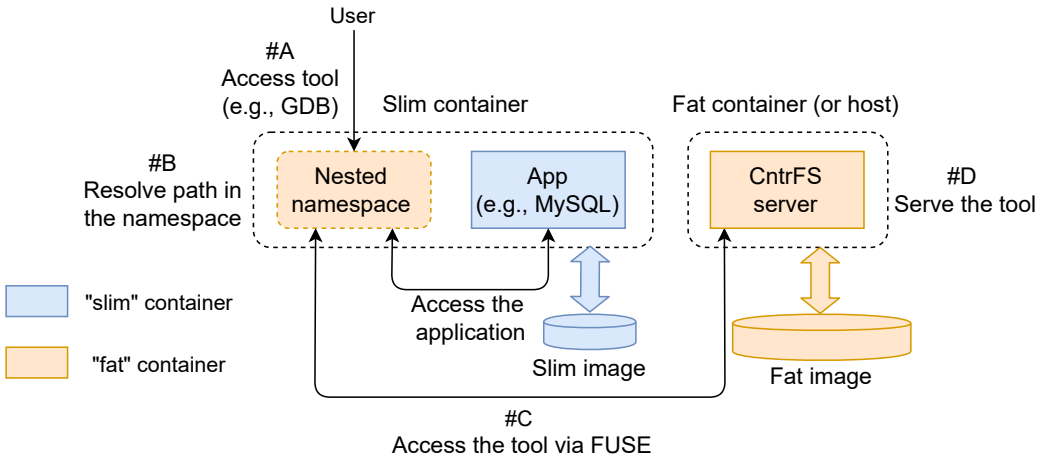


Figure 3.1: Basic design

break the isolation between the two containers by transparently redirecting the requests based on the accessed path. CNTR achieves this redirection using the CNTRFS filesystem. CNTRFS is mounted as the root filesystem (/), and the application filesystem is remounted to another path (`/var/lib/cntr`) in the nested namespace. CNTRFS implements a filesystem in userspace (FUSE), where the CNTRFS server handles the requests for auxiliary tools installed on the fat container (or on the host).

At a high-level, CNTR connects with the CNTRFS server via the generic FUSE kernel driver. The kernel driver simply acts as a proxy between processes accessing CNTRFS, through Linux VFS, and the CNTRFS server running in userspace. The CNTRFS server can be in a different mount namespace than the nested namespace, therefore, CNTR establishes a proxy between two mount namespaces through a request/response protocol. This allows a process that has all its files stored in the fat container (or the host) to run within the mount namespace of the slim container.

Cntr workflow CNTR is easy to use. The user simply needs to specify the name of the "slim" container and, in case the tools are in another container, the name of the "fat" container. CNTR exposes a shell to the user that has access to the resources of the application container as well as the resources forwarded from the fat container.

Figure 3.1 (a) explains the workflow of CNTR when a user requests to access a tool from the slim container (#A): CNTR transparently resolves the requested path for the tool in the nested namespace (#B). Figure 3.1 (b) shows an example of CNTR's nested namespace, where the requested tool (e.g., gdb) is residing in the fat container. After resolving the path, CNTR redirects the request via FUSE to the fat container (#C). Lastly, CNTRFS serves the requested tool via the FUSE interface (#D). Behind the

scenes, CNTR executes the following steps:

1. *Resolve container name to process ID and get container context.* CNTR resolves the name of the underlying container process IDs and then queries the kernel to get the complete execution context of the container (container namespaces, environment variables, capabilities, . . .).

2. *Launch the* CNTRFS *server.* CNTR launches the CNTRFS server. CNTR launches the server either directly on the host or inside the specified "fat" container containing the tools image, depending on the settings that the user specified.

3. *Initialize the tools' namespace.* Subsequently, CNTR attaches itself to the application container by setting up a nested mount namespace within the namespace of the application container. CNTR then assigns a forked process to the new namespace. Inside the new namespace, the CNTR process proceeds to mount CNTRFS, providing access to files that are normally out of the scope of the application container.

4. *Initiate an interactive shell.* Based on the configuration files within the debug container or on the host, CNTR executes an interactive shell, within the nested namespace, that the user can interact with. CNTR forwards its input/output to the user terminal (on the host). From the shell, or through the tools it launches, the user can then access the application filesystem under `/var/lib/cntr` and the tools' filesystem in `/`. Importantly, tools have the same view on system resources as the application (e.g., `/proc`, `ptrace`). Furthermore, to enable the use of graphical applications, CNTR forwards Unix sockets from the host/debug container.

3.3.2 Design details

This section explains the design details of CNTR.

3.3.2.1 Step #1: Resolve container name and obtain container context

Because the kernel has no concept of a container name or ID, CNTR starts by resolving the container name, as defined by the used container manager, to the process IDs running inside the container. CNTR leverages wrappers based on the container management command line tools to achieve this translation and currently, it supports Docker, LXC, rkt, LXD, Podman, Containerd, and systemd-nspawn.

After identifying the process IDs of the container, CNTR gathers OS-level information about the container namespace. CNTR reads this information by

inspecting the `/proc` filesystem of the main process within the container. This information enables CNTR to create processes inside the container in a *transparent* and *portable* way.

In particular, CNTR gathers information about the container namespaces, cgroups (resource usage limits), mandatory access control (e.g., AppArmor [174] and SELinux [111] options), user ID mapping, group ID mapping, capabilities (fine-grained control over super-user permissions), and process environment options. Additionally, CNTR could also read the seccomp options, but this would require non-standard kernel compile-time options and generally has limited value because seccomp options have significant overlap with the capability options. CNTR reads the environment variables because they are heavily used in containers for configuration and service discovery [262].

Before attaching to the container, in addition, to gather the information about the container context, the CNTR process opens the FUSE control socket (`/dev/fuse`). This file descriptor is required to mount the CNTRFS filesystem, after attaching to the container.

3.3.2.2 Step #2: Launch the CNTRFS server

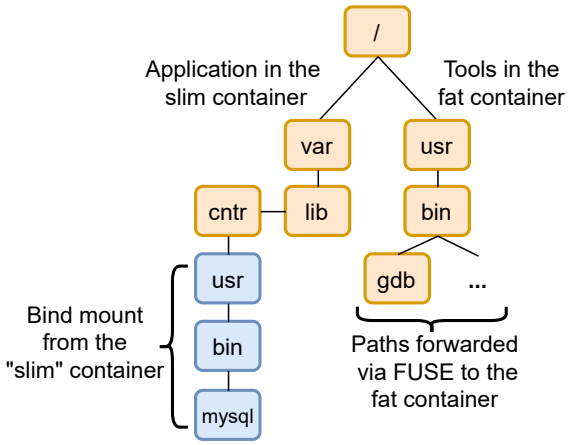


Figure 3.2: Example of nested namespace

The CNTRFS is executed either directly on the host or inside the "fat" container, depending on the option specified by the user (i.e., the location of the tools). In the host case the CNTRFS server simply runs like a normal host process.

In case the user wants to use tools from the "fat" container, the CNTRFS process forks and attaches itself to the "fat" container. Attaching to the "fat" container is implemented by calling the `setns()` system call, thereby assigning the child process to the container namespace that was collected in the previous step.

After initialization, the CNTRFS server waits for a signal from the nested namespace (Step #3) before it starts reading and serving the FUSE requests (reading before an unmounted FUSE filesystem would otherwise return an error). The FUSE requests then will be read from the `/dev/fuse` file descriptor and redirected to the filesystem of the server namespace (i.e., host or fat container).

3.3.2.3 Step #3: Initialize the tools namespace

CNTR initializes the tool namespace by first attaching to the container specified by the user—the CNTR process forks and the child process assigns itself to the cgroup, by appropriately setting the `/sys/` option, and namespace of the container, using the `setns()` system call.

After attaching itself to the container, CNTR creates a new nested namespace, and marks all mountpoints as private so that further mount events (regarding the nested namespace) are not propagated back to the container namespace. Subsequently, CNTR creates a new filesystem hierarchy for the nested namespace, mounting the CNTRFS in a temporary mountpoint (`TMP/`).

Within the nested namespace, the child process mounts CNTRFS, at `TMP/`, and signals the parent process (running outside of the container) to start serving requests. Signalling between the parent and child CNTR processes is implemented through a shared Unix socket.

Within the nested namespace, the child process remounts all pre-existing mountpoints, from the application container, by moving them from `/` to `TMP/var/lib/cntr`. Note that the application container is not affected by this since all mountpoints are marked as private.

In addition, CNTR also mounts special container-specific files from the application over files from the tools or host (using bind mount [184]). The special files include the pseudo filesystems `procfs` (`/proc`), ensuring the tools can access the container application, and devtmpfs (`/dev`), containing block and character devices that have been made visible to our container. Furthermore, we bind mount a set of configuration files from the application container into the temporary directory (e.g., `/etc/passwd`, and `/etc/hostname`).

Once the new filesystem hierarchy has been created in the temporary directory, CNTR atomically executes a `chroot` turning the temporary directory (`TMP/`) into the new root directory (`/`).

To conclude the container attachment and preserve the container isolation guarantees, CNTR updates the remaining properties of the nested namespace: (1) CNTR drops the capabilities by applying the AppArmor/SELinuxprofile and (2)

CNTR applies all the environment variables that were read from the container process; with the exception of `PATH` – the `PATH` is instead inherited from the debug container since it is often required by the tools.

3.3.2.4 Step #4: Start interactive shell

Lastly, CNTR launches an interactive shell within the nested namespace, enabling users to execute the tools. CNTR forwards the shell I/O using a pseudo-TTY, and supports graphical interface using Unix sockets forwarding.

Shell I/O Interactive shells perform I/O through standard file descriptors (i.e., stdin, stdout, and stderr file descriptors) that generally refer to terminal devices. For isolation and security reasons, CNTR prevents leaking the terminal file descriptors of the host to a container by leveraging pseudo-TTYs – the pseudo-TTY acts as a proxy between the interactive shell and the user terminal device.

Unix socket forwarding CNTR forwards connections to Unix sockets, e.g., the X11 server socket and the D-Bus daemon running on the host. Unix sockets are also visible as files in our FUSE. However, since our FUSE has inode numbers that are different from the underlying filesystem, the kernel does not associate them with open sockets in the system. Therefore, we implemented a socket proxy that runs an efficient event loop based on epoll. It uses the splice syscall to move data between clients in the application container and servers listening on Unix sockets in the debug container/host.

3.3.3 Optimizations

We experienced performance slowdown in CNTRFS when we measured the performance using the Phoronix benchmark suite [203] (§3.5.2). Therefore, we incorporate the following performance optimizations in CNTR.

Caching: Read and writeback caches A major performance improvement comes from allowing the FUSE kernel module to cache data returned from the read requests as well as setting up a writeback buffer for the writes. CNTR avoids automatic cache invalidation when a file is opened by setting the `FOPEN_KEEP_CACHE` flag. Without this flag the cache cannot be effectively shared across different processes. To allow the FUSE kernel module to batch smaller write requests, we also enable the writeback cache by specifying the `FUSE_WRITEBACK_CACHE` flag at the mount setup time. This optimization sacrifices write consistency for performance by delaying the sync

operation. However, we show that it still performs correctly according to the POSIX semantics in our regression experiments (see § 3.5.1).

Multithreading Since the I/O operations can block, we optimize the CNTRFS implementation by using multiple threads. In particular, CNTR spawns independent threads to read from the CNTRFS file descriptor independently to avoid contentions while processing the I/O requests.

Batching In addition to caching, we also batch operations to reduce the number of context switches. In particular, we apply the batching optimization in three places: (a) pending inode lookups, (b) forget requests, and (c) concurrent read requests.

Firstly, we allow concurrent inode lookups by applying `FUSE_PARALLEL_DIROPS` option on mount. Secondly, the operating system sends `forget` requests, when inodes can be freed up by CNTRFS. The kernel can batch a forget intent for multiple inodes into a single request. In CNTR we have also implemented this request type. Lastly, we set `FUSE_ASYNC_READ` to allow the kernel to batch multiple concurrent read requests at once to improve the responsiveness of read operations.

Splicing: Read and write Previous work suggested the use of splice reads and writes to improve the performance of FUSE [266]. The idea behind splice operation is to avoid copying data from and to userspace. CNTR uses splice for read operations. Therefore, the FUSE userspace process moves data from the source file descriptor into a kernel pipe buffer and then to the destination file descriptor with the help of the `splice` syscall. Since splice does not actually copy the data but instead remaps references in the kernel, it reduces the overhead.

We also implement a splice write optimization. In particular, we use a pipe as a temporary storage, where the data is part of the request, and the data is not read from a file descriptor. However, FUSE does not allow to read the request header into userspace without reading the attached data. Therefore, CNTR has to move the whole request to a kernel pipe first in order to be able to read the request header separately. After parsing the header it can move the remaining data to its designated file descriptor using the splice operation. However, this introduces an additional context switch, and slowdowns all FUSE operations since it is not possible to know in advance if the next request will be a write request. Therefore, we do not to enable this optimization by default.

3.4 Implementation

To ensure portability and maintainability, we decided not to rely on container-specific APIs, since they change quite often. Instead, we build our system to be as generic as possible by leveraging more stable operating system interfaces. Our system implementation supports all major container types: Docker, LXC, rkt, LXD, Podman, Containerd, and systemd-nspawn. CNTR's implementation resolves container names to process IDs. Process IDs are handled in an implementation-specific way. On average, we add only 70 LoCs for each container implementation to support a new container engine.

At a high-level, our system implementation consists of the following four components:

- *Container engine* (1549 LoC) analyzes the container that a user wants to attach to. The container engine also creates a nested mount namespace, where it starts the interactive shell.
- CNTRFS (1481 LoC) to serve the files from the fat container. We implement CNTRFS based on Rust-FUSE [226]. We extend Rust-FUSE to be able to mount across mount namespaces and without a dedicated FUSE mount executable.
- A *pseudo TTY* (221 LoC) to connect the shell input/output with the user terminal.
- A *socket proxy* (400 LoC) to forward the Unix socket connection between the fat (or the host) and slim containers for supporting X11 applications.

All core system components of CNTR are implemented in Rust (total 3651 LoC). To simplify deployment, we do not depend on any non-Rust libraries. In this way, we can compile CNTR as a $\sim$ 1.2 MB single self-contained static executable by linking against musl-libc [185]. This design is imperative to ensure that CNTR can run on container-optimized Linux distributions, such as CoreOS [61] or RancherOS [217], that do not have a package manager to install additional libraries.

Since CNTR makes heavy use of low-level filesystem system calls, we have also extended the Rust ecosystem with additional 46 system calls to support the complete Linux filesystem API. In particular, we extend the nix Rust library [227], a library wrapper around the Linux/`POSIX` API. The changes are available in our fork [199].

3.5 Evaluation

In this section, we present the experimental evaluation of CNTR. Our evaluation answers the following questions.

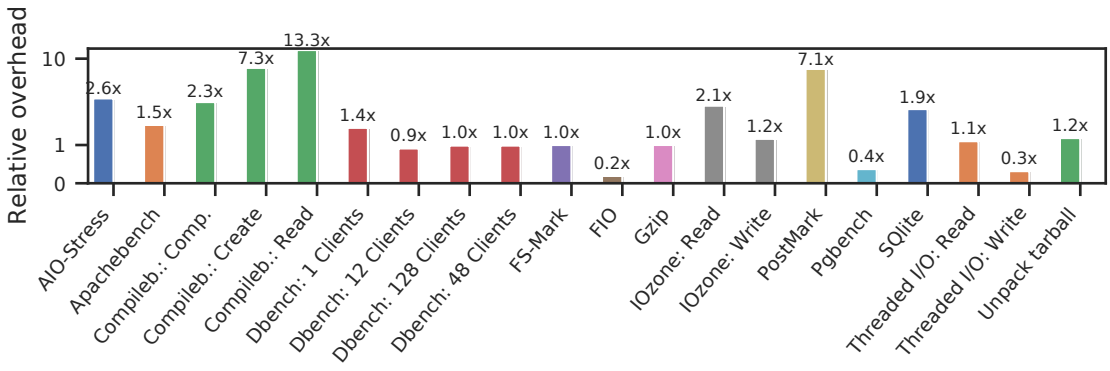


Figure 3.3: Relative performance overheads of CNTR compared to native file access for the `Phoronix` suite. The absolute values for each benchmark is available online on the openbenchmark platform [218].

1. Is the implementation complete and correct? (§3.5.1)
2. What are the performance overheads and how effective are the proposed optimizations? (§3.5.2)
3. How effective is the approach to reducing container image sizes? (§3.5.3)

3.5.1 Completeness and correctness

We first evaluate the completeness and correctness claim of the CNTR implementation. The primary goal is to evaluate whether CNTR implements the same features (completeness) as required by the underlying filesystem, and it follows the same `POSIX` semantics (correctness).

Benchmark: xfstests regression test suite For this experiment, we use the `xfstests` [82] filesystem regression test suite. The `xfstests` suite was originally designed for the XFS filesystem, but it is now widely used for testing all of Linux's major filesystems. It is regularly used for quality assurance before applying changes to the filesystem code in the Linux kernel. `xfstests` contains tests suites to ensure correct behavior of all filesystem related system calls and their edge cases. It also includes crash scenarios and stress tests to verify if the filesystem correctly behaves under load. Further, it contains many tests for bugs reported in the past.

Methodology We extend `xfstests` to support mounting CNTRFS. For running tests, we mount CNTRFS on top of `tmpfs`, an in-memory filesystem. We run all tests in the generic group once.

Experimental results `xfstests` consists of 94 unit tests that can be grouped into the following major categories: *auto*, *quick*, *aio*, *prealloc*, *ioctl*, and *dangerous*.

Overall, CNTR passed 90 out of 94 (95.74%) unit tests in `xfstests`. Four tests fail due minor implementation details that we currently do not support. Specifically, these four unit tests are automatically skipped by `xfstests` because they expect our filesystem to be backed by a block device or expected some missing features in the underlying `tmpfs` filesystem, e.g. copy-on-write ioctl. We next explain the reasons for the failed four test cases:

1. **Test #375** fails since `SETGID` bits are not cleared in `chmod` when the owner is not in the owning group of the access control list. This would require manual parsing and interpreting ACLs in CNTR. In our implementation, we delegate `POSIX` ACLs to the underlying filesystem by using `setfsuid/setfsguid` on inode creation.

2. **Test #228** fails since we do not enforce the per-process file size limits (`RLIMIT_FSIZE`). As replay file operations and `RLIMIT_FSIZE` of the caller is not set or enforced in CNTRFS, this has no effect.

3. **Test #391** fails since we currently do not support the direct I/O flag in `open` calls. The support for direct I/O and `mmap` in FUSE is mutually exclusive. We chose `mmap` here, since we need it to execute processes. In practice, this is not a problem because not all docker drivers support this feature, including the popular filesystems such as `overlayfs` and `zfs`.

4. **Test #426** fails since our inodes are not exportable. In Linux, a process can get inode references from filesystems by the `name_to_handle_at` system call. However, our inodes are not persisted and are dynamically requested and destroyed by the operating system. If the operating system no longer uses them, they become invalid. Many container implementations block this system call as it has security implications.

To summarize, the aforementioned failed test cases are specific to our current state of the implementation, and they should not affect most real-world applications. As such, these features are not required according to the `POSIX` standard, but, they are Linux-specific implementation details.

3.5.2 Performance overheads and optimizations

We next report the performance overheads for CNTR's split-containers approach (§3.5.2.1), detailed experimental results (§3.5.2.2), and effectiveness of the proposed

optimizations (§3.5.2.3).

Experimental testbed To evaluate a realistic environment for container deployments [14], we evaluate the performance benchmarks using `m4.xlarge` virtual machine instances on Amazon EC2. The machine type has two cores of Intel Xeon E5-2686 CPU (4 hardware threads) assigned and 16GB RAM. The Linux kernel version was 4.14.13. For storage, we use a 100GB EBS volume of type GP2 formatted with `ext4` filesystem mounted with default options. GP2 is an SSD-backed storage and attached via a dedicated network to the VM. Amazon recommends this storage for most workloads and low-latency application [14].

Benchmark: Phoronix suite For the performance measurement, we use the disk benchmarks [210] from the `Phoronix` suite [203]. `Phoronix` is a meta benchmark that has a wide range of common filesystem benchmarks, applications, and realistic workloads. We compile the benchmarks with GCC 6.4 and CNTR with Rust 1.23.0.

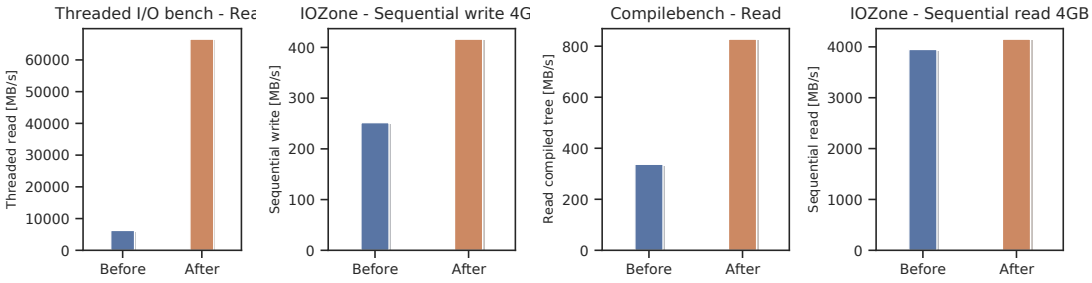
Methodology For the performance comparison, we run the benchmark suite once on the native filesystem (the baseline measurement) and compare the performance when we access the same filesystem through CNTRFS. The `Phoronix` benchmark suite runs each benchmark at least three times and automatically adds additional trials if the variance is too high. To compute the relative overheads with respect to the baseline, we compute the ratio between the native filesystem access and CNTRFS (*native*/*cntr*) for benchmarks where higher values are better (e.g. throughput), and the inverse ratio (*cntr*/*native*), where lower values are better (e.g. time required to complete the benchmark).

3.5.2.1 Performance overheads

We first present the summarized results for the entire benchmark suite. Thereafter, we present a detailed analysis of each benchmark individually (§3.5.2.2).

Summary of the results Figure 3.3 shows the relative performance overheads for all benchmarks in the `Phoronix` test suite. We have made the absolute numbers available for each benchmark on the openbenchmark platform [218].

Our experiment shows that 13 out of 20 (65%) benchmarks incur moderate overheads below 1.5× compared to the native case. In particular, three benchmarks showed significantly higher overheads, including `compilebench-create` (7.3×) and `compilebench-read` (13.3×) and the `postmark` benchmark (7.1×). Lastly, we also had



(a) Read cache (b) Writeback cache (c) Batching (d) Splicing

Figure 3.4: Effectiveness of optimizations

three benchmarks, where CNTRFS was faster than the native baseline execution: `FIO` (0.2×), `PostgreSQL Bench` (0.4×) and the write workload of `Threaded I/O` (0.3×).

To summarize, the results show the strengths and weaknesses of CNTRFS for different applications and under different workloads. At a high-level, we found that the performance of inode lookups and the double buffering in the page cache are the main performance bottlenecks in our design (much like they are for FUSE). Overall, the performance overhead of CNTR is reasonable. Importantly, note that while reporting performance numbers, we resort to the worst-case scenario for CNTR, where the "slim" application container aggressively uses the "fat" container to run an I/O-intensive benchmark suite. However, we must emphasize the primary goal of CNTR: to support auxiliary tools in uncommon operational use-cases, such as debugging or manual inspection, which are not dominated by high I/O-intensive workloads.

3.5.2.2 Detailed experimental results

We next detail the results for each benchmark.

AIO-Stress `AIO-Stress` submits 2GB of asynchronous write requests. In theory, CNTRFS supports asynchronous requests, but only when the filesystem operates in the direct I/O mode. However, the direct I/O mode in CNTRFS restricts the `mmap` system call, which is required by executables. Therefore, all requests are, in fact, processed synchronously resulting in 2.6× slowdown.

Apache Web server The `Apache Web server` benchmark issues 100K `http` requests for test files (average size of 3KB), where we notice a slowdown of up to 1.5×. However, the bottleneck is not due to serving the actual content, but due to writing of the webserver access log, which triggers small writes ($<$ 100 bytes) for each

request. These small requests trigger lookups in CNTRFS of the extended attributes `security.capabilities`, since the kernel currently neither caches such attributes nor it provides an option for caching them.

Compilebench `Compilebench` simulates different stages in the compilation process of the Linux kernel. There are three variants of the benchmark: (a) the `compile` stage compiles a kernel module, (b) the `read tree` stage reads a source tree recursively, and lastly, (c) the `initial creation` stage simulates a tarball unpack. In our experiments, `Compilebench` has the highest overhead of all benchmarks with the `read tree` stage being the slowest (13.4×). This is due to the fact that inode lookups in CNTRFS are slower compared to the native filesystem: for every lookup, we need one `open()` system call to get a file handle to the inode, followed by a `stat()` system call to check if we already have lookup-ed this inode in a different path due hardlinks. Usually, after the first lookup, this information is cached in the kernel, but in this benchmark for every run, a different source tree with many files are read. The slowdown of lookups for the other two variants, namely the `compile stage` (2.3×) and `initial create` (7.3×) is lower, since they are shadowed by write operations.

Dbench `Dbench` simulates a file server workload, and it also simulates clients reading files and directories with increasing concurrency. In this benchmark, we notice that with increasing number of clients, CNTRFS is able to cache directories and file contents in the kernel. Therefore, CNTRFS does not incur performance overhead over the native baseline.

FS-Mark `FS-Mark` sequentially creates 1000 1MB files. Since the write requests are reasonably large (16 KB per `write` call) and the workload is mostly disk bound. Therefore, there is no difference between CNTRFS and `ext4`.

FIO benchmark The `FIO` benchmark profiles a fileserver and measures the read/write bandwidth, where it issues 80% random reads and 20% random writes for 4GB data with an average blocksize of 140KB. For this benchmark, CNTRFS outperforms the native filesystem by a factor of 4× since the writeback cache leads to fewer and larger writes to the disk compared to the underlying filesystem.

Gzip benchmark The `Gzip` benchmark reads a 2GB file containing only zeros and writes the compressed version of it back to the disk. Even though the file is highly compressible, `gzip` compresses the file slower than the data access in CNTRFS or `ext4`.

Therefore, there is no significant performance difference between CNTR and the native version.

IOZone benchmark `IOZone` performs sequential writes followed by sequential reads of a blocksize of 4KB. For the write requests, as in the `apache` benchmark, CNTR incurs low overhead (1.2×) due to extended attribute lookup overheads. Whereas, for the sequential read request, both filesystems (underlying native filesystem and CNTRFS) can mostly serve the request from the page cache. For smaller read sizes (4GB) the read throughput is comparable for both CNTRFS and `ext4` filesystems because the data fits in the page cache. However, a larger workload (8GB) no longer fits into the page cache of CNTRFS and degrades the throughput significantly.

Postmark mailserver benchmark `Postmark` simulates a mail server that randomly reads, appends, creates or deletes small files. In this benchmark, we observe higher overhead (7.1×) for CNTR. In this case, inode lookups in CNTRFS dominates over the actual I/O because the files are deleted even before they were sync-ed to the disk.

PGBench – PostgreSQL Database Server `PGBench` is based on the `PostgreSQL` database server. It simulates both read and writes under normal database load. Like `FIO`, CNTRFS is faster in this benchmark also, since `PGBench` flushes the writeback buffer less often.

SQLite benchmark The `SQlite` benchmark measures the time needed to insert 1000 rows in a SQL table. We observe a reasonable overhead (1.9×) for CNTR, since each insertion is followed by a filesystem `sync`, which means that we cannot make efficient use of our disk cache.

Threaded I/O benchmark The `Threaded I/O` benchmark separately measures the throughput of multiple concurrent readers and writers to a 64MB file. We observe good performance for reads (1.1×) and even better performance for writes (0.3×). This is due to the fact that the reads can be mostly served from the page cache, and for the writes, our writeback buffer in the kernel holds the data longer than the underlying filesystem.

Linux Tarball workload The `Linux tarball` workload unpacks the kernel source code tree from a compressed tarball. This workload is similar to the `create` stage of the `compilebench` benchmark. However, since the source is read from a single tarball instead of copying an already unpacked directory, there are fewer lookups performed

in CNTRFS. Therefore, we incur relatively lower overhead (1.2×) even though many small files are created in the unpacking process.

3.5.2.3 Effectiveness of optimizations

We next evaluate the effectiveness of the proposed optimizations in CNTR (as described in §3.3.3).

Read cache The goal of this optimization is to allow the kernel to cache pages across multiple processes. Figure 3.4a (a) shows the effectiveness of the proposed optimization for `FOPEN_KEEP_CACHE`: we observe 10× higher throughput with `FOPEN_KEEP_CACHE` for concurrent reads with 4 threads for the `Threaded I/O` benchmark.

Writeback cache The writeback optimization is designed to reduce the amount of write requests by maintaining a kernel-based write cache. Figure 3.4b (b) shows the effectiveness of the optimization: CNTR can achieve 65% more write throughput with the writeback cache enabled compared to the native I/O performance for sequential writes for the `IOZone` benchmark.

Batching To improve the directory and inode lookups, CNTRFS batches requests to kernel by specifying the `PARALLEL_DIROPS` flag. We observe a speedup of 2.5× in the `compilebench` read benchmark with this optimization (Figure 3.4c (c)).

Splice read Instead of copying memory into userspace, we move the file content with the `splice()` syscall in the kernel to achieve zero-copy I/O. Unfortunately, we do not notice any significant performance improvement with the splice read optimization. For instance, the sequential read throughput in `IOZone` improved slightly by just 5% as shown in Figure 3.4c (d).

3.5.2.4 Extended effectiveness evaluation of optimizations

While the original evaluation of various optimisations that were part of the original publication of cntr only covered selected benchmarks, we extend this below by running the entire Phoronix suite for various optimisations.

Experimental testbed Since the original testbed (the m4.xlarge EC2 instance) is no longer available, we instead run the experiment on a machine with a AMD EPYC 7713P CPU with 64 cores (128 hyper-threads, 256 MB), 503.25 GiB DDR4 memory. All

disc benchmarks are run on a dedicated Dell EMC PowerEdge AGN MU AIC Gen4 NVMe 1.6TB drive. The host OS is Linux version 5.10.89. The used phoronix version version is newer than in the previous run (v10.8.1), which has a list of benchmarks compared to the old version in Section 3.5.2.

Methodology To make the experiment more comparable to the old run, we use cgroups and CPU tasksets to limit the resources available for the experiment to 16 GiB RAM and 4 cpus. For each run of the phoronix test suite, we discard all data on the NVMe disc with the SSD TRIM command.

As a baseline, we run Phoronix once with all optimisations and then 4 runs each with one optimisation disabled (Read cache, Batching, Splice Read and Writeback cache). Each configuration is run several times (the exact number of trials is chosen by phoronix itself based on the deviation) and a median is calculated from the result.

Summary of the results Figure 3.5 shows impact of different optimisations for all benchmarks in the `Phoronix` test suite. On average, the slowdown we see from disabled individual optimisations is 10%. The most effective optimisation is splice read, especially for read heave benchmarks, as it reduces data copies. This is followed by read and writeback caches. Batching, on the other hand, actually seems to lead to lower performance in some benchmarks. By allowing concurrent inode requests, it seems to lead to more expensive locking in other parts of the system.

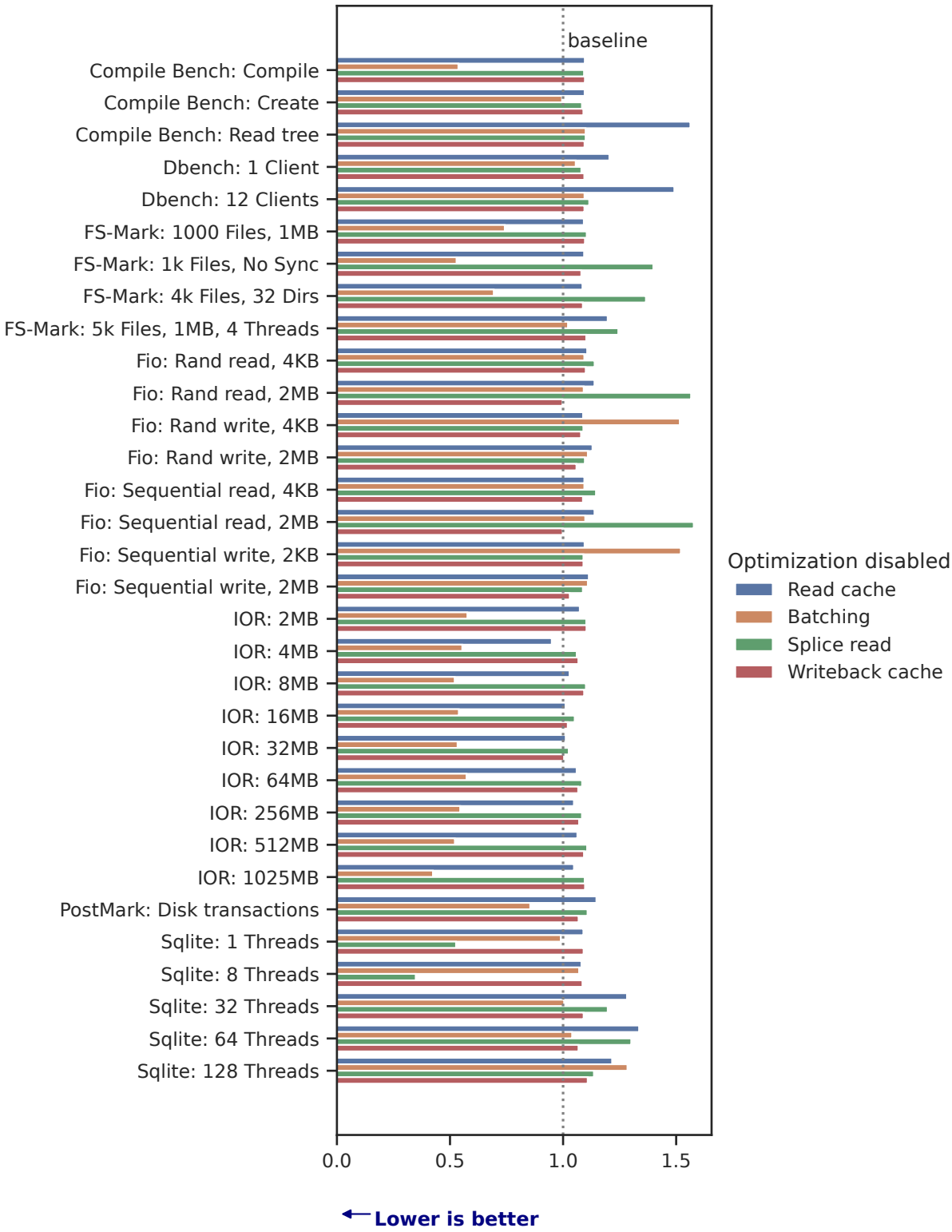


Figure 3.5: Phoronix benchmark with individual optimizations in CNTR disabled compared to all optimizations enabled, i.e. a ratio of 1.5 means that disabling this optimization slows down the benchmark by 0.5.

3.5.3 Effectiveness of CNTR

To evaluate the effectiveness of CNTR's approach to reducing the image sizes, we use a tool called Docker Slim [71].

Docker Slim applies static and dynamic analyses to build a smaller-sized container image that only contains the files that are actually required by the application. Under the hood, Docker Slim records all files that have been accessed during a container run efficiently using the `fanotify` kernel module.

For our analysis, we extend Docker Slim to support container links, which are extensively used for service discovery, and it is available as a fork [198].

Dataset: Docker Hub container images For our analysis, we choose the Top-50 popular official container images hosted on Docker Hub [70]. These images are maintained by Docker and contain commonly used applications such as web servers, databases and web applications. For each image, Docker provides different variants of Linux distributions as the base image. We use the `default` variant as specified by the developer.

Note that Docker Hub also hosts container images that are not meant to be used directly for deploying applications, but they are meant to be used as base images to build applications (such as language SDKs or Linux distributions). Since CNTR targets concrete containerized applications, we do not include such base images in our evaluation.

Methodology For our analysis, we instrument the Docker container with Docker Slim and manually run the application so it would load all the required files. Thereafter, we build new smaller containers using Docker Slim. These new smaller images are equivalent to containers that developers could have created when having access to CNTR. We envision the developers will be using a combination of CNTR and tools such as Docker Slim to create smaller container images. Lastly, we tested to validate that the smaller containers still provide the same functionality.

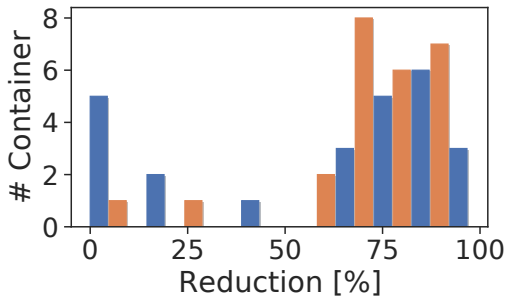


Figure 3.6: Reduction of container size after applying docker-slim on Top-50 Docker Hub images.

Experimental results On average, we could reduce the size by 66.6% for the Top-50 Docker images. Figure 3.6 depicts the histogram plot showcasing percentage of container size that could be removed in this process. For over 75% of all containers, the reduction in size was between 60% and 97%. Beside the applications, these containers are packaged with common used command line auxiliary tools, such as coreutils, shells, and package managers. For only 6 out of 50 (12%) containers, the reduction was below 10%. We inspect these 6 images and find out they contain only single executables written in Go and a few configuration files.

3.6 Related Work

In this section, we survey the related work in the space of lightweight virtualisation.

Lambda functions Since the introduction of AWS Lambda [12], all major cloud computing providers offer serverless computing, including Google Cloud Functions [98], Microsoft Azure Functions [29], IBM OpenWhisk [117]. Moreover, there exists a research implementation called Open Lambda [107]. In particular, serverless computing offers a small language runtime rather than the full-blown container image. Unfortunately, lambdas offer limited or no support for interactive debugging or profiling purposes [249] because the clients have no access to the lambda's container or container-management system. In contrast, the goal of the CNTR is to aim for lightweight containers, in the same spirit of lambda functions, but to also provide an on-demand mechanism for auxiliary tools for debugging, profiling, etc. As a future work, we plan to support auxiliary tools for lambda functions [4] using CNTR.

Microkernels The microkernel architecture [106, 26, 137] shares a lot of commonalities with the CNTR architecture, where the applications/services are horizontally partitioned, and the communication happens via the inter-process communication (IPC) mechanism. In CNTR, the application container obtains additional service by communicating with the “fat" container via IPC using CNTRFS.

Containers Recently, there has been a lot of interest in reducing the size of containers, but still allowing access to the rich set of auxiliary tools. For instance, Toolbox [255] in CoreOS [57] allows to bind the mount of the host filesystem in a container to administrate or debug the host system with installing the tools inside the container. In contrast to Toolbox, CNTR allows bidirectional access with the debug container. Likewise, `nsenter` [191] allows entering into existing container namespaces, and spawning a process into a new set of namespaces. However, `nsenter` only covers namespaces, and it does not provide the rich set of filesystem APIs as provided by CNTR. Lastly, Slacker [105] proposed an opportunistic model to pull images from registries to reduce the startup times. In particular, Slacker can skip downloading files that are never requested by the filesystem. Interestingly, one could also use Slacker to add auxiliary tools such as gdb to the container in an “on-demand" fashion. However, Slacker could support additional auxiliary tools to a container, but these tools would be only downloaded to the container host, if the container is started by the user. Furthermore, Slacker also has a longer build time and greater storage requirements in the registry. In contrast, CNTR offers a generic lightweight model for the additional auxiliary tools.

Virtual machines Virtual machines [154, 32, 63] provide stronger isolation compared to containers by running applications and the OS as a single unit. On the downside, full-fledged VMs are not scalable and resource-efficient [236]. To strike a balance between the advantages of containers and virtual machines, Intel Clear Containers (or Kata Containers) [119] and SCONE [25] offer stronger security properties for containers by leveraging Intel VT and Intel SGX, respectively. Likewise, LightVM [172] uses unikernel and optimized Xen to offer lightweight VMs. In a similar vein, CNTR allows creating lightweight containers, which are extensively used in the data centre environment.

Unikernels and Library OSes Unikernels [157, 155] leverage library OSes [230, 36, 259, 35] to selectively include only those OS components required to make an application work in a single address space. Unikernels use a fraction of the resources required compared to full, multipurpose operating systems. However, Unikernels

also face a similar challenge as containers — If Unikernels need additional auxiliary tools, they must be statically linked in the final image as part of the library OS. Moreover, unikernel approach is orthogonal since it targets the kernel overhead, whereas CNTR targets the tools overhead.

3.7 Limitations and future work

While CNTR can attach to container workloads it cannot attach to virtual machines, since it relies on the host os to spawn processes. In chapter 4 we also address virtual machine support, which enables a range of interesting use cases (see § 4.6.4).

CNTR needs to run on the same host that runs the container. It does not integrate itself into cluster container orchestrators like Kubernetes [143] or Nomad [190] that provision containers on multiple hosts. Cluster container orchestrators are widely in use in production. Hence, it would be interesting to extend CNTR to interact with their API to locate the host where the application container has been deployed to and attach a debug container on the same host.

Another interesting target would be Function-as-a-Service (FaaS) platforms. In FaaS platforms, applications are also distributed over multiple hosts and scaled up on demand based on external events (i.e. an incoming HTTP request). Many of the open-source implementations are also based on kubernetes [139, 195, 142]. To debug the application in case of errors the current practice is to re-deploy the application. Running the application in a local environment is often not possible because the application might rely on a provider-specific API. CNTR would be useful here to extend the minimal FaaS application environments with an interactive shell with debug tools available. For a virtual-machine based Function-as-a-Service platform we build a similar tool in § 4.6.4.

3.8 Summary

In this chapter we presented CNTR, a system for building and deploying lightweight OS containers. CNTR partitions existing containers into two parts: “slim" (application) and “fat" (additional tools). CNTR efficiently enables the application container to dynamically expand with additional tools in an on-demand fashion at runtime. Further, CNTR enables a set of new development workflows with containers for improved productivity:

- When testing the configuration changes, instead of rebuilding containers from scratch, the developers can use their favorite editor to edit files in place and

reload the service.

- Debugging tools no longer have to be manually installed in the application container, but can be placed in separate debug images for debugging or profiling in production.
- To securely run an application inside a container, one can enforce stricter security policies (`AppArmor/SELinux`), and run CNTR with relaxed security rules.

To the best of our knowledge, CNTR is the first generic and complete system that allows attaching to containers and inheriting all their sandbox properties. We use CNTR to debug existing container engines [224]. In our evaluation, we test extensively the completeness, performance, and effectiveness properties of CNTR.

Source code availability We make CNTR with the complete experimental setup publicly available for the research community [56].

While CNTR makes containers more maintainable with its functionality, it does not improve their security, which is what we address in the next chapter with RKT-IO.

Chapter 4

>_ VMSH : Hypervisor-agnostic Guest Overlays for VMs

Lightweight virtual machines (VMs) are prominently adopted for improved performance and dependability in cloud environments. To reduce boot-up times and resource utilisation, they are usually "pre-baked" with only the minimal kernel and userland strictly required to run an application. This introduces a fundamental trade-off between the advantages of lightweight VMs and available services within a VM, usually leaning towards the former. We propose VMSH, a hypervisor-agnostic abstraction that enables on-demand attachment of services to a running VM—allowing developers to provide minimal, lightweight images without compromising their functionality. The additional applications are made available to the guest via a file system image. To ensure that the newly added services do not affect the original applications in the VM, VMSH uses lightweight isolation mechanisms based on containers. We evaluate VMSH on multiple KVM-based hypervisors and Linux LTS kernels and show that: *(i)* VMSH adds no overhead for the applications running in the VM, *(ii)* de-bloating images from the Docker registry can save up to 60% of their size on average, and *(iii)* VMSH enables cloud providers to offer services to customers, such as recovery shells, without interfering with their VM's execution.

4.1 Introduction

Virtualisation is the cornerstone of cloud computing. Cloud providers predominately use virtual machines (VMs) to consolidate and isolate multiple tenants on a single physical host [279, 2]. To enable virtualisation, the Linux kernel-based virtual machine (KVM) [160] is the de facto mechanism in the cloud since it uses hardware acceleration

to enforce compartmentalisation [115, 239, 97, 196].

With an increased demand to support performance-critical workloads, there is a significant thrust towards designing lightweight VM solutions to minimise the virtualisation overheads [2, 172, 211, 54]. These solutions provide reduced memory footprints and fast boot up times [182], which makes them suitable for increasingly popular deployment models, such as serverless [264, 31]. Furthermore, lightweight VMs improve dependability properties since they strive to minimise the trusted and reliable computing base [172].

The key to build lightweight VMs is to minimise their root image size. This entails removing additional services, such as monitoring and inspection tools, which are not used in normal application deployments. Therefore, the VM images strive for reduced software dependencies; thus, enabling agile development. While lightweight VMs provide a promising approach for modern cloud workloads, they are limiting in other crucial scenarios at the same time. In particular, the deployed file system images are typically pre-built and must be re-deployed for every change—even during testing. This limitation is especially amplified when the users need additional tools or services on-demand that are initially not a part of the lighter VMs. Re-building images can be particularly bothersome when development tools are missing for debugging, monitoring or repairing VMs. The following re-deployment requires complex interplay between many cloud components and configurations. And finally, it always means that the virtualised system is restarted and all measurable or debuggable state is lost.

This fundamental trade-off between the advantages of lightweight VMs and available services within a VM manifests in the form of restricted functionalities provided by VMs. On the one hand, the users want pre-baked lightweight VMs for performance. However, adding more software in a non-disruptive way is difficult because of the variety of lightweight VM stacks. To work around these limitations, the cloud providers offer a plethora of purpose-built and highly specialised management agents [11, 94, 93, 180] and tracing libraries [8] which again counteract advantages of lightweight VMs such as their dependability properties (§ 4.7).

To this end, we ask the following research question: *Can lightweight VMs be extended with external functionality on-demand and non-disruptively?* To address this problem, we propose VMSH, which provides an abstraction for accessing KVM based VMs for tasks such as inspection, debugging, or modification. VMSH enables users to add functionality to VMs non-disruptively and connect to newly attached programs via a console. Software packages added to lightweight VMs with VMSH do not require modifications. Moreover, the original guest userspace is protected from accidental

harm. To maintain generality, VMSH provides an abstraction over the hardware and APIs of different hypervisor implementations, to offer a uniform hardware interface.

VMSH achieves this by side-loading kernel code from the hypervisor into the guest. This code registers hypervisor-independent block and console devices. It then spawns a container-based system overlay that mounts the file system from the block device, which contains the service to be started in the container. The user can interact with the injected service over the console device and work with the original guest outside of the guest overlay. To protect the guest from accidental harm, VMSH is container aware and mounts namespaces selectively.

Our implementation currently targets KVM-based hypervisors with Linux kernels both in the host and the guest. To side-load external code in the VM, VMSH operates directly on the hypervisor's KVM and conducts a binary analysis on the VM's memory to load a kernel library into the guest. For VMSH to serve a file system image to start programs from, we implement a block device following the VirtIO standard.

We evaluate VMSH across four dimensions: robustness, generality and performance. Lastly, we evaluate three use-cases. For robustness, we run the `xfstests` [278] suite and show that VMSH's block device does not have any regressions compared to the QEMU implementation (§ 4.6.1). We show VMSH's generality by successfully testing 4 industry leading KVM based hypervisors and all current long-term support versions of the Linux kernel (§ 4.6.2). We measure performance with the Phoronix Test Suite [203] and `fio` [126], and find no slowdown of the guest while VMSH is attached (§ 4.6.3). Finally, we implement three real-world use-cases that make VMSH a key element in cloud infrastructures (§ 4.6.4).
Our contributions can be summarised as follows:

- We propose an abstraction which allows extending lightweight VMs at run time independently of the guest and hypervisor (§ 4.2). This enables lighter VMs by removing tools from VM images while still being able to attach them back to the VM on-demand (§ 4.2.1).

- We design a system for hypervisor-independent side-loading into a VM of a generic guest-overlay that does not impose limitations on both the original guest application or the spawned service, and a device that can be attached to hypervisors non-cooperatively (§ 4.4).

- We implement (§ 4.5) and evaluate (§ 4.6) VMSH. We show that it is compatible across many hypervisors and Linux versions, that it does not slow down the original VM guest, and that its use-cases have the potential to reduce image sizes of lightweight VMs.

4.2 Motivation

Attaching programs or services at run time to today's VMs is a complex task since accessibility is provided by services like SSH, requiring key management and configuration. New applications have to be integrated into the file system, which typically requires compatibility with a given package manager. On serverless platforms, that often means redeploying the whole application, which is disruptive and might mask the error's origin due to the loss of the VM state. Moreover, the lack of a consistent hypervisor management API is a hindrance for adding virtual devices at run time.

We therefore need an abstraction that reduces this complexity down to a universal and simple interface that is used to execute arbitrary applications on-demand inside VMs. Container runtimes offer a similar user experience with container-exec tools like *docker exec*. VMSH aims to satisfy this requirement for VMs too, and aims to work with many state-of-the-art hypervisors and Linux kernel versions.

Using our new abstractions, we show multiple use cases that target different application scenarios, that we hope can empower cloud providers and application developers alike. In the long run, we also hope that we can motivate new virtualisation standards which improve performance and long-term stability compared to VMSH. We envision a *vm-exec* device that allows one to start binaries, while not depending on vendor-specific guest agents.

4.2.1 Example use-cases enabled by VMSH

Given the *vm-exec* device abstraction, we can enable a range of new services that help administrators and developers to operate or run VM workloads (also see § 4.6.4)

Dependability services Cloud customers tend to have a wide range of distributions and versions installed [17]. Therefore, integrating provider tools into guests can be challenging. VMSH makes it possible to decouple these services from the guest userland. For example, one could implement the following services using VMSH:

- *Rescue systems* in case of misconfiguration, including network misconfiguration or forgotten passwords. Existing implementations of such services require rebooting into a recovery system [109, 67].
- *Monitoring tools* are currently used to gather coarse-grained information about the resource usage of the entire guest [161]. VMSH provides a more fine-grained view as it gives access to the guest OS metadata, such as the process list, resource usage, etc.

- *Security scanner* tools that track out-of-date or insecure packages. This is already done in the container space [92, 9, 116]. VMSH enables similar techniques to track and update packages in the VM space.

De-bloat VM images VMSH allows building lightweight VMs [32, 216, 269, 267] by omitting debugging and administration tools from main applications deployed in a VM. Such an approach reduces the size of deployed images, providing multiple advantages. First, the cost of storage is reduced. Second, smaller image sizes lead to faster scale-up times as the amount of data transferred over the network is low. Finally, the build time required to generate the images is also reduced. Moreover, on-demand debugging environments can be packed with more tools compared to the current installations that only contain tools that are required by the administrators or developers to log into the VM. This improves the security of running the VM, as services such as SSH are no longer required.

Serverless frameworks Serverless offerings usually run in lightweight VMs to improve isolation between instances. Developers usually do not have access to the environment running the instances. Additionally, these images often contain only a minimal management layer from the service provider, and the main application that the developer wants to deploy. For error and performance debugging, the user has access to minimal resource metrics exposed by the provider [10] and logging information from the application itself [6]. With VMSH, users could gain access to these serverless instances, e.g. by integrating VMSH into a Web-IDE and perform interactive debugging. This would allow for more time-efficient debug cycles compared to having to re-deploy the application on every modification.

4.3 Overview

4.3.1 System overview

To realise the *vm-exec* abstraction for VMs (Section 4.2), we design VMSH, a system that allows users to extend VMs at run time. A dedicated file system image provides the additional tools and services that execute transparently, without any help from a guest agent, the hypervisor or the guest OS.

As shown in Figure 4.1, VMSH runs natively on the host, in parallel to the hypervisor process. VMSH attaches to the hypervisor, and spawns a container-based overlay running on top of the guest kernel. From the supplied file system image, this overlay can start applications, connecting them to VMSH's console. These

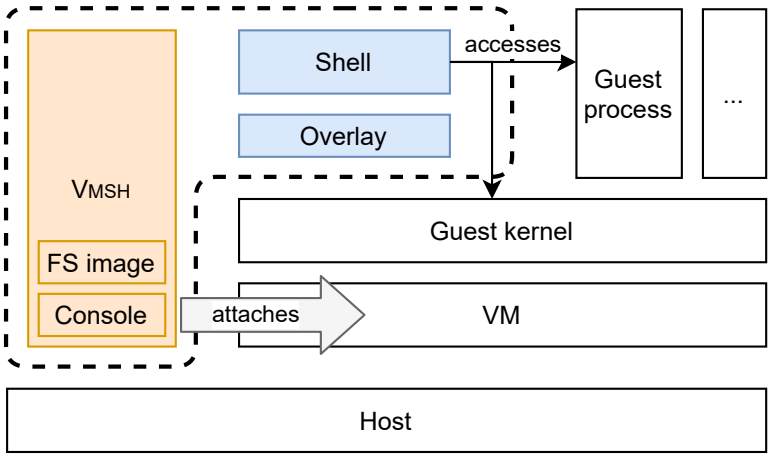


Figure 4.1: A user attaches a custom file system image to a VM and starts a shell from the image using VMSH. (*Orange refers to* VMSH *components running on the host and blue to the ones in the guest.*)

applications, e.g., a shell, run in guest userspace. To this end, VMSH strives for the following design goals:

- *Non-cooperativeness*: VMSH must not rely on agents in guest userspace.
- *Generality*: VMSH shall be agnostic to the underlying hypervisors and should not depend on hypervisor-specific APIs. Also, it shall support a wide range of different guest kernel versions.
- *Performance*: We aim to have no degradation in performance of applications running in a guest where VMSH is attached. Performance of the attached tools and services is secondary, but they need to be usable.

Figure 4.2 shows how VMSH attaches to a VM and spawns tools and services to interact with the applications and kernel inside the VM. In step ①, VMSH attaches its console and block device to the hypervisor to serve the user supplied file system image. In step ②, a library is side-loaded into the guest kernel. The library starts the guest drivers to make VMSH's console and the file system image available to the guest kernel. In step ③, the library spawns a process that creates the guest overlay container. The file system image is mounted as the overlay's root file system and existing guest mountpoints are made available under the directory */var/lib/*VMSH. In step ④, the spawned process starts tools or services, from the mounted file system image and redirects its input/output to the VMSH's console device.

4.3.2 Threat model

In a typical cloud deployment scenario we consider for VMSH, VMs are used to multiplex hardware resources on a single physical machine among multiple untrusted tenants. Through hardware-assisted virtualisation, the VMs are isolated from each other; thereby protecting their confidentiality, integrity and availability. Hence, we assume that the hardware, host OS and the hypervisor is included in VMSH's trusted computation base (TCB). While attacks on this TCB have been successful [209], they are out of scope for VMSH.

Attackers may compromise a VM in multiple ways. To escape a VM, they can attempt to exploit vulnerabilities in the hypervisor [219], as they contain complex device implementations that contribute to a relatively large attack surface. In Section 4.4.5, we describe the design choices we take as countermeasures to reduce the risk of such an attack. Previous work on hardening the security of KVM [20] and of the hypervisor [150] is orthogonal to our contributions with VMSH.

Exploiting VMSH to gain access to a VM is an another attack vector and requires another successful exploit. VMSH drops all privileges beyond the ones of the hypervisor after the setup phase for security hardening (see § 4.4.5).

Other attack vectors to gain access to the VM are possible, e.g., due to misconfiguration errors, but are not under our control. It is the responsibility of the host provider to ensure that there are no configuration/provisioning errors. For example, in VMSH's scenario, the host providers are the ones making VMSH available to their customer, *i.e.,* the VM owner. Therefore, they must enforce policies to allow the attachment of VMSH only by a set of authorised customers.

Related research, motivated by active IT security inspection of VMs, focuses on the stealthiness and integrity of injected code execution [270, 47]. In our scenario, the VMSH user and the VM owner are the same entity and trust the guest. This assumption makes intrusion detection with VMSH unreliable, but enables other hardware intensive workloads as shown in our evaluation (see § 4.6).

4.3.3 Design challenges

Next, we present the three challenges that we address when designing the *vm-exec* abstraction.

#1 Side-loading code into guest VMs As described in § 4.3.1, VMSH works by side-loading a library into the guest kernel, which then mounts the file system image with the required applications. Side-loading code into the guest VM would traditionally

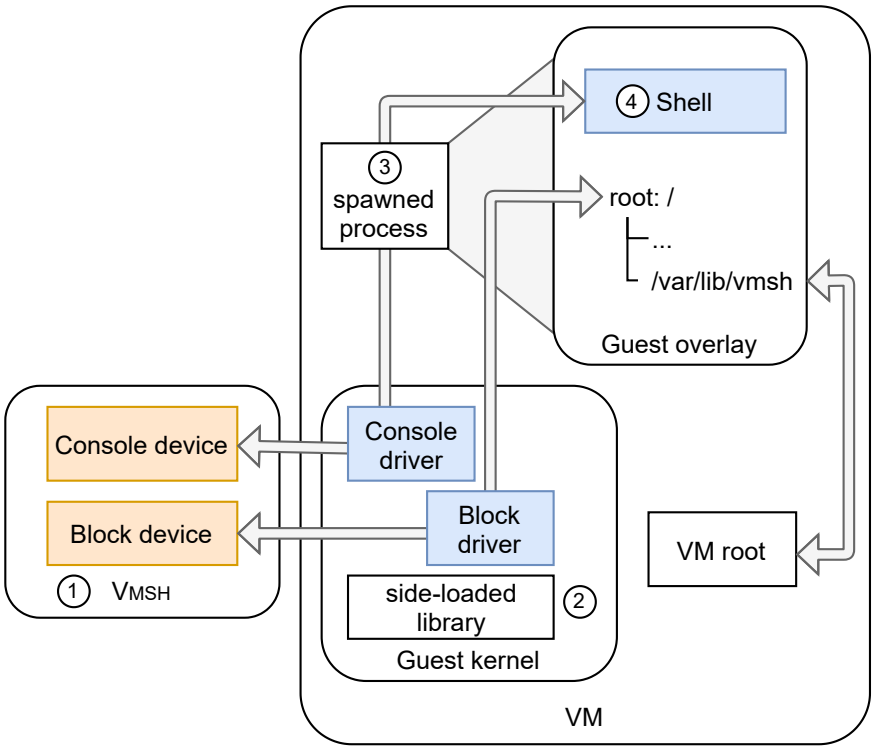


Figure 4.2: VMSH sets up its devices in the guest by side-loading a kernel library. The virtual block device backs the overlay's root. The virtual console handles console inputs/outputs of the spawned process.

require a cooperative guest agent running inside the VM or hypervisor-specific APIs that enables one to do so.

The increasing variety of new, lightweight hypervisors lack common APIs. QEMU provides a debugger interface that can be used for code side-loading, while also allowing one to attach disks at run time. Crosvm [95] only has the former whereas Firecracker [2] and kvmtool [275] lack both. In other cases, such features, even when supported by hypervisors, are obscured by orchestration frameworks, such as OpenStack [87] or Containerd [86].

APIs for interacting with hypervisors are therefore sparse, heterogeneous and incomplete. Consequently, side-loading code into the guest is challenging for VMSH since it aims to be hypervisor agnostic and not require guest agents. To overcome this, we design VMSH to access the underlying KVM API (see § 2.3.1) without any help from the hypervisor (see § 4.4.1).

#2 Building a side-loadable library VMSH aims to ensure that the side-loaded kernel library integrates with a wide range of kernel versions and without a guest agent. Therefore, VMSH has to find kernel function addresses which the library needs

and calls at run time. Finding those functions through binary analysis is difficult, especially with the Linux kernel as the internal kernel API and data structures are not considered stable. Hence, it is not trivial to build a side-loadable library that would work for all kernel versions. We need to strike a balance between the number of kernel features needed by VMSH and the functions it interacts with that could possibly change across kernel versions.

To address this issue, we build a minimal kernel library by offloading as much functionality as possible to existing kernel drivers (see § 4.4.2).

#3 Communication over VirtIO devices From an end-user perspective, one should be able to run any application, by attaching to the VM, and access application's input and output. However, there is currently no easy and transparent way in which we can make additional application files available to the guest at run time and redirect their IO to the host.

Therefore, we build a block and a console device that enable us to overcome these issues. Hypervisors such as QEMU and Firecracker emulate devices within the hypervisor itself. Since we aim to be hypervisor agnostic, the devices have to run outside the hypervisor process, *without* its cooperation. This requires us to overcome two challenges:

1. VMSH needs to handle MMIO-triggered `VMEXIT`s in the hypervisor which are caused by the guest accessing MMIO addresses of the devices.

2. Data to be exchanged between the guest driver and the VMSH device needs to be written to queues located in virtual guest memory and shared with VMSH.

To (1.) intercept MMIO accesses, VMSH uses one of two methods: a slower debugger-based approach and a novel KVM feature called ioregionfd [244]. The (2.) queues themselves are read from the hypervisor memory via system calls. We describe the design of our hypervisor-independent VirtIO devices in § 4.4.3.

4.4 Design

To address the design challenges, we next describe how we load kernel code into the guest VM (§ 4.4.1) and techniques to analyse the guest memory to enable VMSH to load the kernel library (§ 4.4.2). We describe mechanisms to serve VirtIO devices (§ 4.4.3). Then, we explain the layout of our container-based system overlay (§ 4.4.4), and finally discuss the security implications (§ 4.4.5).

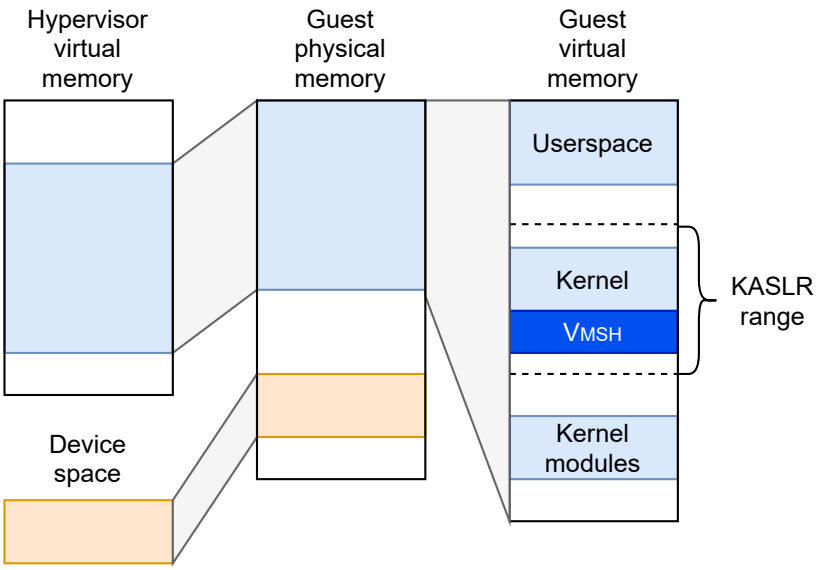


Figure 4.3: Address space mappings between hypervisor virtual memory and guest physical/virtual memory.

4.4.1 Hypervisor-agnostic side-loading for VMs

As described in § 4.3, it is the responsibility of the side-loaded kernel library to mount devices and spawn the userspace process that creates the guest overlay container. However, this has to occur without any help from a guest agent or the hypervisor. Hence, to address challenge #1, we present the design of VMSH's framework that enables side-loading code into a guest VM in a hypervisor agnostic manner.

To side-load arbitrary applications into the guest, VMSH first side-loads a kernel library into the guest to mount devices and spawn userspace guest processes. This can be done by loading the library into guest physical memory. The hypervisor has the guest physical memory mapped into its own address space (see Figure 4.3). VMSH can use this fact to find the location of the guest physical memory and side-load code. However, we cannot rely on hypervisor-specific APIs to perform this operation in a hypervisor-agnostic way. VMSH circumvents this limitation by injecting system calls into the hypervisor process. This is required as the OS only allows to manipulate the guest from the hypervisor process.

To be able to run system calls in the hypervisor process, we rely on debugging APIs provided by process tracers such as `ptrace`. This allows VMSH to control and inspect the state of the hypervisor process, and consequently the guest VM. It does so by interacting directly with the low-level kernel API, KVM in our case. Using this, VMSH first interrupts the hypervisor process. Next, VMSH prepares the system call arguments by updating the CPU registers according to the CPU-specific system call

ABI. When system calls require pointers to memory, VMSH allocates and maps the allocated memory into the hypervisor address space, and performs reads and writes to that memory region via inter-process memory access system calls. We describe this in § 4.5, specifically for the KVM API.

Using the low-level hypervisor API, *i.e.* KVM, VMSH queries the vCPUs of VM. It then dumps the register state of a vCPU to reveal the location of the page table, *i.e.* CR3 register on x86 and TTBR0 on arm64, which provides information about virtual memory mappings of the guest VM.

Using this information, VMSH side-loads the kernel library into the guest VM. It then uses the low-level hypervisor API to update the guest instruction pointer register to run from the library's code. However, modern operating systems employ hardening techniques such as Kernel Address Space Layout Randomisation (KASLR), that maps the kernel into random locations in virtual memory on every boot. In the next section, we describe the binary analysis techniques used by VMSH to recover random location of the kernel and its functions in memory.

4.4.2 Kernel-agnostic library

As previously stated, VMSH side-loads a kernel library into the guest that enables mounting devices and spawning a guest userspace process. This library is not a Linux kernel module as we do not use Linux's load mechanism (also see § 4.5). Because of KASLR, mapping the kernel library into the correct location is challenging. Hence, to address challenge #2, we present the design of VMSH's binary analysis framework that provides VMSH with information about the location of the kernel and relevant kernel function addresses that are used within the side-loaded kernel library.

Although KASLR randomizes the kernel location, the kernel itself is placed into a fixed number of slots in memory, located in a fixed address range [125]. VMSH can therefore locate the kernel by iterating over the guest VM's page table entries.

VMSH also searches for the location of the function name section in the guest OS, e.g. located at *.ksymtab_strings* in Linux (other OSes provide similar mechanisms [141]). The actual function addresses are stored in a different data structure (*.ksymtab*), whose size is unknown. Since this data structure contains references to the function name section, VMSH checks for valid references to estimate its size. VMSH then uses the data structure to figure out the addresses of all exported kernel functions in memory. These addresses are used by VMSH to fix up kernel function references in the library being side-loaded into the guest via VMSH's custom binary loader.

With the kernel function references resolved, VMSH uses the discovered kernel

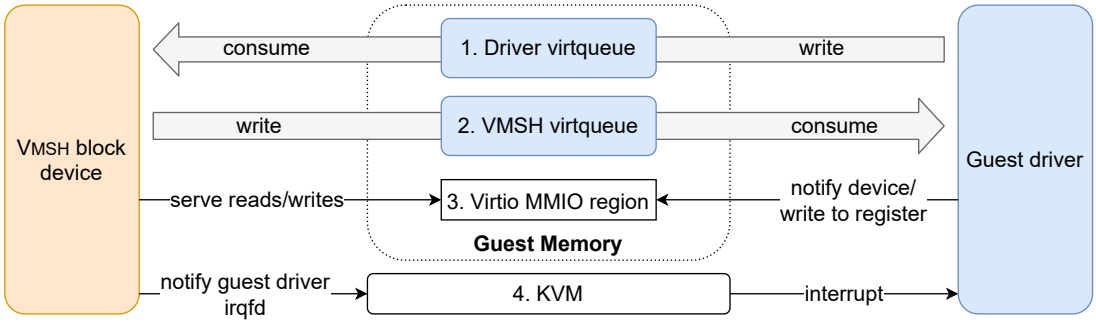


Figure 4.4: VMSH communication infrastructure based on the VirtIO protocol. Guest and host components share data through virtqueues (1, 2). Notification is performed through MMIO regions (3) and KVM (4).

address range to side-load the kernel library into the guest, by writing it into hypervisor memory. To load it in such a manner that there are no collisions with existing guest physical allocations set up by the hypervisor, VMSH allocates new guest physical memory at the upper end of the guest address space. Hypervisors often advertise the same CPU model across different physical machines to allow VM migration, which results in physical address sizes in the guest being larger than on the actual hardware. Since guests do not crash despite this, we conclude that the upper end of memory is not used in practice.

With the kernel library side-loaded into guest physical memory, it needs to be mapped into guest virtual memory, so that it can be run from within the guest VM. The library is mapped into the guest virtual memory by updating the guest's page tables. Once again, we take advantage of the fact that the KASLR range is known, as described previously. Moreover, once the kernel is loaded at boot time into a random location in memory, no more changes are made afterwards. Hence, it is safe to map the side-loaded library in virtual memory right after the kernel, as shown in Figure 4.3.

Once the library is loaded into the guest VM and can be executed, VMSH modifies the instruction pointer of the guest VM's vCPU, via the low-level hypervisor API, to run the library's code. To synchronise events between VMSH running on the host and the side-loaded library running in the guest, we use a shared memory region that the guest polls for updates from VMSH and vice versa.

4.4.3 Hypervisor-independent VirtIO devices

The side-loaded kernel library is used to register VMSH's VirtIO devices. These devices need to be run in a hypervisor agnostic manner, and must therefore run in a process external to the hypervisor. Hence, to address challenge #3, we design VirtIO

block and console devices that run inside the VMSH process. VMSH uses the block device to serve the file system image containing applications and the console device to redirect the application's input and output outside the guest VM.

VMSH uses the VirtIO protocol to serve both types of devices. In the following, we explain the general flow for the block device driver. The guest driver enqueues block IO requests into its virtqueue for the VMSH block device to consume (Fig. 4.4/1.). VMSH's block device processes the request and enqueues the response into the other virtqueue (Fig. 4.4/2.). To indicate new requests in the queue, the guest driver also notifies the block device by writing to an MMIO register (Fig. 4.4/3.). As the corresponding MMIO addresses are not backed by physical memory, writing to them causes a `VMEXIT`. Since VMSH's devices run in a process external to the hypervisor, we need to trap such accesses and handle them in VMSH's respective device. In § 4.5, we describe the two ways in which we can trap and handle MMIO accesses to VMSH's devices from the guest. To notify the guest driver about new items in VMSH's virtqueue, we trigger an interrupt through KVM using an irqfd (Fig. 4.4/4.).

4.4.4 Container-based system overlay

After setting up the devices, the kernel library spawns a userspace process in the guest (see Figure 4.2). However, the spawned process and additional devices may require an environment that would conflict with the guest VM's root file system, e.g. configuration files in */etc*. Such conflicts can be avoided by using containerisation techniques.

These conflicts arise when applications rely on absolute paths to files existing on both file systems. To resolve possible conflicts, VMSH employs mount namespaces. The file system on the block device provided by VMSH is mounted as the root file system in a newly created mount namespace. All old mount points of the guest are moved under the directory (*/var/lib/*VMSH). Using a mount namespace ensures that these mount points are not propagated to existing guest processes except the ones started by VMSH.

Additionally, VMSH can attach to containers running inside VMs, which is becoming the standard method to run container workloads due to improved security benefits. VMSH is not tied to a specific container engine, e.g., Docker, lxc, containerd. Instead, it uses the process ID of a containerised process running inside the VM to get information about the process (UID, GID, Apparmor/Selinux profiles, namespaces, cgroups, capabilities) and applies the context to the newly established interactive shell.

4.4.5 Security

In this section, we discuss the design decisions to minimise VMSH's impact on security, as it increases the attack surface by adding more functionality to the hypervisor. As explained in § 4.3.2, our main threat is from an attacker who controls the VM, and could exploit VMSH to escape from the VM and get access to the host.

Firstly, the side-loaded library in the guest kernel and the application runs in the same privilege domain as the guest. Hence, it does not impact the attacker's capability because they could run similar code without VMSH.

Secondly, the largest part of the attack surface is contributed by VMSH's devices which run on the host. VMSH is written in Rust to further improve memory safety by the use of safe abstractions. VMSH relies on production-tested libraries that are also used by Firecracker, crosvm and Cloud Hypervisor. We expect that a bug in these libraries would also affect those hypervisors.

Thirdly, to find the physical memory inside the hypervisor address space, we use an eBPF program. Therefore, VMSH currently needs privileges beyond those of an unprivileged user. In our prototype, those capabilities are dropped before interacting with the guest/hypervisor to not increase the privileges exposed to a potential attacker. In the future, we plan to move this part into a dedicated setuid binary to improve security.

In comparison to guest agents installed by some VM providers, VMSH shifts the responsibility of secure authentication and authorisation from the customer to the provider. But we believe that, comparatively, VMSH does not increase the TCB (Trusted Computing Base), because we do not require network access from the guest network, which mitigates remote code execution bugs [179].

4.5 Implementation

VMSH is written in Rust (13k LoC), except for a small trampoline code written in assembly, used in our kernel library entrypoint. VMSH consists of three programs: the host executable with VirtIO devices, a guest kernel library and a guest userspace program. The host executable contains both the sideloader that uploads the code into the guest kernel as well as the VirtIO device implementation. For ease of deployment, we build VMSH as a single binary, with the guest kernel library and guest userspace program embedded in its data section.

Sideloader The sideloader is responsible for uploading our guest kernel code into the guest. In our implementation, we target the KVM API instead of relying on a

particular KVM userland hypervisor. To figure out the number of vCPUs, we use Linux's */proc* file system to iterate over the hypervisor's file descriptors and identify those that belong to KVM by resolving symbolic links in */proc*. The sideloader then uses the `ptrace` system call to interrupt the hypervisor process with `PTRACE_INTERRUPT` to perform the system call injection described in § 4.4.1. VMSH uses the `process_vm_readv()` and `process_vm_writev()` system calls to read from and write to the hypervisor memory, respectively. Some system calls, e.g. the KVM irqfd system call, return a file descriptor that the sideloader sends back to its respective host process using an injected UNIX socket.

Prior to uploading, the sideloader needs to locate the guest memory in the hypervisor memory. Since there exists no KVM API to figure out the physical memory layout of the VM and its corresponding mappings in the hypervisor virtual address space, we extract this information from the host kernel data structures using a small eBPF program we attach to the KVM function `kvm_vm_ioctl()`. This function is called by the host kernel when a KVM system call is injected. Our eBPF program parses the data structure containing all guest allocations and their offsets in the hypervisor memory from the function's arguments.

VirtIO devices VirtIO devices run as background threads in VMSH. We implement the devices by using existing Rust libraries from the rust-vmm [169] project. These libraries are also used in Firecracker, crosvm and Cloud Hypervisor. We extend their backend to read from and write to another process' memory as described in § 4.4.3. We optimise the performance by mapping the block device as a file into memory and use the `process_vm_readv()/process_vm_writev()` system calls to copy data between the hypervisor process and the block device file, directly in the host kernel. This doubles the performance in Phoronix benchmarks as shown in § 4.6.3.

As outlined in § 4.4.3, VMSH traps accesses to MMIO addresses for device initialisation and driver updates (also see Figure 4.4/3.). In VMSH, we either rely on a `ptrace`-based solution or KVM's ioregionfd as described next.

Ptrace To start executing a vCPU, the hypervisor uses the `ioctl(KVM_RUN)` system call and blocks, waiting to be woken up by returning from the system call. Inside the guest, when an MMIO access occurs, a `VMEXIT` is triggered, unblocking the hypervisor process. We use `ptrace` to hook into this system call's entry and exit, effectively allowing us to create a wrapper around it. The hypervisor thread running the respective vCPU will be interrupted each time, until we resume it. During this period, we use the memory mapped vCPU file descriptor of KVM to parse the MMIO

request and handle it.

Ioregionfd Using `ptrace` adds an overhead to all `VMEXIT`s, as we add context switches to the VMSH process. This can hurt the performance of the guest application. Therefore, we offer support for KVM ioregionfd, a feature currently under review for inclusion into the Linux kernel [244], as an alternative to wrap_syscall. This feature allows an MMIO region to be associated with a file descriptor, that can in turn be used to notify the VMSH process. It uses sockets to send MMIO accesses to the device that handles them.

Guest kernel library We build this component as a shared ELF library. The entry point to the library uses a trampoline that saves and restores registers. This allows VMSH to ensure the guest jumps to the library rather than having to call it. Most common OSes do not provide a stable ABI. Hence, the kernel interface that our library uses should be minimal to avoid possible breaking changes between different kernel versions and maximise code reuse. In our prototype, we target the Linux kernel and also test portability across different kernel versions in § 4.6.2. In total, we use twelve kernel functions (two for driver registration, four related to file IO, five related to process/threads).

Guest userspace program To keep the kernel library small, we offload as much functionality as possible to the guest userspace. The guest program is a statically linked executable that is copied into the guest VM by the kernel library into a writable path, *i.e. /dev*. We choose the devtmpfs file system as the default option because it is writable even on systems that have a read-only root file system (a common practice for serverless VM images). Once started, the guest program will then set up the container-based system overlay that is described in § 4.4.4.

Implementation status VMSH currently targets the Linux kernel. The VirtIO devices are standardised and portable to other OS guests. However, the side-loaded kernel driver and userland code need to be adapted to other operating systems. We do not see this as a major limitation given that Linux dominates the public cloud market share [59]. Due to the low-level nature of the project, we only support the x86_64 architecture. We have plans to port our system to arm64. An architecture port would require to extend the system call injection, as well as register and page table handling.

4.6 Evaluation

We evaluate VMSH across the following dimensions: robustness (§ 4.6.1), generality (§ 4.6.2) and performance (§ 4.6.3), Lastly, we evaluate three use-cases (§ 4.6.4).

Experiment setup We perform our experiments on a machine with an Intel Core i9-9900K CPU with 8 cores (16 hyper-threads, 16 MiB L3 cache), 64 GiB of DDR4 memory. All disk benchmarks are run on a dedicated Intel P4600 NVMe 2TB drive. The host OS is Linux version 5.12.14. For performance related benchmarks, we use QEMU with KVM as the hypervisor and start the VM with 8 GiB of RAM and 4 vCPUs. For better reproducibility, we pin the hypervisor vCPUs and disable Intel Turbo boost. Before each IO related benchmark, we discard all data with the SSD TRIM command.

4.6.1 Robustness

We evaluate the robustness of VMSH, and more precisely the VMSH block device, VMSH-BLK, to ensure completeness and correctness according to the POSIX standard.

Benchmark We use `xfstests`, a test suite widely adopted by the kernel community for fuzzing and regression testing of file systems [278] and block devices [135]. `xfstests` contains tests suites to ensure *correctness* and *completeness* of all file system related system calls and their edge cases, including crashes and reported bugs.

Methodology We select the “quick" test group which contains the majority of tests. We run those tests by provisioning a physical block device with two XFS partitions to be supplied as test and scratch partitions. We aim at being as robust as the native and the QEMU block device (short: qemu-blk), and define failure of this benchmark as VMSH-BLK failing any test that succeeds on native or qemu-blk. Since the "quick" `xfstests` mostly produce small block device accesses, we create a long-running test, the *sustained load test*, that calculates the sha256 checksum of a large OS image.

Results Out of the 619 tests, all succeed natively. For both qemu-blk and VMSH-BLK, three tests (0.5%) fail. The three failed test cases are related to *quota reporting*, *i.e.* reporting file system statistics. Additionally, some tests do not apply to our setup, *i.e.* tests for a different file system or wrong XFS version, and are automatically skipped by `xfstests`. To summarise, since VMSH-BLK passes all tests that are passed by known-good devices, we conclude that the VMSH-BLK device has no regressions w.r.t. qemu-blk.

Supported Hypervisor	QEMU, kvmtool, Firecracker, crosvm
Unsupp. Hypervisor	Cloud Hypervisor
Tested LTS kernels	v5.10, v5.4, v4.19, v4.14, v4.9, v4.4

Table 4.1: Hypervisor and kernel support.

4.6.2 Generality

To showcase the generality of our approach, we evaluate the portability of VMSH across different hypervisors and stable Linux kernel versions (see Table 4.1).

Hypervisors We develop VMSH using QEMU as the primary target. However, we expand our scope to the following KVM-based hypervisors: QEMU [212], kvmtool [275], Firecracker [2], crosvm [95], and Cloud Hypervisor [54].

VMSH is able to support 4 out of the 5 hypervisors. Cloud Hypervisor is the exception as it uses PCIe's MSI-X messages for its interrupt handling. Therefore, it is incompatible with MMIO as a VirtIO transport channel. We plan to extend VMSH to support VirtIO over PCI for Cloud Hypervisor.

The second challenge we face is Firecracker's restrictions on what system calls are allowed to be executed by each thread individually, using seccomp [74]. For now, we disable the seccomp filter for Firecracker as it interferes with our system call injection. In the future, we will either provide a VMSH compatible seccomp profile for Firecracker or implement a heuristic that only runs system calls on threads that are allowed by seccomp.

Kernel versions Side-loading code into the Linux kernel can be quite challenging as there is no stable internal kernel API or ABI. To keep a project like VMSH maintainable, it is necessary to ensure that only a minimal kernel API is used. We develop VMSH against the latest version of the kernel, 5.12 at the time of development. To estimate how much maintenance will be required to support future versions, we backport to older kernel versions. We focus mainly on long-term support (LTS) versions, as those versions are guaranteed to receive security and build fixes for a long period. The analysed kernel versions are listed in Table 4.1. We run VMs using QEMU with the guests running each of the kernel versions. We then try to attach to them with VMSH and analyse the changes needed for that kernel version to work.

The most impactful change across versions is that the memory layout of kernel symbols, which we need to parse before uploading our own binary to the guest, changed twice. However, by using consistency checks, *i.e.* checking whether a kernel symbol name points to a valid string, we are able to check all variants in parallel. For

2 out of the 10 required kernel functions (kernel_read and kernel_write), we have to support different variants to maintain compatibility.

Structure definitions that we pass to kernel functions when registering devices are more brittle: 2 out of 4 kernel structures have to be conditioned depending on the kernel version. It took one person a week's worth of time to cover 5 years of kernel development. From this, we conclude that we can also support newer kernel versions in the future with a reasonable amount of effort.

4.6.3 Performance

We evaluate VMSH's performance using a range of workloads: (A) the Phoronix test suite [203], (B) impact of attaching VMSH on the guest application's performance, (C) fio [126], and (D) the responsiveness of the console device.

A: Phoronix test suite We start with evaluating how VMSH affects performance of real-world applications based on the Disk test suite [210] of the Phoronix Test Suite [203]. The version of Phoronix Test suite is newer (v10.6.0) than the version used in CNTR evaluation (v7.8.0), hence the set of tested applications is different. This suite consists of Compile bench [1], DBENCH [19], fs_mark [274], Flexible I/O Tester [126], IOR [263], PostMark [133] and Sqlite [243]. We use the default parameters defined by the Phoronix Test Suite. In this benchmark, we compare QEMU's block device, qemu-blk, with VMSH-BLK, our block device in VMSH.

The results are shown in Figure 4.5. On average, VMSH is $1.5 \times \pm 0.6$ slower than qemu-blk. The fio tests accessing large chunks of data (2 MB) are the slowest benchmarks, being up to $3.7\times$ slower than qemu-blk. fio is the only benchmark of the suite that uses direct IO. This bypasses the guest page cache and hits the block device with every request, which explains the slow down. Other applications (CompileBench: IO workload of a Linux kernel build process, Postmark: mailserver workload with small files, FS-Mark: file creation, DBENCH: file server workload) are more read and file system metadata (inode) heavy. These types of workloads benefit more from a fast page cache and fast in-kernel processing, and therefore have less or no overhead. Unexpectedly, Sqlite insertion turns out to be not very write-heavy, but it spends significant time creating and unlinking its journal (inode heavy operation). The IOR benchmark writes a file with increasing block size. In contrast to fio, it uses the page cache with a hit rate of approximately 20%. Therefore, there is less overhead when run in VMSH compared to the baseline.

To summarise, we see acceptable overheads w.r.t. to the real-world applications, with an average $1.5\times$ slowdown compared to qemu-blk. In practice, the guest

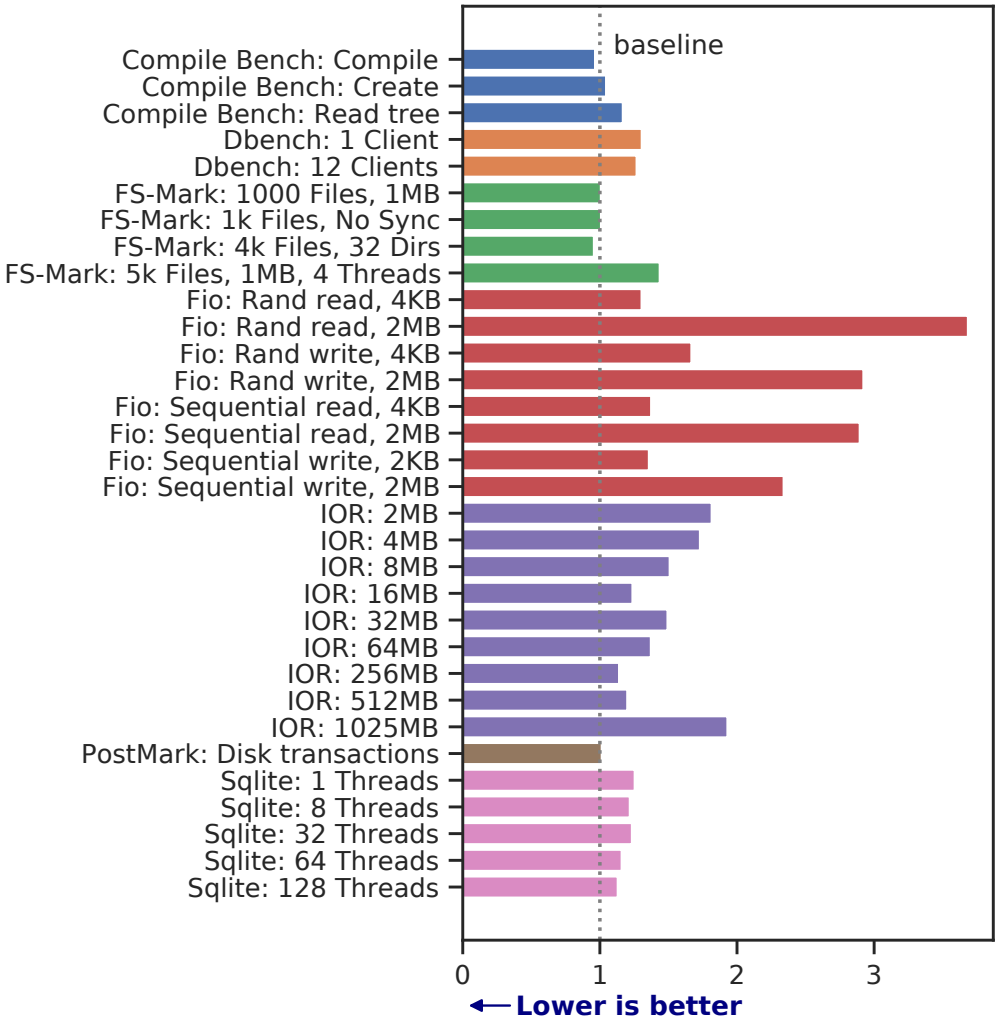


Figure 4.5: Relative performance overhead of VMSH-BLK for the Phoronix Test Suite compared to qemu-blk.

workload applications will continue to use the QEMU block device, and not VMSH-BLK. They will therefore not suffer from these slowdowns. The only applications affected by this slowdown are the ones using the device mounted through VMSH-BLK, which should not impact developers' productivity significantly.

B: Guest device performance under VMSH We now evaluate the performance impact of VMSH on the other devices attached to the guest, unrelated to VMSH. We do so by running comparative benchmarks with fio [126] and measure two metrics, throughput and the number of operations per second (IOPS), on qemu-blk devices while a VMSH-BLK device is attached to the VM. Using fio's libaio backend, we measure the maximal throughput by using the most favourable conditions, *i.e.* large block sizes (256 KiB) and sequential accesses. We measure the IOPS by choosing

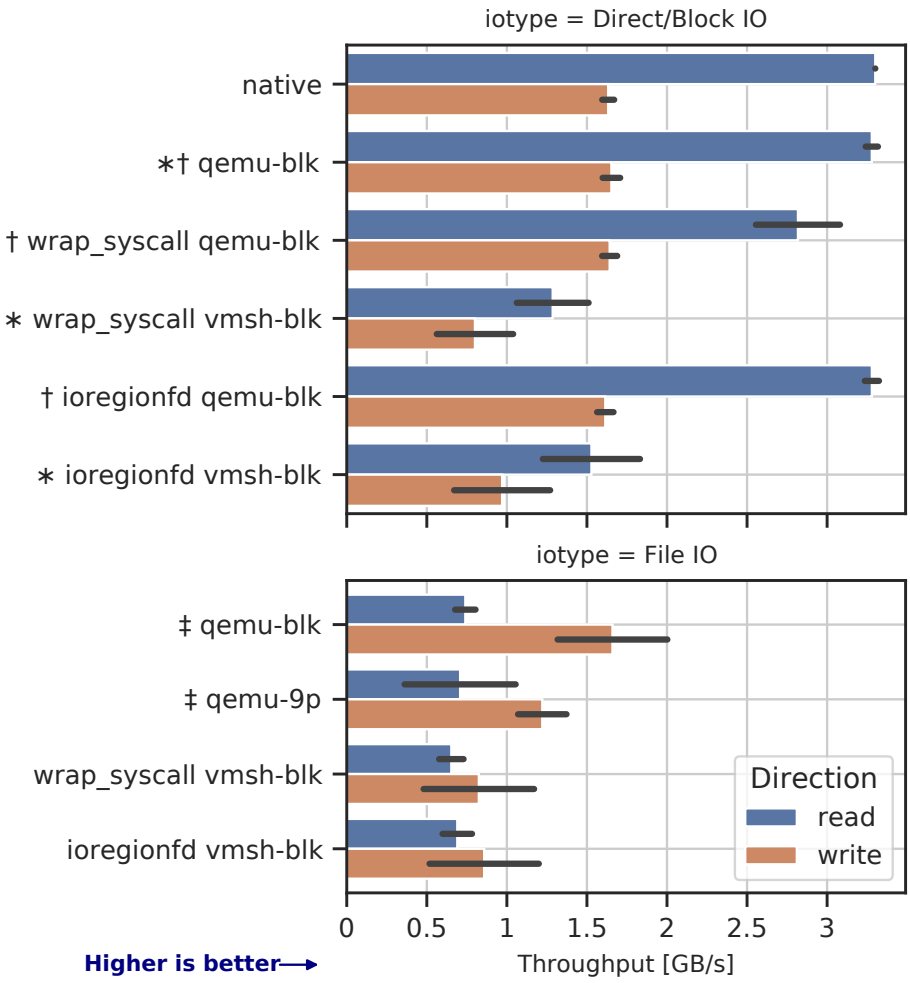


Figure 4.6: IO bandwidth/throughput. Best-case scenario.

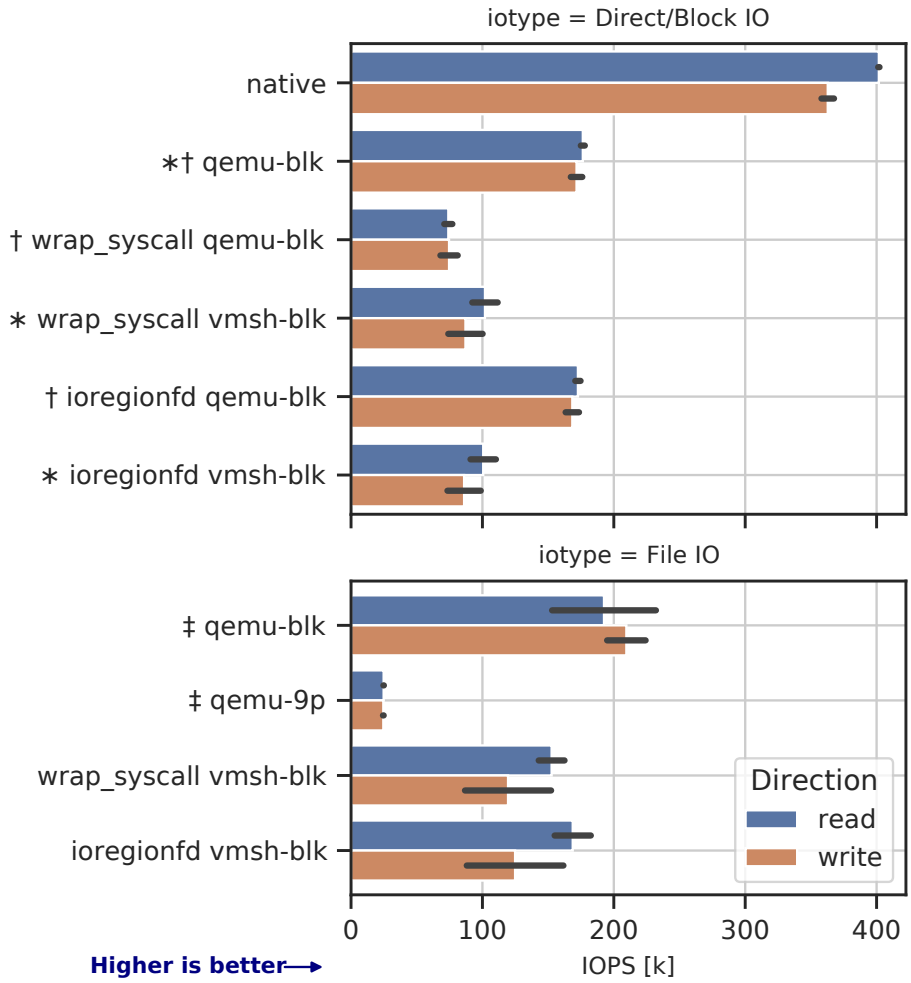


Figure 4.7: IO operations per second (IOPS). Worst case scenario.

Figure 4.8: `fio` with different configurations featuring qemu-blk and VMSH-BLK with direct IO, and file IO with qemu-9p.

small block sizes (4 KiB), thereby maximising per-access software overheads, and sequential accesses, to avoid hardware bottlenecks.

Figure 4.6 shows the throughput results of these experiments while Figure 4.7 shows IOPS. Qemu-blk shows the performance of the vanilla QEMU block device, with no VMSH-BLK device attached. The other setups with qemu-blk show the performance of the device while a VMSH-BLK device is attached, using different implementations of the device (ptrace or ioregionfd, see § 4.5). The interesting values for guest device performance under VMSH are tagged by a † symbol.

Our measurements show that when VMSH is attached to a VM, the throughput and IOPS of qemu-blk devices on the VM are the same as without VMSH when using the ioregionfd implementation. However, with the wrap_syscall implementation, both throughput and IOPS on the qemu-blk device are negatively impacted. Read throughput is reduced by 1.5× and IOPS by 6×. This performance degradation is due to the overhead added to every system call performed by QEMU and its devices. For every `VMEXIT` triggered by an MMIO access, VMSH has to check if it is related to a VMSH-BLK device. This is not a problem with the ioregionfd implementation since KVM already filters MMIO accesses for the VMSH MMIO region in the kernel.

The overheads of the ptrace implementation violate the goal of non-invasiveness. Since this is the most important performance metric for VMSH, ioregionfd is the best implementation of VMSH-BLK.

C: VMSH-BLK performance with fio Using the same fio benchmarks, we now evaluate the intrinsic performance of VMSH-BLK. We first compare it to qemu-blk using direct/block IO. We then compare our block device-based approach to the host file system sharing using file based IO with the 9p protocol (virtio-9p [213]). Results are also shown in Figure 4.8. `native` shows the performance of the benchmarks running directly on the host, with no virtualisation involved, and showcases the best performance achievable on the machine. The setups with VMSH-BLK show the IO performance of the device attached through VMSH with both implementations.

First, we observe that the native throughput can be achieved through virtualisation with direct IO. However, in terms of operations per second, `native` is at least 2× faster than any virtualised solution. This is due to additional data copies and context switches between the hypervisor and the host kernel.

As for VMSH-BLK, throughput and IOPS are halved compared to qemu-blk, indifferent to the used implementation (see results tagged with ∗). This degradation is expected since VMSH triggers more context switches than qemu-blk. IO operations are cooperatively handled by the guest driver running in the VM process and the

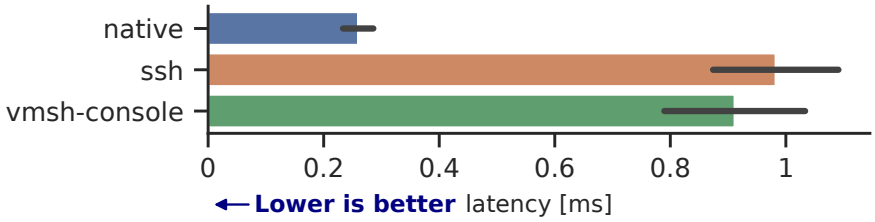


Figure 4.9: VMSH-console responsiveness compared to SSH.

VMSH virtual device running on the host (see Figure 4.4). In this benchmark, the time spent copying data between the guest and the host page cache is identical for qemu-blk and VMSH-BLK, thus leaving the number of context switches as the main reason for the performance hit. Over the same sampling period, we measure twice as many context switches for VMSH-BLK compared to qemu-blk.

With file-based IO, the read throughput significantly drops, due to the use of the page cache (see ‡). Fio sequentially accesses new blocks and never reuses previously read blocks, therefore suffering the cache's overhead while never actually using it. Qemu-9p has poor IOPS compared to qemu-blk (7.8× lower) because of the use of two stacked file systems. Every operation goes through the guest file system and page cache, as well as through the host's file system and page cache, therefore crippling qemu-9p's IOPS.

Finally, VMSH-BLK suffers a 94% write and 7% read overhead in throughput compared to qemu-blk (40% write/2.3% read overhead compared to qemu-9p), but still has good IOPS (14% degradation compared to qemu-blk and is 7× better than qemu-9p). The latter is the most important metric for VMSH because attached devices would be more prone to small sized IOs than large ones (see § 4.6.4 for use cases).

D: VMSH-console responsiveness For interactive scenarios, e.g. a console, throughput is less relevant than latency. We evaluate this by comparing the latency of the VMSH console to SSH and to *"the minimum viewing time needed [by a human] for visual comprehension"* [206].

We measure the round-trip of a shell input by connecting one end of a pseudo-terminal seat (`pts`) [164] to a shell. We then use the other end to submit an echo command to the shell and measure the time elapsed until the echo response arrives. Our measurements show that, with around 0.9ms, the latency of the VMSH console is very similar to the one of SSH (see Figure 4.9). The latency of the VMSH console is an order of magnitude faster than the capabilities of the human eye [206], making it sufficient for real life use cases.

4.6.4 Use-cases

To show the applicability of VMSH in real-world scenarios, we implement and publish three use cases.

Use-case #1: Serverless debug shell First, we demonstrate that VMSH fits well into serverless stacks, and improves their dependability properties [194]. In general, Function-as-a-Service (FaaS) systems are hard to debug because when requests cause errors, it is difficult to pinpoint the source of the error [228]. To help developers debug FaaS deployments, we provide them with an interactive shell in lambda-function instances. In particular, we integrate VMSH into vHive [264], a *knative*-compliant stack running serverless workloads in slim Firecracker-containerd VMs [2, 83]. Thereafter, we parse logs from vHive's lambda functions for errors, and then locate the Firecracker process that hosts the faulty lambda in order to attach to its hosting VM with VMSH and provide an interactive shell to it. While the user interacts with this shell provided by VMSH, our integration prevents shutdown of the lambda-function's VM by scale-down events. Overall, VMSH can thus be integrated into existing virtualised lambda environments, e.g. vHive, in a non-invasive manner without changing the environment's fundamental design.

Use-case #2: VM rescue system In cloud environments, when users lock themselves out of their VMs, they need rescue assistance from their hosting provider. Therefore, the providers offer a range of rescue systems, e.g., password recovery services to their customers [109, 67]. However, this usually requires a user-installed agent in the VM image, a reboot to access the file system directly, or booting a recovery virtual machine that has access to the file system. With VMSH, we build a simple, agent-less recovery image containing the `chpasswd` [159] command, that can be attached while the VM is still running. In general, VMSH can be used to build different kinds of rescue systems without interrupting the VM.

Use-case #3: Package security scanner With the increasing popularity of containers, cloud providers offer services to scan containers automatically for security vulnerabilities [92, 9, 116]. With VMSH, this service can be expanded to VMs without the need for additional agents inside the VMs. In particular, we write a scanner that checks the installed packages in Alpine Linux-based virtual machines against an online database [5] of known security vulnerabilities and report them.

4.7 Related work

We discuss the related work that solves similar use-cases.

Guest agents The trivial solution to many of VMSH's use-cases is to install agents connected to a network into the VM guest. For instance, SSH [165] is typically used for interactive debugging. For tasks like automated management of updates, user accounts or configurations, cloud providers offer a multitude of agents [11, 93, 94, 180, 167, 96]. Agents are also used for distributed tracing in serverless environments [130, 238, 85, 50, 254]. According to Sambasivan et al., this variety is justified, as one size does not fit all use-cases [228]. VMSH on the other hand is agent-less, attaches on-demand and does not interfere with the guest's userspace by default. Its maintenance, configuration and policy enforcement can be done independently of the guests.

Virtualisation Chen and Noble describe the problem of recovering high-level OS state from guest memory [51]. This 'semantic gap' has since been approached [91] and formalised [202]. Executing code inside a guest has been done by reusing userspace execution contexts [103, 75] or by injecting kernel modules [200, 234, 270], akin to VMSH's sideloader. Introspection usually aims at stealthiness and erases proofs of tampering the guest's execution [47, 89, 270].

To keep the host isolated from the guest, additional VMs are proposed to contain the inspection tool [200, 88]. VMSH has the same guarantees towards the host by only exposing a dedicated block device and console. However, our guest overlay is more tightly coupled to the guest, which enables an easier tooling workflow for the user.

Container Contrary to VM introspection as done by VMSH, there is no semantic gap to bridge with containers. CNTR [251] creates a nested namespace in a container. The host file system is then made available via fuse and mounted into the root. This way, a user can bring all their tools with them into the container. In the context of Kubernetes, ephemeral containers [144] can be used to deploy software, e.g. an interactive debugging environment, into another pod. This approach is locked in to Kubernetes. Systemd-sysext [171] overlays file systems with extension images using overlayfs [166]. It can be used to install packages without modifying the underlying file system. VMSH's guest overlay, on the other hand, avoids all dependencies on the guest userspace to maximise generality.

VM miniaturisation Library OSes [230, 36, 260, 35] reduce VM size by merging the kernel and user application into a single binary.

Unikraft [145] combines the aspects of micro-library OSes and unikernels [158, 155, 156] to reduce the kernel's CPU and RAM overheads. VMSH is orthogonal since it targets the overhead due to userspace tools not vital to the application. Micro-kernels instead split up their functionality horizontally, which is beneficial for verification and security [106, 26, 137]. VMSH follows similar principles by offering essential functionality in a separate host process and guest overlay, acting upon IPC.

New hypervisors are built [211, 2, 95, 54], smaller and less complex, to reduce overheads [172, 214, 182] and attack surface. Many of them are written in memory-safe languages [2, 95, 54, 170], while others are formally verified [151]. Their miniaturisation is also advanced, as virtual devices are extracted into separate processes with vhost-user [168, 248, 281]. While vhost still requires modifications on the hypervisor side, VMSH does not and operates non-cooperatively.

4.8 Limitations and future work

The current implementation of VMSH is limited to Linux operating system on x86_64 cpu architecture. In future, we want to port VMSH to more architectures like ARM64 and RISC-V and also support different operating systems. We discuss these steps needed for a port in § 4.5.

Another aspect to improve could be the integration into VM orchestration frameworks such as Vmware Vcloud [276] or OpenStack [87] to allow VMSH to work in a distributed setting.

Finally, to make loading code into the guest operating system more robust, a new virtio device would be useful. Such device could allow to launch a static binary with specified arguments in the guest os to allow running new programs independent of the guest userspace. This would simplify maintenance a lot, since the custom kernel component of VMSH is expected to require the most adaption with future OS versions.

4.9 Summary

In this chapter, we presented VMSH, a hypervisor-agnostic system to build lightweight VMs. VMSH provides an abstraction of guest overlays to extend lightweight VMs at run time independently of the guest and hypervisor. Using this abstraction, VMSH enables lightweight VMs to extend their VM images with additional tools and services on-demand at run time. We design VMSH as a system for hypervisor-independent side-loading into a VM, a generic guest-overlay that does not impose limitations on both the original guest application or the spawned

service, and a device that can be attached to hypervisors non-cooperatively. Our evaluation shows that VMSH is compatible across many hypervisors and Linux versions, that it does not slow down the original VM guest, and that its use-cases have the potential to reduce image sizes of lightweight VMs.

Source code availability VMSH is publicly available as an open-source project along with the complete evaluation setup [128].

Chapter 5

RKT-IO : A Direct I/O Stack for Shielded Execution

The shielding of applications using trusted execution environments (TEEs) can provide strong security guarantees in untrusted cloud environments. When executing I/O operations, today's shielded execution frameworks, however, exhibit performance and security limitations: they assign resources to the I/O path inefficiently, perform redundant data copies, use untrusted host I/O stacks with security risks and performance overheads, and fail to encrypt I/O data universally. This prevents TEEs from running modern I/O-intensive applications that require high-performance networking and storage.

We describe RKT-IO,[1] a direct userspace network and storage I/O stack specifically designed for TEEs that combines high-performance, POSIX compatibility and security. RKT-IO achieves high I/O performance by employing direct userspace I/O libraries (DPDK and SPDK) inside the TEE for kernel-bypass I/O. For efficiency, RKT-IO polls for I/O events directly interacting with the hardware instead of relying on interrupts, and it avoids data copies by mapping DMA regions in the untrusted host memory. To maintain full Linux ABI compatibility, the userspace I/O libraries are integrated with userspace versions of the Linux VFS and network stacks inside the TEE. Since the I/O stack runs entirely within the TEE, thus omitting the host OS from the I/O path, RKT-IO can transparently encrypt all I/O data and does not suffer from host interface/Iago attacks. Our evaluation with Intel SGX TEEs shows that RKT-IO is 9× faster for networking and 7× for storage compared to host- (SCONE) and libOS-based (SGX-LKL) I/O approaches (see Figure 5.6).

[1] Pronounced "rocket I/O"

5.1 Introduction

Cloud computing offers economies of scale for computational resources combined with ease of management, elasticity, and fault tolerance. At the same time, it increases the risk of security violations when applications run in untrusted third-party cloud environments. Attackers (or even malicious cloud administrators) can compromise the security of applications. In fact, many studies show that software bugs, configuration errors, and security vulnerabilities pose serious threats to cloud systems, and software security is cited as a barrier to the adoption of cloud solutions [229].

Hardware-assisted *trusted execution environments* (TEEs), such as Intel SGX [122], ARM Trustzone [23], RISC-V Keystone [222, 148], and AMD-SEV [15], offer an appealing way to make cloud services more resilient against security attacks. TEEs provide a secure memory region that protects application code and data from other privileged layers in the system stack, including the OS kernel/hypervisor. TEEs are now commercially offered by major cloud computer providers, including Azure [181], Google [100], and Alibaba [53].

TEEs, however, introduce new challenges to meet the performance requirements of modern I/O-intensive applications that rely on high-performance networking hardware (e.g. >20 Gbps NICs) and storage (e.g. SSDs). Since TEEs are primarily designed to protect in-memory state, they only offer relatively expensive I/O support to interact with the untrusted host environment [62]. Early designs relied on expensive *world switches* between the trusted and untrusted domains for I/O calls. A thread executing an I/O operation must exit the TEE before issuing a host I/O system call, which incurs overhead due to the security sanitisation of the CPU state including registers, TLBs, etc.

More recent designs used by shielded execution frameworks (e.g., SCONE [25], Eleos [197], and SGX-LKL [208]) employ a *switchless* I/O model in which dedicated host I/O threads process I/O calls from TEE threads using shared memory queues. To avoid blocking TEE threads when waiting for I/O results, these frameworks employ user-level threading libraries inside the TEE to execute I/O calls *asynchronously* [241].

While such *switchless asynchronous* designs improve I/O performance over the strawman *synchronous world switching* design, current frameworks still exhibit significant performance and security limitations: (1) they manage resources inefficiently by requiring dedicated I/O threads outside the TEE, which incurs extra CPU cycles when busy polling syscalls queues. These additional I/O threads also require fine-grained performance tuning to determine their optimal number based on the application threads and I/O workload; (2) they perform additional data copies

between the trusted and untrusted domains, and the indirection via shared memory queues significantly increases I/O latency; (3) the untrusted host interface on the I/O path has security and performance issues: the host interface is fundamentally insecure [49, 183], and requires context switches, which are expensive for high-performance network and storage devices; and (4) they lack a universal and transparent mechanism to encrypt data on the I/O path. Instead, they rely on application-level encryption, which is inefficient, potentially not comprehensive, and incompatible with full VM encryption models.

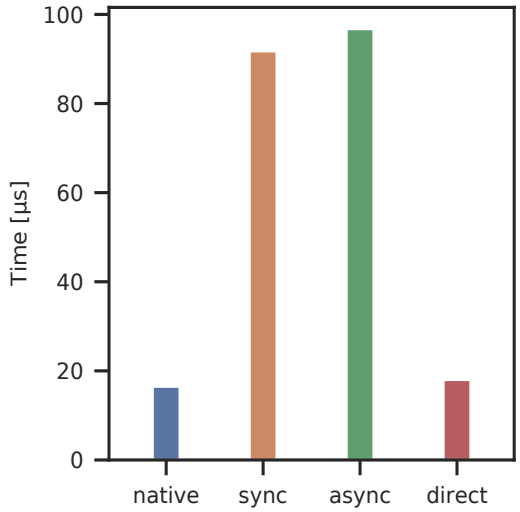


Figure 5.1: System call latency with `sendto()`

To overcome these limitations, we argue for a fundamentally different design point where we re-design the I/O stack based on *direct userspace I/O* in the context of TEEs. To exemplify our design choice, we compare the direct I/O approach within TEEs with three alternative I/O approaches, measuring the performance of the `sendto()` syscall with 32-byte UDP packets over a 40GbE link for (i) native (not secured), (ii) synchronous and (iii) asynchronous syscalls within TEEs (secured). As Figure 5.1 shows, native system calls (16.4 µs) and the direct I/O based approach (17.9 µs) take approximately the same time, while we see higher per-packet processing time for the synchronous (91.7 µs) and asynchronous (96.7 µs) system calls. By bypassing the host I/O support, TEE I/O stacks can avoid both performance overheads and security limitations.

Our design for a TEE I/O stack therefore has the following goals: (a) *performance:* we aim to provide near-native performance by accessing the I/O devices (NICs or SSDs) directly within the TEEs; (b) *security:* we aim to ensure strong security guarantees, mitigating against OS-based Iago [49] and host interface attacks [183];

and (c) *compatibility:* we aim to offer a complete POSIX/Linux ABI for applications without having to rewrite their I/O interface.

To achieve these design goals, we describe RKT-IO, an I/O stack for shielded execution using Intel SGX TEEs. The key idea behind the RKT-IO design is to combine (a) I/O userspace libraries (DPDK [77] and SPDK [121]) for direct hardware I/O access with (b) the POSIX abstractions provided by a Linux-based library OS (LKL [192]) inside the TEEs. This combination results in a high-performance I/O path, while preserving compatibility with off-the-shelf, well-tested Linux filesystems and network protocol implementations inside the TEE. Since the I/O stack runs in the protected domain of the TEE, RKT-IO provides improved security, as it does not rely on information from the untrusted host OS.

The design of RKT-IO embodies four principles to address the aforementioned limitations of current frameworks:

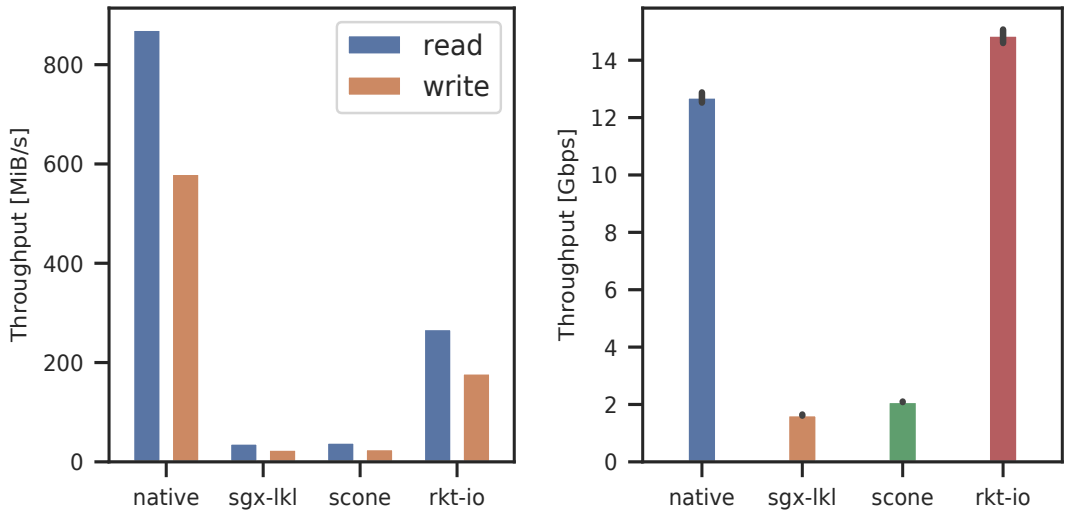
- RKT-IO adopts a *host-independent I/O interface* to improve performance and security. This interface leverages a direct I/O mechanism in the context of TEEs, where it bypasses the host OS when accessing external hardware devices. At the same time, it leverages a Linux-based libOS (LKL [192]) to provide full Linux compatibility.

- RKT-IO favors a polling-based approach for *I/O event handling* since TEEs do not provide an efficient way to receive interrupts on I/O events.

- RKT-IO proposes a judicious *I/O stack partitioning* strategy to efficiently utilize resources and eliminate spurious data copies. It partitions the I/O stack by directly mapping the (encrypted) hardware DMA regions into untrusted memory outside the TEE, and runs the I/O stack within the TEE.

- RKT-IO provides *universal and transparent encryption* in the I/O stack to ensure the confidentiality and integrity of data entering and leaving the TEE. It supports Layer 3 network packet encryption (based on Linux' in-kernel Wireguard VPN [76]) for networking, and full disk encryption (based on Linux' `dm-crypt` disk mapper [68]) for storage.

Our evaluation with a range of micro-benchmarks and real-world applications shows that RKT-IO provides better performance compared to SCONE (a host-OS based approach) and SGX-LKL (a libOS-based approach). For example, the throughput of RKT-IO's network stack (measured by `iPerf` [123]) is up to 9× higher (see Figure 5.2b), and the read/write bandwidth of RKT-IO's storage stack (measured by `fio` [126]) is up to 7× higher (see Figure 5.2a).



(a) Storage stack performance with `fio` (b) Network stack performance with `iPerf`

Figure 5.2: Micro-benchmarks to showcase the performance storage and network stacks across different systems

5.2 Motivation

TEEs provide the ability to create hardware-assisted protected domains in a process address space, as shown in Figure 5.3. TEEs protect the confidentiality and integrity of the application's code and data inside the TEE. It is also possible to verify the integrity of the code running inside the TEE via remote attestation. This enables users to run security-sensitive workloads in an otherwise untrusted execution environment.

5.2.1 Threat model

Our threat model extends the standard threat model for TEEs [34, 25]. As in the prior work, we assume a powerful adversary who has control of the entire system software stack, including the host OS and the hypervisor. In line with previous work, we do not address the physical tampering of the CPU package, denial of service attacks, and side channel attacks [45, 102, 265, 104, 280].

For I/O operations, applications running inside the TEE rely on the untrusted host OS for access to I/O devices, such as NICs and SSDs. On the I/O path, an adversary may tamper with the data being exchanged through the untrusted host interface, and compromise the confidentiality and integrity of the application running inside the TEE [49, 48, 272, 42, 149]. More specifically, the host interface may leak sensitive data, also known as interface attacks [183], which can expose the application state to the untrusted host. In addition, the host interface implementation

can be malicious itself, and therefore compromise the security of the application running inside the TEEs e.g., by manipulating the return values of syscalls, also known as Iago attacks [49]. Lastly, an attacker may use software/hardware probing to intercept data on the host's I/O path by DRAM interface snooping, installing malicious hardware with DMA access [177], or performing cold boot attacks [240]. Only universal end-to-end encryption of data on the I/O path can mitigate these types of attacks.

5.2.2 Analysis of existing I/O mechanisms

A protected application within the TEE communicates with the outside environment by performing I/O operations for accessing the filesystem or network stack. To support the I/O operations with the untrusted environment, TEEs require a *world switch*, where a thread executing the I/O operation switches between the trusted and untrusted domains, and then issues the syscall to the host OS. I/O operations, when invoked through the synchronous syscall mechanism adds a constant world switch overhead, incurred due to the necessary micro-architectural security-associated sanitizations, such as additional cache/ TLB flushes, page permission checks, etc. [62]. For example, a world switch costs ≈10,170 cycles, which is roughly 5× expensive than a syscall (≈1800 cycles).

To avoid the costly world switches between the trusted and untrusted domains, current shielded execution frameworks [25, 197, 208, 258, 34, 120] have adopted alternative *switchless* designs, which can be broadly categorized as (Figure 5.3): (i) host-based; and (ii) library OS-based. We compare these approaches across three dimensions: performance, security, and compatibility.

(1) Host-based frameworks (e.g., SCONE [25], Eleos [197], Intel SGX SDK [120]) rely on the host OS for the I/O operations. Among these frameworks, we focus on SCONE as the state-of-the-art system for our baseline. SCONE improves the I/O performance by leveraging the concept of asynchronous system calls [241]. In the async model, a set of dedicated I/O threads run (busy polling) outside the TEE to process the syscall requests issued by a thread from inside the TEE via shared memory queues.

(2) LibOS-based frameworks (e.g., Haven [34], Graphene-SGX [258], SGX-LKL [208]) rely on customized library OSes inside the TEE for handling I/O operations. Note that these library OSes still use the underlying host OS in the backend (via the untrusted host-based syscall interface) to access the hardware devices. As far as the I/O path is concerned, Haven and Graphene-SGX rely on synchronous mode for I/O operations, where the I/O threads block for the request completion. In this regard, SGX-LKL supports improved performance based on *asynchronous* I/O mechanism [241], similar

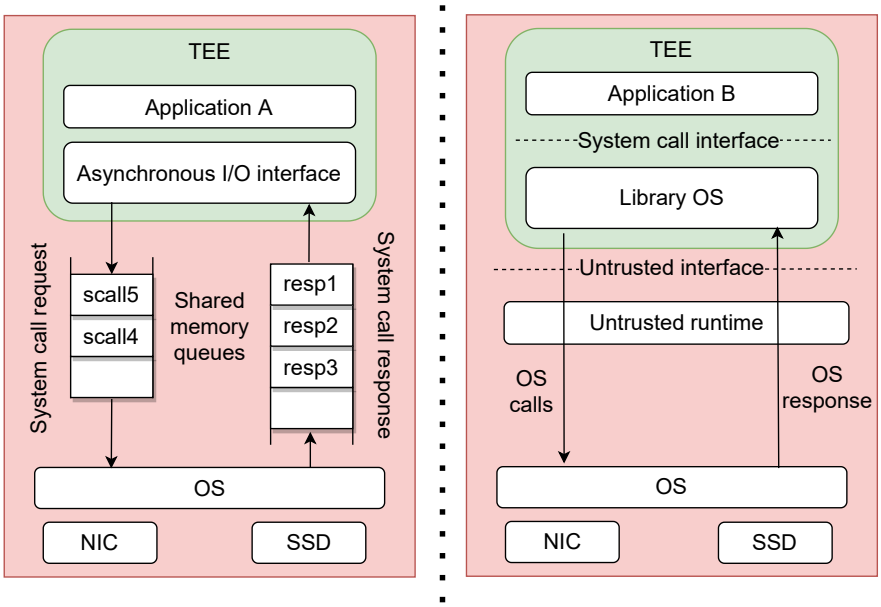


Figure 5.3: Two possible shielded execution architectures for I/O support in TEEs: *(left)* application A uses a pure host OS based approach, and *(right)* application B uses a library OS inside the TEE to process the I/O operations. (Regions in green are trusted, whereas red regions are untrusted.)

to SCONE. Therefore, we focus on SGX-LKL as the state-of-the-art system for our LibOS baseline.

Performance. Both approaches use the resources on the I/O path inefficiently: (1) they rely on dedicated *I/O threads* for issuing the I/O calls. This incurs extra CPUs cycles due to the busy polling of the syscalls queues; (2) they require fine-grained tuning to set the optimal number of I/O threads due to the tight coupling with the application threads and I/O workload; (3) they require additional data copies—data needs to be copied from the TEE to the untrusted host memory for the asynchronous I/O threads, before it is processed by the underlying host OS, requiring another copy; and (4) both approaches incur latency penalties due to the indirection involved on the asynchronous I/O path.

Security. Both approaches depend on the untrusted host OS, which make them vulnerable to Iago attacks [49] and host-interface attacks [183], but they differ with respect to the degree of dependence on the host OS: the host-based approach expose a wider interface with the untrusted host OS by allowing protected applications to directly issue a large set of syscalls. Although syscalls return values are sanitized by network and file systems *shielding layers*, they are susceptible to more attacks due to the increased interactions with the untrusted host OS. On the other hand, the

libOS-based approaches provide better security since they expose only a limited set of syscalls to the untrusted host OS. Furthermore, since the libOS-based approaches are flexible to adopt, a custom library OS to fit the application requirements can be developed, using which a protected application can further improve its security by adopting a libOS with lower TCB.

Compatibility. Host-based approaches can provide POSIX or Linux ABI compatibility, allowing them to support unmodified legacy applications. For instance, SCONE can support off-the-shelves filesystems and network stacks based on Linux. However, libOS-based approaches offer the possibility of specialization at the cost of limited or no support for the existing filesystem and network stacks. In general, this makes them less amenable for supporting unmodified legacy applications, with a notable exception of SGX-LKL that offers full POSIX/Linux ABI compatibility by using a library version of the complete Linux kernel (LKL [192]).

5.2.3 Problem statement and our approach

In this work, we aim to build a high-performance, secure, and compatible I/O stack for shielded execution. For performance, we aim to improve latency and throughput of I/O operations compared to a switchless asynchronous I/O approach. We also want to minimize reliance on the untrusted host for improved security. Lastly, we aim for full compatibility for existing applications by supporting the Linux ABI/POSIX standard. To achieve these goals, we next summarize our high-level approach and associated *four design principles*.

#1: Host-independent I/O interface. Current host OS- and libOS-based shielded execution frameworks rely on the underlying host OS for I/O operations. Instead, we argue for a fundamentally different design point in which we favor a host-independent I/O interface that uses direct I/O in with TEEs. A direct I/O approach improves performance and security: compared to a switchless asynchronous syscall mechanism, it reduces the latency and increases the throughput of I/O operations by directly accessing the hardware (NICs and SSDs) and minimizing the number of data copies. Since direct I/O minimizes the host OS interactions as much as possible by accessing the I/O hardware directly from the TEE (i.e., there are no I/O-related syscalls after the initialization phase), it also leads to improved security. We combine this approach with a Linux Library OS (LKL) inside the TEE to provide full Linux ABI compatibility.

#2: I/O event handling. In the context of SGX, we cannot rely on the interrupt-driven I/O execution because there is no efficient way to receive interrupts or timer events

within TEEs. Instead of interrupt-based I/O, RKT-IO uses a polling-based approach for handling I/O events in TEEs in which RKT-IO explicitly polls I/O response queues for completed requests or new data. Such an approach for I/O event handling is a natural fit with direct I/O libraries (DPDK/SPDK) that combine polling with the run-to-completion model for fast I/O devices, which avoids the performance bottlenecks of interrupt-based execution [37, 201, 131].

#3: I/O stack partitioning. While direct I/O libraries fit better with I/O event handling, their adoption in the context of TEEs presents an interesting challenge: DMA regions for untrusted I/O devices cannot be mapped directly into the TEE as DMA access is prohibited for security reasons. We therefore need a way to efficiently write to and read from a DMA region. RKT-IO achieves this by judiciously partitioning the direct I/O stack into two parts: the driver code for I/O stacks runs inside the TEE, and DMA memory regions for I/O devices are outside, as part of the untrusted host memory.

#4: Transparent encryption. Since we cannot trust the host, network or storage hardware, all data leaving the TEE must be encrypted to ensure confidentiality, which is typically handled at the application layer in today's frameworks. Unfortunately, such an approach is error-prone, as application may not universally encrypted all of its I/O paths, e.g., exposing data through unencrypted legacy network protocols or file systems. Instead, RKT-IO supports a transparent and universal mechanism to provide full disk (block layer) and network encryption (Layer-3) by relying on the library OS.

5.3 Overview

Figure 5.4 shows the high-level architecture of RKT-IO. It consists of: (a) a *network stack* that is derived from the Linux kernel exposing a socket API and is backed by the Data Plane Development Kit (DPDK) [77]. Using DPDK, RKT-IO gets direct access to the NIC from within the TEE; (b) a *storage stack* that provides a complete filesystem abstraction (e.g., the Linux ext4 filesystem). It uses the Linux VFS layer to interact with the block device layer, which is implemented by the Storage Performance Development Kit (SPDK) [121] over the NVMe protocol to communicate with the SSD; and (c) a *runtime environment* that integrates the storage and network stacks based on the Linux kernel library (LKL), a userspace library OS port of the Linux kernel [192]. LKL provides Linux system calls as userspace function calls inside the TEE. A modified version of the musl standard C library (libc) [185] exposes a POSIX interface to the application on top of the LKL system call interface.

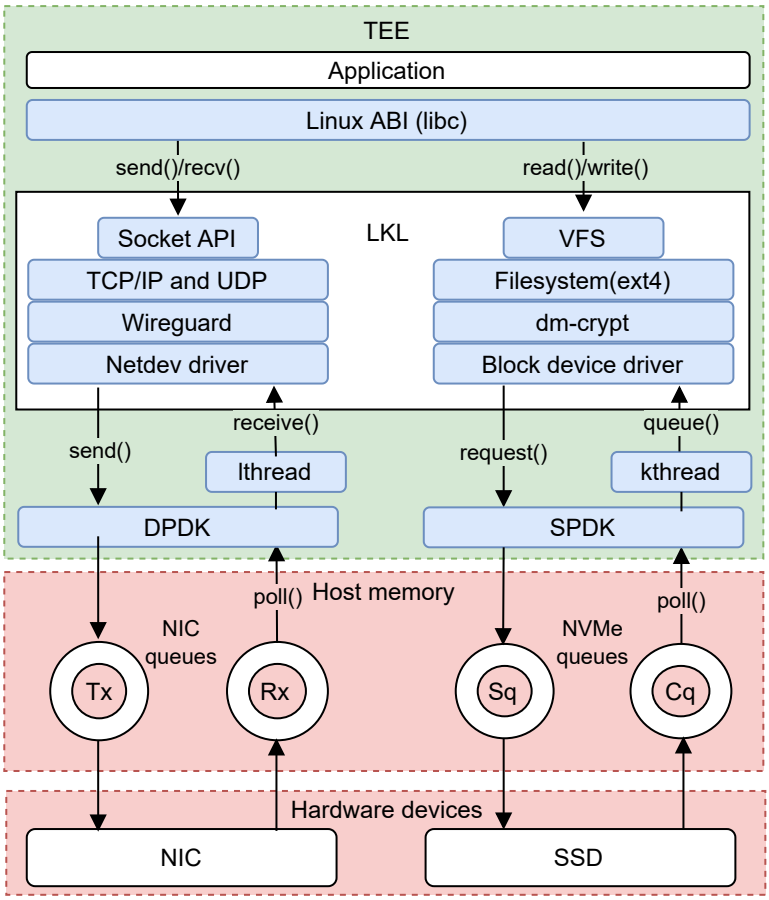


Figure 5.4: Architecture overview of RKT-IO (network stack on left; storage stack on right)

An application can be built with common toolchains/package managers and put into a Linux ext4 filesystem image. At runtime, a loader sets up the TEE, the I/O stacks, the LKL library OS, and then mounts the filesystem image as the root file system. After that, the application and its linked libraries are loaded into memory from the root file system, and the application can now use RKT-IO's modified musl libc library.

A typical I/O request issued by the application begins with a system call, via the libc API. The system call request is processed by the library OS within the TEE. Depending on whether the request is made to a network or storage device, the library OS issues calls to the userspace driver via the network and storage stacks, respectively. The userspace drivers add requests to the appropriate request queues (transmission queue (Tx) for NIC; submission queue (Sq) for NVMe), which are mapped into the untrusted DMA memory region. Likewise on the receiving path, the drivers continuously poll the completion queues (receive queue (Rx) for NIC; completion queue (Cq) for NVMe), which are also mapped into the untrusted region.

The userspace drivers notify the library OS on response completion and can return the data if requested.

Next we explain the two building blocks of RKT-IO: (a) the direct userspace I/O mechanism; and (b) the POSIX abstractions from the Linux-based library OS.

Direct I/O libraries. Our I/O stack incorporates direct I/O libraries in the TEE to avoid system calls and directly access the I/O devices. Our work builds on two popular userspace direct I/O libraries, DPDK [77] and SPDK [121], which support network and storage devices, respectively.

A direct I/O approach in the context of TEEs has both advantageous and disadvantageous. On one hand, its polling-based approach is well-suited for TEEs because interrupts are not permitted within TEEs. Furthermore, polling for completion reduces the total latency, and has been shown to lead to a better design for high-performance I/O devices (NICs and SSDs) [37, 201, 131]; on the other hand, a direct I/O (zero-copy) philosophy is fundamentally incompatible with TEEs, because a DMA memory region cannot be mapped directly inside the TEE due to security restrictions.

To overcome this limitation, RKT-IO adopts a split architecture in which driver code for DPDK and SPDK runs inside the TEE, but it maps the DMA memory regions for the NIC/NVMe queues outside the TEE in untrusted host memory. More specifically, the DPDK driver polls the NIC for received packets, which are then explicitly copied into the TEE from the DMA regions. Likewise, the NVMe driver uses its highly parallel asynchronous, lockless and poll-for-completion design to access the underlying SSD. The drivers map hardware queues and PCIe registers into the DMA region, and add requests and poll responses to distinct queues. Thereby, our design follows a "one-copy" approach that copies the data between the untrusted DMA region and the TEE.

Although DPDK and SPDK help improve the performance and security of applications inside TEEs, it is challenging for developers to use them due to their low-level I/O interfaces. DPDK provides high-speed packet processing capabilities at Layer 2 in the network stack; SPDK offers only a block layer interface (and a rudimentary file system called BlobFS [44]). These low-level interfaces are not sufficient for most applications, which rather need a full network stack (e.g., TCP/IP) and full filesystem (e.g., ext4) support.

Library OS with Linux ABI. RKT-IO uses the Linux Kernel Library (LKL) [192] to provide a mature POSIX implementation with a virtual file system (VFS) layer and a TCP/IP stack. LKL is a complete architecture port of Linux to userspace, which provides components such as the kernel page cache, work queues, filesystem and

network stacks, and crypto libraries.

In our design, the application and the LKL library OS run in a single virtual address space within the TEE. RKT-IO thus avoids user/kernel context switches, as system calls are invoked through functions calls, and it also eliminates data copies between the user/kernel space. By combining LKL with DPDK/SPDK, applications do not need to be modified to use low-level I/O APIs and instead can use POSIX APIs, while taking advantage of the performance and security guarantees offered by RKT-IO.

In addition, RKT-IO leverages LKL to provide universal and transparent encryption to ensure the confidentiality of data entering and leaving the TEE. RKT-IO supports Layer-3 network packet encryption based on Linux' in-kernel Wireguard VPN [76], and full disk encryption based on Linux' `dm-crypt` disk mapper [68].

5.4 Design

We next present the detailed architecture of RKT-IO around the four design principles from §5.2.3.

5.4.1 Host-independent I/O interface

RKT-IO's design aims to provide support for I/O operations while reducing dependencies on the host OS. After boot up of the TEE environment, RKT-IO loads the user-provided application and its dependencies into the encrypted memory. It provides its own ABI-compatible variant of the musl libc implementation, which makes system calls against the integrated LKL library OS – a non-MMU Linux architecture port.

Multi-threading and scheduling is implemented in RKT-IO's libc: it implements cooperative userland threads that are scheduled on a fixed number of host OS threads. The userland threads yield control and allow other threads to be scheduled on the same host OS thread when they are blocked, e.g., when locks are taken; when a thread sleeps; or when a blocking system call against the library OS kernel is issued. To build the host-independent I/O interface, we next discuss three main design issues that RKT-IO addresses to adapt LKL for high-performance networking and storage.

Symmetric multiprocessing (SMP). To allow high-performance I/O operations, the I/O stack must be parallel to take advantage of SMP architecture. By default, LKL does not support multi-threading, as shown in Figure 5.5 (original design on the left). When multiple threads attempt to enter the kernel context, they need to obtain a single lock. This lock protects the data structures associated with a virtual CPU. This make

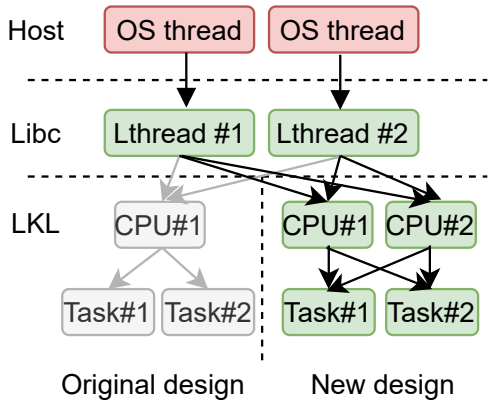


Figure 5.5: RKT-IO SMP architecture

the LKL kernel a bottleneck, because the backend I/O drivers and most applications are parallel.

To make the kernel scalable, we modify LKL to add SMP support. With that, LKL can provide multiple virtual CPUs as shown in Figure 5.5 (new design on the right). The threading primitives required by the kernel for SMP are adapted from the native architecture (i.e. x86 in our prototype). This change also introduces additional kernel threads that are needed to handle inter-process interrupts and timer events that are broadcast to multiple virtual CPUs.

To evaluate the effectiveness of our SMP design, we use fio [126] with random read/write requests on a 1 GB file while increasing concurrency. Figure 5.6a shows that the throughput for the storage stack increases linearly with more threads (from 1 to 8 threads) for both read and write requests.

Threads stack management. With an SMP architecture, the number of threads in LKL also increases. To implement threads, LKL uses its OS-specific host interface. In its default implementation for a POSIX-compatible OS, LKL creates a POSIX thread with the architecture's default stack size (8 MB on x86-64). In an environment in which the OS has MMU support, the stack is only backed by physical memory as it grows in size.

RKT-IO, however, has no MMU support and must therefore pre-allocate stack memory. This results in a significant memory overhead when implementing SMP, as more threads consume proportionately higher physical memory. This is an issue given that TEE technologies such as SGX have limited physical memory (≈94 MB in x86 SGX enclaves) that are usable without costly paging operations.

To solve this problem, we reduce the kernel thread stack size to 8 KB, which is the same stack size that the Linux kernel uses for *x*86_64. The SPDK/DPDK framework makes heavy use of inline functions, resulting in stack depths larger than 8 KB. In

RKT-IO, we therefore remove function inlining in parts of the SPDK/DPDK code base.

This approach for thread stack management turns out to be extremely effective: we observe 155 kernel threads for a single-threaded application with 8 virtual LKL cores. By switching from 8 MB to 8 KB stacks, we save 1.2 GB of memory.

Event scheduling timer. An I/O stack relies on timer event for several periodic tasks, e.g., to flush out dirty pages, to schedule TCP re-transmissions, etc. In our experiments, the original timer support in LKL is too slow for our design because it would only schedule 1–3 events per second. The slow timer is not an issue for the native LKL storage and network drivers, because they only delegate I/O requests to threads that execute host system calls, making it less reliant on periodically scheduled tasks. In RKT-IO, however, timer events are used to periodically poll I/O devices, which makes them a bottleneck for scheduling tasks. Therefore, we need to design a new timer implementation to meet the scheduling requirements on the direct I/O path.

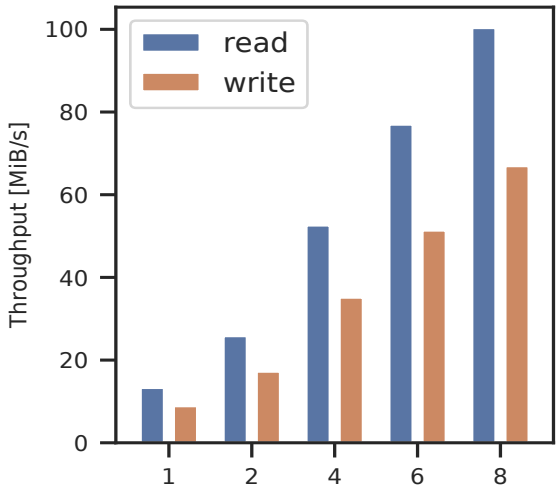
Originally, LKL implements a one-shot timer interface in which the kernel registers functions to be called after a certain time by creating a thread per event. The thread sleeps for a given time interval before invoking the kernel callback. RKT-IO implements a periodic timer instead. With a frequency of 50 Hz, it calls a generic interrupt function, and the kernel checks which tasks must be executed within this tick. To do so, a single thread is created once, which performs a sleep system call in a loop before notifying the kernel. With this new design, the polling mode I/O stack becomes able to handle I/O events at a high rate.

5.4.2 I/O event handling

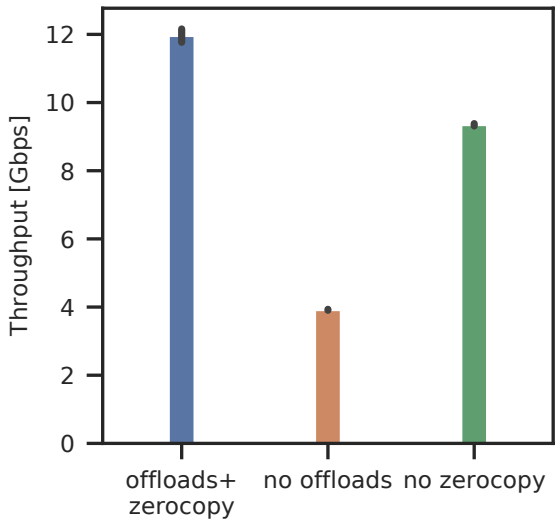
We design our I/O stack based on polling, and therefore we must re-design the library OS's network and filesystem interfaces to support this. We achieve this by mapping registers and DMA memory regions into RKT-IO's virtual address space.

While polling can consume more CPU cycles than interrupt handling, especially in I/O-intensive applications, interrupts become a bottleneck. There is a recent trend in OS design to switch to hybrid polling/interrupt approaches to meet the performance requirements of network and storage hardware [283, 79]. We next explain our polling-base design, which we find most suitable for both block (SSDs) and network (NICs) devices.

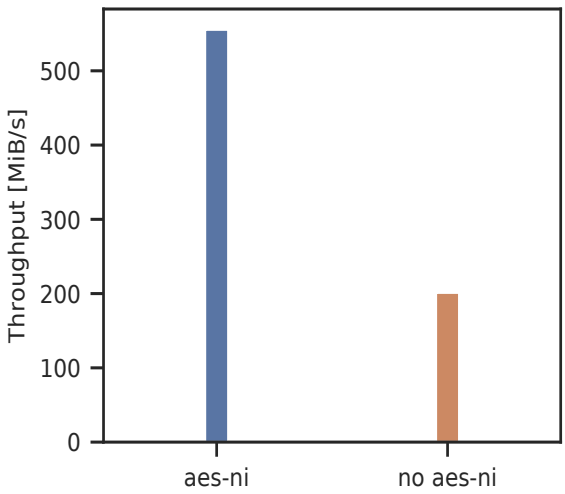
Block device polling. Figure 5.4 (right) shows the data path when applications access the filesystem. System calls issued by the application are first dispatched by the virtual filesystem and then delegated to the actual filesystem (in our experiments, ext4). When the filesystem reads/writes data from the underlying block device, the



(a) Effectiveness of the SMP design w/ `fio` with increasing number of threads



(b) `iPerf` throughput w/ different optimizations



(c) Effectiveness of hardware-accelerated crypto routines

Figure 5.6: Micro-benchmarks to showcase the effectiveness of design choices in RKT-IO

data is cached in the page cache.

The SPDK driver puts incoming requests in the NVMe queue. When queuing requests, the driver also polls for completed requests in the corresponding completion queue. If there are outstanding requests, it schedules a polling task. The polling task periodically polls the completion queue until all outstanding requests are acknowledged and notifies the kernel about each completed item.

In an SMP environment, single hardware queue pairs can easily become a bottleneck due to lock contention, which is caused by multiple threads trying to issue requests concurrently. To overcome this problem, the NVMe standard allows the creation of multiple request/response queue pairs. This allows I/O requests to be issued in parallel, while improving data locality in NUMA systems. We assign one queue pair per virtual LKL CPU and bind one polling thread to each.

As described in §5.4.1, LKL has the concept of virtual CPUs, which are protected by locks, so that only one thread can access them at a time. Due to this, RKT-IO does not need additional locks around its own queues, as they are already protected by the CPU locks. One challenge when introducing multiple queues is to not increase the CPU overhead due to polling: too much polling on a particular queue steals CPU cycles from the application or other queues; while not enough polling increases latency and decreases throughput. RKT-IO puts the polling threads to sleep if no outstanding requests are due.

An advantage of RKT-IO's design with one queue-pair per CPU is that it can also safely poll without locks in the request function, because the actual poll thread cannot be run at the same time on the same CPU. This reduces context switches, as the request function is often called from the application thread during a system call.

Network device polling. Similar to the storage stack, the network stack also relies on polling. Figure 5.4 (left) shows the data path for networking. An application can use the full POSIX socket API with all extensions, as supported by Linux. New data sent by the application is stored in a kernel-side socket buffer, and the socket buffer is placed in a software queue. On a software interrupt, buffers from this queue are passed to the DPDK-based network driver, which puts the data into the NIC's transmission queue. Packets are received by a dedicated polling thread.

While implementing the network stack, we experimented with different setups on how to manage polling. Our first design used multiple queues for sending and receiving. This approach, however, makes network throughput worse: each receiving queue must be polled by a dedicated polling thread, which takes too many CPU cycles away from the application and increases latency. Likewise, for sending queues, RKT-IO needs to poll the completion status, as the Linux kernel uses this information to

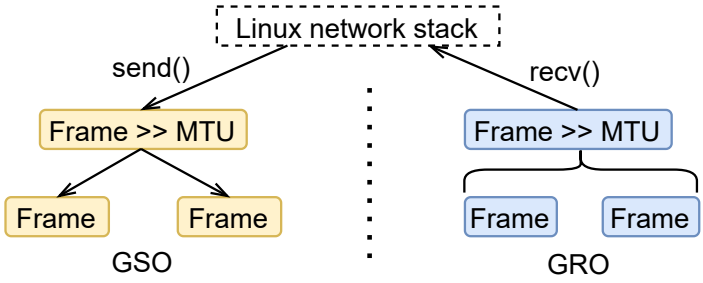


Figure 5.7: Generic segmentation offload and generic-receive offload

adjust its TCP window size.

Ultimately, we decide on having a single thread dedicated to polling a single queue, which is faster, because the cost of context switches exceeds the cost of polling. To reduce the overhead of scheduling the polling thread, we move it out of the kernel scheduler into the underlying userland scheduler. Before it starts polling, it acquires an LKL CPU lock to get ownership, so it can safely access Linux kernel data structures. After it has processed every received packet, it releases the ownership of the CPU lock.

Bufferbloat mitigation through eager queue cleanup. To counter bufferbloat [46], network congestion avoidance for the TCP stack in Linux measures how many packets are still queued by the NIC. In the original DPDK driver design, however, old packets are not freed as soon as packets are sent, but when new packet buffers override the old entries in the send ring buffer. This is too late for Linux, where the critical threshold is around 0.5 MB while a send queue in a NIC is significantly larger (i.e. 8 MB for our NIC). As a result, connections are throttled to a rate below 1 Gbps on a 40 Gbps NIC.

To counter that, we redesign the queue cleanup algorithm to free old send buffers when new packets are queued for sending or when the Linux network stack's threshold for a TCP connection is exceeded. In our experiments, this boosts `iPerf` throughput from 300 Mbps to 25 Gbps without encryption and 15 Gbps with encryption.

NIC offloading support. To achieve high throughput when processing packets, the offloading mechanism available in modern NICs must be used by a high-performance network stack. When network streams are sent, they need to be divided into smaller Ethernet frames with a maximum transfer unit (MTU) of commonly 1500 bytes. The smaller Layer-3 packets (i.e. TCP/IP) need new updated headers to reflect the changed size, sequence number and checksum. On the receiver side, it is also typically too expensive to traverse the whole network stack for each packet. Optimizations in either software or hardware are required to re-assemble the

payload from smaller TCP packets into larger buffers. In addition, the payload and network headers are often not stored continuously in memory. This needs to be communicated to the NIC to avoid having to copy parts to a new continuous buffer.

To address the problem of assembling and disassembling network streams, we use generic segmentation offloading (GSO) and generic receive offloading (GRO). GSO, as shown in Figure 5.7 (left), requires implementation changes in DPDK to allow for the segmentation of large network streams into smaller Ethernet frames in the hardware. For GRO, as shown in Figure 5.7 (right), we rely on a software solution, as described in §5.5.2, to avoid entering the network stack for each small packet by aggregating them in larger buffers. Furthermore, RKT-IO configures the NIC to offload checksum computation for network headers. We also make use of DPDK's segmentation support to make the NIC read headers and payload chunks from different memory locations, thus avoiding copying them to a new buffer.

To evaluate the effectiveness of these offloading techniques, we run `iPerf` with and without them, as shown in Figure 5.6b, with a single TCP stream. As for all network benchmarks, TLS is enabled in `iPerf`. With offloading, we obtain a 3× higher throughput, making DPDK's performance comparable to NIC-enabled offloading in the Linux kernel.

5.4.3 I/O stack partitioning for TEEs

Since storage and network devices cannot directly access TEE memory, their DMA memory regions need to mapped outside of the TEE. Conversely, the POSIX API forces the kernel to make a copy of the data passed in a system call because applications expect the memory to be re-usable. A naive implementation would therefore do two copies: one from the application buffer to the kernel and one from the kernel to the NIC DMA region.

RKT-IO reduces the number of copies to one, by copying data for sending directly from the application buffer to the hardware's DMA region. In systems that have access to an MMU, usually from a privileged ring, this can be achieved by re-mapping pages in virtual memory. RKT-IO, however, runs in unprivileged userspace and instead extends the Linux kernel memory allocator to support memory allocations in both encrypted TEE memory and unencrypted DMA memory (see §5.5.1).

One-copy for networking. This support in the Linux kernel memory allocator allows RKT-IO to allocate the data part of a socket buffer (short skb) in the NIC DMA memory. In turn, RKT-IO makes use of DPDK's external buffer support [64] (see §5.5.2) to transfer packets to the NIC without an extra copy.

To evaluate the performance improvements of a one-copy data path, we use the same `iPerf` benchmark as in §5.4.2 with the result shown in Figure 5.6b. When comparing all optimizations enabled with disabling the copy optimization in the receive/send path, we see a 21% improvement (11.6 Gbps vs. 15 Gbps).

One-copy for storage. Similarly, the NVMe device needs data to be written to a special DMA memory region in which NVMe queues are allocated. To avoid extra copies of the encrypted pages from the disk encryption layer, RKT-IO allocates those pages in the DMA memory outside of the TEE. When transferring pages from or to the NVMe device, RKT-IO uses the gather-scatter API of SPDK. This API allows to pass I/O vectors instead of continuous buffers, which is needed to pass multiple scattered kernel pages directly to the hardware. This optimization results in a throughput improvement of 7% for the block device.

5.4.4 Transparent encryption

Since all data that leaves the TEEs must be protected to avoid information leakage to the host, RKT-IO implements both transparent encryption of network traffic using a Layer 3 virtual private network (VPN) and full disk encryption.

For network protection, we find that many network-facing applications already support transport encryption using TLS. This is the preferred way, as it provides high-throughput and low protocol overhead thanks to highly optimized TLS stacks, such as OpenSSL [268]. If an application does not support TLS, RKT-IO also supports the Wireguard VPN [76] to encrypt network packets on Layer 3. It is integrated into the library OS as a tunnel, and it encapsulates encrypted IP packets into the UDP protocol using the ChaCha20 [147] stream cipher before forwarding them to the NIC.

For storage protection, we use the Linux `disk-mapper` crypto target that gives full disk encryption. It is set up to use AES 256-bit in XTS cipher mode before passing encrypted pages to the underlying block device.

Hardware acceleration. The LKL architecture by default only provides slow generic routines for AES encryption or cryptographic hashing, which makes full disk encryption slow. Optimized routines must be loaded from kernel modules, depending on which CPU extensions are available (i.e. AES-NI on Intel x86). RKT-IO ports the crypto modules from the respective native CPU architecture (i.e. x86) to speed-up block disk encryption. Therefore, we implement kernel module loading support (see §5.5.3).

Figure 5.6c shows the throughput before and after enabling hardware-accelerated crypto routines for sequential writes to a 10 GB file. Enabling acceleration increases throughput by 2.8×.

5.5 Implementation

RKT-IO builds based on SGX-LKL [208]. SGX-LKL provides the musl libc abstraction, userland threading and integration into LKL. RKT-IO extends SGX-LKL to the support the direct I/O network and storage stacks. In addition, RKT-IO re-designs several components to suite the direct I/O performance requirements, which were described previously in Section 5.4 and are explained in further detail in this section.

5.5.1 Runtime environment for the I/O stack

Driver setup. DPDK/SPDK configure the system to map the hardware queues into the RKT-IO virtual memory space. This is a privileged action requiring root permissions. RKT-IO delegates this task to a dedicated setuid binary, so that the actual SGX enclave can run with user privileges. For this to work we use DPDK's multiprocess feature, where the privileged process acts as a primary process and communicates over shared memory with our enclave, which runs as a secondary process. Additionally, we find that DPDK/SPDK needs root access to resolve its own virtual addresses to physical addresses in order to communicate with the hardware. We delegate this task to another setuid binary, that provides this service over a pipe.

Hugetables. DPDK/SPDK allocate the memory they use to communicate with the hardware using huge pages (either 4MB or 1GB large instead of 4KB on x86). RKT-IO uses this huge memory region as a page cache, which is why we want to allocate as many huge pages as possible. We find that with the default 1GB page size recommendation from DPDK this is not possible. The host operating system's page allocator causes memory fragmentation and it cannot find many unused continuous 1GB physical pages (only 4-5 GB pages on the system with 32GB RAM in our tests). Instead, we modify DPDK to allocate 4MB pages and align them continuously in memory by moving DPDK's metadata structure to a different offset. This way the page allocator that runs in our library OS can treat this memory as a one continuous chunk.

Page allocator. We extend the Linux page allocator to use DPDK/SPDK memory. The Linux kernel expects page data structures and cannot work with external buffers. We therefore re-use page flags used in the NUMA architecture to differentiate between memory allocated in the TEE and memory allocated in the DMA memory region. On top of that, we can also identify pages based on its address for additional security checks, whether the memory comes from the protected TEE memory or the unencrypted DMA memory region. We extended LKL to register DPDK/SPDK memory in their own "NUMA" zone on bootup. By default, the kernel never

allocates any memory in these zones except when a special flag (GFP_SPDK_DMA) is passed to the page allocator function. Additionally, we also add DMA memory support for kmalloc, which is the kernel's malloc equivalent. It builds on top of the page allocator by adding additional caches for different size classes. We add new cache data structures for the DMA memory region and make kmalloc select them if the GFP_SPDK_DMA flag is set.

5.5.2 Network stack

RKT-IO implements a new network device driver to integrate DPDK into Linux network stack. Furthermore, we make several modifications to DPDK itself to improve performance in context of the kernel network stack. To improve the TCP send performance and make DPDK competitive with the native Linux kernel driver, we implemented Generic Segmentation Offload (GSO) for the Intel 40-Gigabit Ethernet NIC family (called i40e in DPDK/Linux). On the receiving side we implement GRO (Generic Receiving Offload) using the kernel *napi_gro_receive()* function. This shortcuts parts of the network stack as packets get summarized to larger streams, without traversing the full stack for each packet. By default, DPDK comes with its own allocator for packet buffers. To avoid copying when transferring packets from Linux's socket buffer to the NIC, we make use of DPDK's external buffer support using the *rte_pktmbuf_attach_extbuf* function. For receiving packets DPDK does not offer support for external buffers, so we modify DPDK to allocate Linux socket buffers rather its own packet buffers.

5.5.3 Storage stack

RKT-IO implements a multi-queue block device driver using the *blk-mq* [60] interface to integrate SPDK as a block device into LKL. RKT-IO assigns one queue pair per CPU that is used when doing requests in our drivers *queue_rq* implementation to avoid locks between different CPUs. In the same function it also poll for outstanding requests. If there are outstanding requests, a dedicated polling kernel thread for this queue is woken up.

An important performance optimization we apply to this layer is speeding up disk encryption by loading native x86 kernel modules with hardware-accelerated optimized crypto routines. As the modules contain assembly instruction that are not position-independent code, we needed kernel module loading support in LKL to allow relocations at runtime. RKT-IO builds the kernel crypto modules from the normal x86 port with a small patch (56 LOCs) to align the kernel module initializer struct between the two architectures. At setup time it loads the kernel modules via

syscall. The kernel modules their-self are linked into the RKT-IO binary. The modules itself rely on x86 CPU feature checks based on CPUID to figure out which CPU extensions are available, we modify LKL to load this information during bootup based on CPUID information from outside of the enclave (CPUID itself is an illegal instruction inside the SGX enclave).

5.6 Evaluation

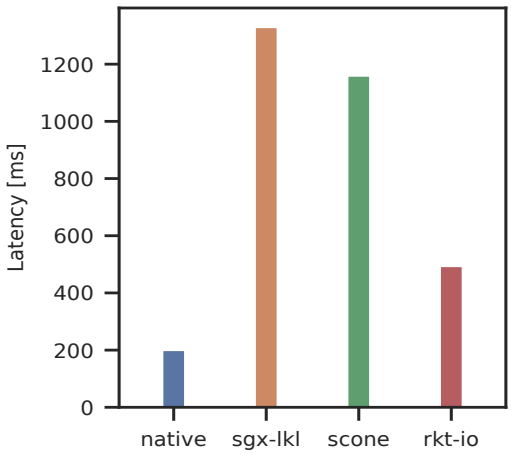
Our experimental evaluation is based on four real-world applications (see Figure 5.8): SQLite, Nginx, Redis and MySQL.

Testbed We perform our experiments using two machines with SGX as our TEE: Intel(R) Core(TM) i9-9900K each with 8 cores (16 HT), memory: 64 GiB, caches: 32 KiB (L1), 256 KiB (L2) and 16 MiB (L3), NIC: Intel Corporation Ethernet Controller XL710 for 40GbE QSFP+ (rev 02). NVMe drive: 2TB P4600. The host OS is running Linux 5.7.12.

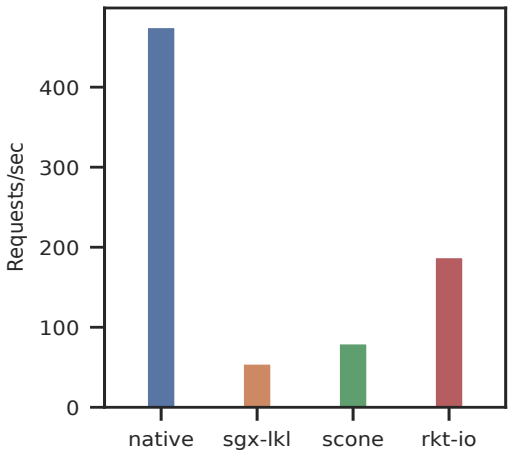
Baselines We compare our performance against the overall performance of these applications across three systems: native Linux (unsecured version), SCONE (a host OS-based approach), and SGX-LKL (a library OS-based approach). The applications are compiled against musl libc. For the native benchmarks, we use ext4 as a filesystem using disk-mapper for encryption with the aes-xts-256 cipher. We use the same configuration inside the TEE for SGX-LKL and RKT-IO. SGX-LKL accesses the block device as a file through the host interface, while RKT-IO accesses it via SPDK. For SCONE we enable fileshield [232] and store the data on ext4 without encryption. All network benchmarks have TLS enabled (Redis, MySQL and Nginx). Both native and SCONE-based benchmarks access the network through the host socket interface, while RKT-IO uses DPDK. To connect SGX-LKL to the native network adapter, we use a TAP interface that is bridged with the native NIC. For SCONE, we use tuning parameters recommended by the SCONE developers for our I/O-heavy workloads: two threads running inside the TEE with a system call queue assigned to each thread. Each system call queue has 7 I/O threads running outside the TEE.

5.6.1 Nginx web server

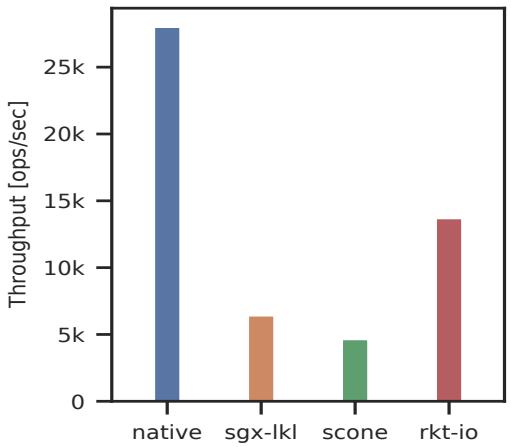
Methodology We evaluate Nginx [188] in a client-server configuration using two machines. We use the `wrk` HTTP benchmarking tool [277] to request a 3 MB file (average page size is according to [113]) via HTTPS. The benchmarking tool is setup



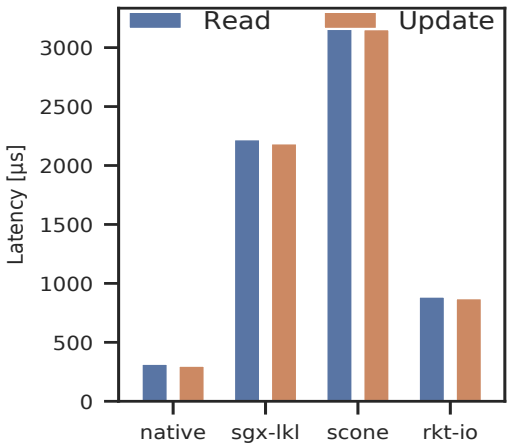
(a) Nginx latency w/ `wrk` [277]



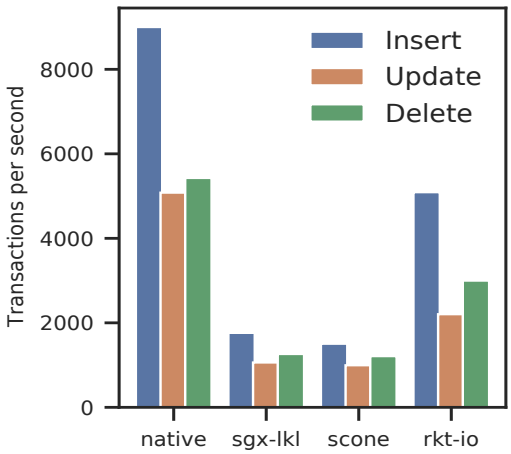
(b) Nginx throughput w/ `wrk` [277]



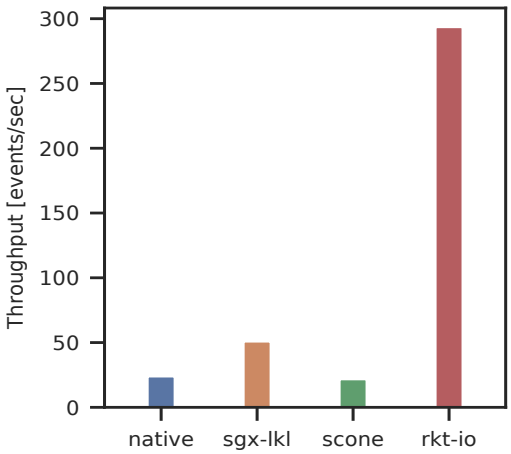
(c) Redis throughput w/ YCSB (A) [58]



(d) Redis latency w/ YCSB (A) [58]



(e) SQLite throughput w/ Speedtest [242]



(f) MySQL OLTP throughput w/ sysbench [245]

Figure 5.8: The above rkt-io/plots compare the performance of four real-world applications (Nginx, Redis, SQlite, and MySQL) while running atop native linux (no security) and three secure systems: SCONE, SGX-LKL and RKT-IO

as a client process running on another server machine, for 30 seconds using 16 threads and 100 concurrent HTTPS connections. We then report the throughput and latency of the server on this workload as requests per second and milliseconds, respectively. We compare the results across the four system configurations as shown in Figure 5.8a and Figure 5.8b.

Results RKT-IO incurs a lower average per request latency than SGX-LKL (2.7×) and SCONE (2.3×) as well as a higher throughput than SGX-LKL (3.4×) and SCONE (2.3×). There is still a performance gap compared to the native non-secure run, which has 2.5× higher throughput and 2.4× lower latency.

Our profiling of the web server shows that while serving the requests, Nginx spends 92% of the time in the kernel to process the network packets, while the rest of the time is being spent mostly in the userspace for encryption. This benchmark therefore shows the differences in the network stacks. In SGX-LKL, the network packets have to traverse the network stacks in the host and LKL. The host then has two additional network devices that the network packets have to pass (the tap interface and the bridge), and their respective firewalls.

SCONE can transfer network packets to the native host interface directly, however it still spends more time copying data from the TEE to its system call queue and from the system call queue to the host than RKT-IO, which can interact with the NIC directly.

5.6.2 Redis key-value store

Methodology We evaluate Redis [221] with the YCSB benchmarking framework [58], which is set up on another client machine. The key-value store is loaded with 100K key-value pairs. Thereafter, workload A of YCSB is used for a total of 10k operations using 16 threads. We then report the throughput and latency for the read and update operations on the key-value store in terms of operations per second and milliseconds, respectively. We compare the results across the four system configurations as shown in Figure 5.8c and Figure 5.8d.

Results RKT-IO's throughput is better compared to SCONE (2.9×) and SGX-LKL (2.1×); however it is slower than in the native execution (2.0×). The average latency per operation follows a similar trend, with the Redis server running on RKT-IO $2.8 - 2.9\times$ lower latency than native execution. However, RKT-IO is faster than SCONE (3.6×) and SGX-LKL(2.5×).

Our profiling of the experiments shows that the workload of this benchmark is also network bound, like Nginx, however we measure that only 50% of the time is

spent in the network stack, and remaining of the majority of the time is spent in TLS encryption. This alone would suggest that the SGX-based solutions should be closer to the native execution performance since they require less interaction with their I/O stacks; however, this is not the case. We see an increase in the activity of the enclave paging kernel thread (i.e. from 3× increase in CPU usage for RKT-IO when comparing the Nginx benchmark with Redis). Therefore, the run-time runtime difference is caused by increased EPC paging when Redis' in-memory data structures are accessed.

5.6.3 SQLite database

Methodology We evaluate SQLite [243] using its default configuration with journal mode set to delete and full synchronization. We use the Speedtest benchmark [243] shipped with SQLite to perform 15k transactions. We then configure the benchmark to perform 5k transactions each for the insert, update and delete operations. We report the throughput as transactions per second for each operation. We compare the results across the four system configurations as shown in Figure 5.8e.

Results RKT-IO performs $2.0 - 2.8\times$ better than SGX-LKL and $2.4 - 3.4\times$ better than SCONE. However, the performance of RKT-IO is lower for the native run (outside TEE) by $1.7\times$, $2.3\times$ and $1.8\times$ for insert, update and delete operations, respectively.

Our profiling of the experiments shows that the writes in a transaction are cheaper compared to creating/opening/flushing/unlinking the journal/WAL files. For such an I/O pattern, RKT-IO and SGX-LKL have an advantage over SCONE, since they can directly access inodes from the libOS inode cache because they implement the filesystem themselves, while SCONE has to perform host system calls.

Even though the writes performed by SQLite itself are comparably small (4KiB) to other operations completed around it, but the writes still needs to be synced to the disk to provide crash consistency. This is where the polling-based approach of RKT-IO falls behind the native execution, as RKT-IO has to spend more CPU cycles on polling to wait for the I/O completion.

5.6.4 MySQL database server

Methodology We evaluate MySQL [186] with the SysBench benchmarking tool [245]. The benchmarking tool is setup on another machine as a client to generate the OLTP workloads. We then compare the throughput of the server serving OLTP requests, in transactions per second. We compare the results across the four system configurations as shown in Figure 5.8f.

Results RKT-IO's throughput is better than native, SGX-LKL and SCONE, by 12.2×, 5.7× and 13.5× respectively. After doing an off-cpu analysis [193] we found mysql would spend a significant time doing table locks with futex. Since they both for SGX-LKL and RKT-IO do scheduling in userspace they were faster in switching between different threads. Combined with the fast userspace I/O stack of RKT-IO this speed up the overall execution.

5.7 Related work

I/O support for shielded execution. With the adoption of TEEs in cloud environments, shielded execution frameworks, such as Haven [34], SCONE [25], Graphene-SGX [258], Panoply [237], and SGX-LKL [208], are used to deploy applications with strong security properties. These frameworks provide OS functionality and associated run-time libraries to support unmodified legacy applications in TEEs. They promote portability, programmability and performance for shielded execution, and have been used to implement a wide-range of secure systems for storage [140, 30], data analytics [231, 285], data management [207], file storage [3], network functions [256, 204], decentralized ledgers [152], content delivery networks [108], machine learning [146], etc.

Current shielded execution frameworks primarily rely on existing OS functionality (i.e., syscalls to host OS or a libOS inside the TEE) for I/O operations, which differs from RKT-IO's design of providing a separate direct I/O stack within the TEE for storage and networking. SCONE [25], SGX-LKL [208], and Eleos [197] use switchless asynchronous I/O calls to mitigate I/O bottlenecks in the TEEs. This avoids expensive TEE world switches, and the use of I/O threads outside the TEEs improves I/O performance through asynchronous syscalls [241]. Since this approach relies on the host OS to handle I/O operations via dedicated I/O threads outside the TEE, it suffers from performance and security limitations: in terms of performance, it reduces the number of available threads for application execution, requires extra copies of the data and syscall arguments, and significantly increases I/O latency; since the host OS is responsible for performing I/O operations, it is also susceptible to Iago [49] and host interface attacks [183].

To overcome the limitations of switchless asynchronous I/O mechanisms, ShieldBox [256] uses Intel DPDK [77] as a user mode driver to support secure middleboxes based on the Click modular router. Likewise in the storage domain, Speicher [30] accesses persistent storage (SSDs) through Intel SPDK [121] within the TEE to provide a secure persistent KV store. Since these systems try to address I/O

bottlenecks, the corresponding I/O stacks are designed to operate at the lowest layer, and thus are incompatible with legacy applications that require POSIX network and file support. More specifically, Shieldbox only targets Layer-2 networking without TCP/IP support; in contrast, RKT-IO provides secure network (IP) and transport protocols. Similarly, Speicher operates at the block layer without filesystem support, but RKT-IO's I/O stack supports off-the-shelf filesystems (e.g., ext4, xfs, etc.) available in the Linux kernel. Finally, RKT-IO adopts a holistic design to provide both network and storage support in an integrated I/O stack.

High-performance I/O stacks. To meet the performance needs of I/O-intensive applications and leverage high-performance hardware, a range of I/O stacks have been proposed for networking (e.g., mTCP [127], netmap [223], StackMap [282], Sandstorm [175], and TAS [134]) as well as storage (e.g., Decibel [187], i10 [114], DiskMap [176], ReFlex [138], and PASTE [112]).

Our work builds on the designs of high-performance I/O stacks, especially stacks bypassing the OS kernel. RKT-IO supports secure I/O operations directly within TEEs, whereas these I/O stacks would require non-trivial changes for retro-fitting their architecture in the context of TEEs.

Efficient library OS design. Library OSs can improve application performance, while ensuring portability [81, 205, 39, 52]. Advances in high-performance networking and storage in data centres has led to a resurgence of library OSs support latency-sensitive applications: Arrakis [201], Demikernel [284] and IX [37] adapt a kernel-bypass design that splits functionality across control and data paths in order to support I/O-intensive applications that use high-performance NICs and SSDs. In the same spirit, RKT-IO favors a host-independent I/O interface based on LKL to avoid the OS on the critical I/O path for improved performance (and also for security). In contrast to these systems, we need to address additional fundamental challenges to make the direct I/O compatible in the context of TEEs. Since the DMA region cannot be directly mapped in the TEEs, our design requires additional "one-copy" instead of "zero-copy" to read/write data in the TEE. On the downside, these systems do not support POSIX compliant APIs.

5.8 Limitations and future work

In this section we discuss potential areas where we can improve RKT-IO in future.

While RKT-IO achieves a significant speed-up for IO operation in the SGX enclave, it is limited to Intel x86 hardware. In future the same approach could be ported to similar TEE implementations in other CPU architectures i.e. AMD's Secure Memory

Encryption (SME [65]), RISC-V's keystone [222] or ARM's Realm [24].

Secondly at the time of evaluation SGX still had a memory limitation of 128 MB protected memory per enclave. Intel in the meanwhile released the 3rd generation of Intel Xeon's, which increases the memory to up to 1 TB per enclave [273]. It would be interesting to redo the evaluation on those CPUs to be more competitive in comparison to native performance.

Lastly, the SPDK/DPDK framework is not hardened/optimized for use in TEEs. A re-implementation with focus on a smaller TCB, an untrusted host in mind and the use an of memory-safe language for the driver [80] would greatly improve security of RKT-IO.

5.9 Summary

In this chapter, we presented the design and implementation of RKT-IO, a direct I/O stack for shielded execution targeting high-performance networking and storage. Our I/O stack strives to overcome the performance and security limitations of switchless asynchronous I/O designs adopted in the host OS- and libOS-based shielded execution frameworks. This design goal is achieved by our judicious co-design of the userspace direct I/O libraries with the library OS (LKL) running in the trusted domain of TEEs. Thereby, our I/O stack facilitates switchless direct I/O with improved performance and security, while preserving the rich POSIX environment to support off-the-shelve filesystems and network stacks. We have implemented RKT-IO, as an end-to-end I/O stack, and extensively evaluated it using a wide-range of micro-benchmarks and unmodified real-world applications. Our evaluation shows the effectiveness of the individual system design components and overall approach; for instance, our network and storage stacks are $7-9\times$ compared to SCONE (host-based) and SGX-LKL (libOS-based) based on iPerf and Fio benchmarks, respectively.

Source code availability Our project is publicly available for the research community [250].

While with RKT-IO we address the security of applications in containers and virtual machines and with CNTR we improve the dependability of containers, there was no equivalent of CNTR for virtual machines. For this reason we built VMSH, that we present in the next chapter.

Chapter 6

Conclusion

Given the increasing complexity of virtualisation in data centres, how can we ensure that installed applications remain maintainable and secure? In this dissertation, we present three systems that address this aspect for virtual machines and containers by introducing new I/O abstractions. In our approaches, we focus on practical, deployable solutions that developers can adapt to their applications with minimal or no changes.

CNTR provides a way to extend application containers with tools from debug containers. At runtime, CNTR allows users to efficiently deploy the "slim" image and then extend it with additional tools as needed, by dynamically attaching the "fat" image. To accomplish this, CNTR transparently combines the two container images using a nested namespace, without making any changes to the application, container manager, or operating system.

VMSH allows services to be attached to running virtual machines as needed, so developers can deploy minimal, lightweight images without compromising their functionality. In this way, VMSH provides a zero-configuration out-of-band management for virtual management that does not depend on code in the guest userspace or additional network interfaces. It achieves this by locating the guest kernel and using this information to load and execute additional kernel code by injecting system calls into the KVM hypervisor via the debugging API. The kernel code bootstraps a lightweight container based on a file-system provided by a block device emulated by VMSH.

RKT-IO improves the I/O performance of applications running in cloud environments protected by trusted execution environments with a fast direct userspace network and storage I/O stack. RKT-IO achieves high I/O performance by using direct userspace I/O libraries (DPDK and SPDK) within the TEE for kernel-bypass I/O. To be efficient, RKT-IO polls for I/O events by interacting directly

with the hardware rathern than relying on interrupts, and it avoids data copies by mapping DMA regions in the untrusted host memory.

Bibliography

[1] Chris Mason <chris.mason@oracle.com>. *Homepage of Compilebench*. https://oss.oracle.com/~mason/compilebench/. 2021.

[2] Alexandru Agache et al. "Firecracker: Lightweight virtualization for serverless applications". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, 2020, pp. 419–434.

[3] Adil Ahmad et al. "OBLIVIATE: A Data Oblivious Filesystem for Intel SGX". In: *25th Annual Network and Distributed System Security Symposium (NDSS)*. 2018.

[4] Istemi Ekin Akkus et al. "SAND: Towards High-Performance Serverless Computing". In: *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. 2018.

[5] Alpine maintainers. *Alpine Linux security database*. https://secdb.alpinelinux.org/. 2021.

[6] Amazon. *Accessing Amazon CloudWatch logs for AWS Lambda*. https://docs.aws.amazon.com/lambda/latest/dg/monitoring-cloudwatchlogs.html. 2021.

[7] Amazon. "Amazon Elastic Block Store (EBS)". In: (2021).

[8] Amazon. *AWS X-Ray*. https://aws.amazon.com/xray/. 2021.

[9] Amazon. *Image scanning on Amazon ECR*. https://docs.aws.amazon.com/AmazonECR/latest/userguide/image-scanning.html. 2021.

[10] Amazon. *Working with AWS Lambda function metrics*. https://docs.aws.amazon.com/lambda/latest/dg/monitoring-metrics.html. 2021.

[11] Amazon. *Working with AWS Systems Manager (SSM) Agent*. https://docs.aws.amazon.com/systems-manager/latest/userguide/ssm-agent.html. 2021.

[12] *Amazon AWS Lambdas*. https://aws.amazon.com/lambda/.

[13] *Amazon Elastic Container Service (ECS)*. https://aws.amazon.com/ecs/.

[14] *Amazon's documentation on EBS volume types*. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html.

[15] AMD. *AMD Secure Encrypted Virtualization (SEV)*. https://developer.amd.com/sev/. Last accessed: Oct, 2020. URL: https://developer.amd.com/sev/.

[16] *Intel Active Management Technology*. Last accessed: Dec, 2021. URL: %5Curl%7Bhttps://www.intel.com/content/www/us/en/architecture-and-technology/intel-active-management-technology.html%7D.

[17] Andreas Lundqvist. *Linux distribution timeline*. https://de.wikipedia.org/wiki/Datei:Linux_Distribution_Timeline.svg. 2016.

[18] Andrew Lerner. "Predicting SD-WAN Adoption". In: (2015).

[19] Ronnie Sahlberg Andrew Tridgell. *Homepage of DBENCH*. https://dbench.samba.org/. 2021.

[20] Andy Honig and Nelly Porter. *7 ways we harden our KVM hypervisor at Google Cloud: security in plaintext*. https://cloud.google.com/blog/products/gcp/7-ways-we-harden-our-kvm-hypervisor-at-google-cloud-security-in-plaintext. 2021.

[21] Anil Madhavapeddy and David J. Scott. "Unikernels: Rise of the Virtual Library Operating System". In: (2014).

[22] Ali Anwar et al. "Improving Docker Registry Design based on Production Workload Analysis". In: *16th USENIX Conference on File and Storage Technologies (FAST)*. 2018.

[23] ARM. *Building a Secure System using TrustZone Technology*. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. Last accessed: Oct, 2020. URL: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[24] *Arm Confidential Compute Architecture*. https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture.

[25] Sergei Arnautov et al. "SCONE: Secure Linux Containers with Intel SGX". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016.

[26] Nils Asmussen et al. "M3: A hardware/operating-system co-design to tame heterogeneous manycores". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. Vol. 51. New York, NY, USA: Association for Computing Machinery, 2016, pp. 189–203.

[27] *AWS Lambda Pricing*. `https://aws.amazon.com/lambda/pricing`.

[28] *Azure Container Service (AKS)*. `https://azure.microsoft.com/en-gb/services/container-service/`.

[29] *Azure Functions*. `https://azure.microsoft.com/en-gb/services/functions/`.

[30] Maurice Bailleu et al. "SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution". In: *17th USENIX Conference on File and Storage Technologies (FAST)*. 2019.

[31] Ioana Baldini et al. "Serverless computing: Current trends and open problems". In: *Research advances in cloud computing*. Springer, 2017, pp. 1–20.

[32] Paul Barham et al. "Xen and the Art of Virtualization". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*. New York, NY, USA: Association for Computing Machinery, 2003, pp. 164–177.

[33] Diane Barrett and Greg Kipper. *Virtualization and Forensics: A Digital Forensic Investigator's Guide to Virtual Environments*. 1st. Syngress Publishing, 2010. ISBN: 1597495573.

[34] Andrew Baumann, Marcus Peinado, and Galen Hunt. "Shielding Applications from an Untrusted Cloud with Haven". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014.

[35] Andrew Baumann et al. "Composing OS extensions safely and efficiently with Bascule". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. New York, NY, USA: Association for Computing Machinery, 2013, pp. 239–252.

[36] Adam Belay et al. "Dune: Safe user-level access to privileged CPU features". In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, 2012, pp. 335–348.

[37] Adam Belay et al. "IX: A Protected Dataplane Operating System for High Throughput and Low Latency". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014.

[38] Kamal Benzekki, Abdeslam El Fergougui, and Abdelbaki Elbelrhiti Elalaoui. "Software-defined networking (SDN): a survey". In: *Security and Communication Networks* 9 (2016).

[39] B. N. Bershad et al. "Extensibility Safety and Performance in the SPIN Operating System". In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*. 1995.

[40] Ketan Bhardwaj et al. "Fast, Scalable and Secure Onloading of Edge Functions Using AirBox". In: *2016 IEEE/ACM Symposium on Edge Computing (SEC)*. 2016, pp. 14–27. DOI: 10.1109/SEC.2016.15.

[41] Pramod Bhatotia et al. "Reliable Data-center Scale Computations". In: *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*. 2010.

[42] Andrea Biondo et al. "The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[43] *Blobstore Programmer's Guide*. Last accessed: Nov, 2021. URL: %5Curl%7Bhttps://spdk.io/doc/blob.html%7D.

[44] *BlobFS: Blobstore Filesystem*. Last accessed: Oct, 2020. URL: %5Curl%7Bhttps://spdk.io/doc/blobfs.html%7D.

[45] Ferdinand Brasser et al. "Software Grand Exposure: SGX Cache Attacks Are Practical". In: *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. 2017.

[46] *Bufferbloat project*. Last accessed: Oct, 2020. URL: https://www.bufferbloat.net/.

[47] Martim Carbone et al. "Secure and robust monitoring of virtual machines through guest-assisted introspection". In: *International workshop on recent advances in intrusion detection*. Springer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 22–41.

[48] David Cash et al. "Leakage-Abuse Attacks Against Searchable Encryption". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015.

[49] Stephen Checkoway and Hovav Shacham. "Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface". In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*. 2013.

[50] Mike Y Chen et al. "Pinpoint: Problem determination in large, dynamic internet services". In: *Proceedings International Conference on Dependable Systems and Networks*. 2002.

[51] Peter M Chen and Brian D Noble. "When virtual is better than real [operating system relocation to virtual machines]". In: *Proceedings eighth workshop on hot topics in operating systems*. Elmau, Germany: IEEE, 2001, pp. 133–138.

[52] David R. Cheriton and Kenneth J. Duda. "A Caching Model of Operating System Kernel Functionality". In: *Proceedings of the 6th Workshop on ACM SIGOPS European Workshop*. 1994.

[53] Alibaba Cloud. *Alibaba Cloud's Next-Generation Security Makes Gartner's Report*. `https://www.alibabacloud.com/blog/alibaba-clouds-next-generation-security-makes-gartners-report_595367`. Last accessed: Oct, 2020.

[54] Cloud-hypervisor maintainers. *Project page of cloud-hypervisor*. `https://github.com/cloud-hypervisor/cloud-hypervisor`. 2021.

[55] Cloudflare. *Cloudflware workers*. `https://workers.cloudflare.com/`. 2022.

[56] *Cntr homepage*. `https://github.com/Mic92/cntr`.

[57] *Container optimized Linux distribution*. `https://coreos.com/`.

[58] Brian F. Cooper et al. "Benchmarking cloud serving systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud computing (SoCC)*. 2010.

[59] Jonathan Corbet. *Linux Kernel Development Report*. Tech. rep. Linux foundation, 2017.

[60] Jonathan Corbet. *The multiqueue block layer*. `https://lwn.net/Articles/552904/`. Last accessed: Oct, 2020. 2013.

[61] *CoreOS*. `https://coreos.com/`.

[62] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. 2016.

[63] R. J. Creasy. "The Origin of the VM/370 Time-sharing System". In: *IBM J. Res. Dev.* (1981).

[64] Raslan Darawsheh. "mbuf External Buffer and Usage Examples". In: *Proceedings of the DPDK Userspace, Dublin*. 2018.

[65] Tom Woller David Kaplan Jeremy Powell. *AMD memory encryption*. Tech. rep. AMD, 2016.

[66] *Deploy Python Lambda functions with .zip file archives*. `https://docs.aws.amazon.com/lambda/latest/dg/python-package.html`.

[67] Digitalocean. *How to Regain Access to Droplets using the Recovery Console*. https://docs.digitalocean.com/products/droplets/resources/recovery-console/. 2021.

[68] *dm-crypt/device encryption*. Last accessed: Oct, 2020. URL: %5Curl%7Bhttps://wiki.archlinux.org/index.php/dm-crypt%7D.

[69] *Docker*. https://www.docker.com/.

[70] *Docker Repositories*. https://hub.docker.com/explore/.

[71] *Docker Slim*. https://github.com/docker-slim/docker-slim.

[72] *Docker Swarm*. https://www.docker.com/products/docker-swarm.

[73] *Docker switch to Alpine Linux*. https://news.ycombinator.com/item?id=11000827. 2016.

[74] Linux kernel documentation. *Seccomp BPF (SECure COMPuting with filters)*. https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html. 2021.

[75] Brendan Dolan-Gavitt et al. "Virtuoso: Narrowing the semantic gap in virtual machine introspection". In: *2011 IEEE symposium on security and privacy*. 2011.

[76] Jason A. Donenfeld. *WireGuard: Next Generation Kernel Network Tunnel*. https://www.wireguard.com/papers/wireguard.pdf. Last accessed: Oct, 2020.

[77] *Data Plane Development Kit (DPDK)*. Last accessed: Oct, 2020. URL: http://www.dpdk.org.

[78] Lian Du, Renyu Yang Tianyu Wo, and Chunming Hu. "Cider: A Rapid Docker Container Deployment System through Sharing Network Storage". In: *Proceedings of the 19th International Conference on High Performance Computing and Communications (HPCC)*. 2017.

[79] Eric Dumazet. "Busy Polling: Past, Present, Future". In: *netdev 2.1 Montreal*. 2017.

[80] Paul Emmerich et al. "The Case for Writing Network Drivers in High-Level Programming Languages". In: *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2019)*. Sept. 2019.

[81] D. R. Engler, M. F. Kaashoek, and J. O'Toole. "Exokernel: An Operating System Architecture for Application-Level Resource Management". In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*. 1995.

[82] *File system regression test on linux implemented for all major filesystems*. https://kernel.googlesource.com/pub/scm/fs/ext2/xfstests-bld/+/HEAD/Documentation/what-is-xfstests.md.

[83] Firecracker contributors. *firecracker-containerd*. https://github.com/firecracker-microvm/firecracker-containerd. 2021.

[84] Firecracker contributors. *Firecracker kernel configuration*. https://github.com/firecracker-microvm/firecracker/blob/main/resources/microvm-kernel-x86_64.config. 2021.

[85] Rodrigo Fonseca et al. "X-trace: A pervasive network tracing framework". In: *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*. USA: USENIX Association, 2007, p. 20.

[86] Cloud Native computing foundation. *Containerd – An industry-standard container runtime with an emphasis on simplicity, robustness and portability*. https://containerd.io/. 2021.

[87] Openstack Foundation. *Openstack: Open source cloud computing infrastructure*. https://www.openstack.org/. 2021.

[88] Yangchun Fu and Zhiqiang Lin. "Exterior: Using a dual-vm based external shell for guest-os introspection, configuration, and recovery". In: *Acm Sigplan Notices* 48.7 (2013), pp. 97–110.

[89] Yangchun Fu, Junyuan Zeng, and Zhiqiang Lin. "HYPERSHELL: A Practical Hypervisor Layer Guest OS Shell for Automated In-VM Management". In: *USENIX Annual Technical Conference (USENIX ATC)*. 2014.

[90] *fuse(7) Linux User's Manual*. Sept. 2017.

[91] Tal Garfinkel, Mendel Rosenblum, et al. "A virtual machine introspection based architecture for intrusion detection." In: *Ndss*. Vol. 3. San Diego, California, USA: Citeseer, 2003, pp. 191–206.

[92] Google. *Container analysis and vulnerability scanning*. https://cloud.google.com/container-registry/docs/container-analysis. 2021.

[93] Google. *Google OS Config Agent*. https://github.com/GoogleCloudPlatform/osconfig. 2021.

[94] Google. *Guest Agent for Google Compute Engine*. https://github.com/GoogleCloudPlatform/guest-agent. 2021.

[95] Google. *Homepage of crosvm*. https://chromium.googlesource.com/chromiumos/platform/crosvm/. 2021.

[96] Google. *Installing the guest environment*. https://github.com/GoogleCloudPlatform/osconfig. 2021.

[97] Google. *Nested virtualization overview*. https://cloud.google.com/compute/docs/instances/nested-virtualization/overview. 2021.

[98] *Google Cloud Functions*. https://cloud.google.com/functions/.

[99] *Google Compute Cloud Containers*. https://cloud.google.com/compute/docs/containers/.

[100] *Introducing Google Cloud Confidential Computing with Confidential VMs*. https://cloud.google.com/blog/products/identity-security/introducing-google-cloud-confidential-computing-with-confidential-vms. Last accessed: Oct, 2020. URL: https://cloud.google.com/blog/products/identity-security/introducing-google-cloud-confidential-computing-with-confidential-vms.

[101] *Google: 'EVERYTHING at Google runs in a container'*. https://www.theregister.co.uk/2014/05/23/google_containerization_two_billion/.

[102] Johannes Götzfried et al. "Cache Attacks on Intel SGX". In: *Proceedings of the 10th European Workshop on Systems Security*. 2017.

[103] Zhongshu Gu et al. "Process implanting: A new active introspection framework for virtualization". In: *2011 IEEE 30th International Symposium on Reliable Distributed Systems*. Madrid, Spain: IEEE, 2011, pp. 147–156.

[104] Marcus Hähnel, Weidong Cui, and Marcus Peinado. "High-Resolution Side Channels for Untrusted Operating Systems". In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. 2017.

[105] Tyler Harter et al. "Slacker: Fast Distribution with Lazy Docker Containers". In: *14th USENIX Conference on File and Storage Technologies (FAST)*. 2016.

[106] Gernot Heiser and Kevin Elphinstone. "L4 microkernels: The lessons from 20 years of research and deployment". In: *ACM Transactions on Computer Systems (TOCS)* 34.1 (2016), pp. 1–29.

[107] Scott Hendrickson et al. "Serverless Computation with OpenLambda". In: *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. 2016.

[108] Stephen Herwig, Christina Garman, and Dave Levin. "Achieving Keyless CDNs with Conclaves". In: *29th USENIX Security Symposium (USENIX Security)*. 2020.

[109] Hetzner AG. *Hetzner Rescue System*. https://docs.hetzner.com/robot/dedicated-server/troubleshooting/hetzner-rescue-system/. 2021.

[110] *High-speed packet processing framework*. https://github.com/ntop/PF_RING.

[111] *Homepage of SELinux*. https://selinuxproject.org/page/Main_Page.

[112] Michio Honda et al. "PASTE: A Network Programming Interface for Non-Volatile Main Memory". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2018.

[113] *Tracking Page Weight Over Time*. https://discuss.httparchive.org/t/tracking-page-weight-over-time/1049. Last accessed: Oct, 2020.

[114] Jaehyun Hwang et al. "TCP = RDMA: CPU-efficient Remote Storage Access with i10". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2020.

[115] IBM. *Getting started with KVM*. https://www.ibm.com/docs/en/cic/1.1.3?topic=SSLL2F_1.1.3/com.ibm.cloudin.doc/overview/Getting_started_tutorial.html. 2021.

[116] IBM. *IBM's Vulnerability Advisor*. https://www.ibm.com/docs/en/cloud-private/3.2.0?topic=guide-vulnerability-advisor. 2021.

[117] *IBM OpenWhisk*. https://www.ibm.com/cloud/functions.

[118] Intel. *Intel Trusted Execution Technology (Intel TXT)*. https://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf. 2017. URL: https://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf.

[119] *Intel Clear Containers*. https://clearlinux.org/containers.

[120] *Intel SGX SDK*. Last accessed: Oct, 2020. URL: %5Curl%7Bhttps://github.com/intel/linux-sgx%7D.

[121] *Intel Storage Performance Development Kit*. http://www.spdk.io. Last accessed: Oct, 2020. URL: %5Curl%7Bhttp://www.spdk.io%7D.

[122] *Intel Software Guard Extensions (Intel SGX)*. https://software.intel.com/en-us/sgx. Last accessed: Oct, 2020.

[123] *iPerf - The ultimate speed test tool for TCP, UDP and SCTP*. https://iperf.fr/. Last accessed: Oct, 2020. URL: %5Curl%7Bhttps://iperf.fr/%7D.

[124] *Intelligent Platform Management Interface Specification v2.0 rev. 1.1*. Last accessed: Dec, 2021. URL: %5Curl%7Bhttps://www.intel.de/content/www/de/de/products/docs/servers/ipmi/ipmi-second-gen-interface-spec-v2-rev1-1.html%7D.

[125] Yeongjin Jang, Sangho Lee, and Taesoo Kim. "Breaking Kernel Address Space Layout Randomization with Intel TSX". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 380–392.

[126] Jens Axboe. *Flexible I/O Tester*. https://github.com/axboe/fio. Last accessed: Dec, 2020. 2021.

[127] Eun Young Jeong et al. "MTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementationi (NSDI)*. 2014.

[128] Peter Okelmann Jörg Thalheim. *Project site of vmsh*. https://github.com/Mic92/vmsh. 2021.

[129] Jörg Thalheim, Peter Okelmann, Harshavardhan Unnibhavi, Redha Gouicem, Pramod Bhatotia. "VMSH: Hypervisor-agnostic Guest Overlays for VMs". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. 2022.

[130] Jonathan Kaldor et al. "Canopy: An end-to-end performance tracing and analysis system". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017.

[131] Svilen Kanev et al. "Profiling a Warehouse-Scale Computer". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture(ISCA)*. 2015.

[132] Kata maintainers. *Kata container kernel configuration*. https://github.com/kata-containers/kata-containers/blob/main/tools/packaging/kernel/configs/x86_64_kata_kvm_4.14.x. 2021.

[133] Jeffrey Katcher. *PostMark: A New File System Benchmark*. Tech. rep. Network Appliance Inc., Oct. 1997.

[134] Antoine Kaufmann et al. "TAS: TCP Acceleration as an OS Service". In: *Proceedings of the Fourteenth EuroSys Conference (EuroSys)*. 2019.

[135] Kernel maintainers. *xfstests-dev*. https://git.kernel.org/pub/scm/fs/xfs/xfstests-dev.git/. 2021.

[136] Linux maintainers. *The Definitive KVM (Kernel-based Virtual Machine) API Documentation*. https://www.kernel.org/doc/html/latest/virt/kvm/api.html. 2021.

[137] Gerwin Klein et al. "seL4: Formal verification of an OS kernel". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. New York, NY, USA: Association for Computing Machinery, 2009, pp. 207–220.

[138] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. "ReFlex: Remote Flash ≡ Local Flash". In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2017.

[139] *Knative*. `https://knative.dev/`.

[140] Robert Krahn et al. "Pesos: Policy Enhanced Secure Object Store". In: *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*. 2018.

[141] FreeBSD maintainers. *ksyms – kernel symbol table interface*. `https://www.freebsd.org/cgi/man.cgi?query=ksyms&sektion=4&manpath=FreeBSD+8.0-RELEASE`. 2021.

[142] *Kubeless*. `https://kubeless.io/`.

[143] *Kubernetes*. `https://kubernetes.io/`.

[144] Kubernetes. *Ephemeral Containers*. `https://kubernetes.io/docs/concepts/workloads/pods/ephemeral-containers/`. 2021.

[145] Simon Kuenzer et al. "Unikraft: fast, specialized unikernels the easy way". In: *Proceedings of the Sixteenth European Conference on Computer Systems*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 376–394.

[146] Roland Kunkel et al. "TensorSCONE: A Secure TensorFlow Framework using Intel SGX". In: *CoRR* (2019).

[147] A. Langley. *rfc7539: ChaCha20 and Poly1305 for IETF Protocols*. `https://tools.ietf.org/html/rfc7539`. Last accessed: Oct, 2020.

[148] Dayeol Lee et al. "Keystone: an open framework for architecting trusted execution environments". In: *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*. 2020.

[149] Jaehyuk Lee et al. "Hacking in Darkness: Return-oriented Programming against Secure Enclaves". In: *26th USENIX Security Symposium (USENIX Security 17)*. 2017.

[150] Shih-Wei Li, John S. Koh, and Jason Nieh. "Protecting Cloud Virtual Machines from Hypervisor and Host Operating System Exploits". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA, USA: USENIX Association, 2019, pp. 1357–1374.

[151] Shih-Wei Li et al. "A Secure and Formally Verified Linux KVM Hypervisor". In: *2021 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, 2021, pp. 1782–1799.

[152] Joshua Lind et al. "Teechain: A Secure Payment Network with Asynchronous Blockchain Access". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. 2019.

[153] *Linux Containers*. `https://linuxcontainers.org/`.

[154] *Linux Kernel Virtual Machine (KVM)*. `https://www.linux-kvm.org/page/Main_Page`.

[155] Anil Madhavapeddy and David J Scott. "Unikernels: The rise of the virtual library operating system". In: *Communications of the ACM* 57.1 (2014), pp. 61–69.

[156] Anil Madhavapeddy et al. "Jitsu: Just-in-time summoning of unikernels". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USA: USENIX Association, 2015, pp. 559–573.

[157] Anil Madhavapeddy et al. "Unikernels: Library Operating Systems for the Cloud". In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2013.

[158] Anil Madhavapeddy et al. "Unikernels: Library operating systems for the cloud". In: *ACM SIGARCH Computer Architecture News* 41.1 (2013), pp. 461–472.

[159] shadow-utils maintainer. *chpasswd(8) shadow-utils manual*. Shadow maintainers. 2021.

[160] Kernel maintainers. *Kernel Virtual Machine (KVM)*. `https://www.linux-kvm.org/page/Main_Page`. 2021.

[161] Libvirt maintainers. *Virsh management user interface – domstats*. `https://www.libvirt.org/manpages/virsh.html#domstats`. 2021.

[162] Linux maintainers. *cgroups(7) Linux User's Manual*. Linux foundation. Sept. 2017.

[163] Linux maintainers. *namespaces(7) Linux User's Manual*. Linux foundation. June 2016.

[164] Linux maintainers. *pts(4) Linux Programmer's Manual*. Linux foundation. 2021.

[165] OpenBSD maintainers. *OpenSSH remote login client*. OpenBSD. 2021.

[166] Overlayfs maintainers. *Overlayfs FUSE implementation*. CNCF. 2021.

[167] QEMU maintainers. *QEMU-GA(8) QEMU Guest Agent manual*. QEMU. 2021.

[168] QEMU maintainers. *Vhost-user protocol*. https://qemu.readthedocs.io/en/latest/interop/vhost-user.html. 2021.

[169] Rust-vmm maintainers. *rust-vmm*. https://github.com/rust-vmm. 2021.

[170] Rust-vmm maintainers. *vmm-reference*. https://github.com/rust-vmm/vmm-reference. 2021.

[171] Systemd maintainers. *Systemd-sysext: Activates System Extention Images*. https://www.freedesktop.org/software/systemd/man/systemd-sysext.html. 2021.

[172] Filipe Manco et al. "My VM is Lighter (and Safer) Than Your Container". In: *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. 2017.

[173] *Manpage of systemd-nspawn*. https://www.freedesktop.org/software/systemd/man/systemd-nspawn.html.

[174] *Manual of AppArmor*. http://manpages.ubuntu.com/manpages/xenial/en/man7/apparmor.7.html.

[175] Ilias Marinos, Robert N.M. Watson, and Mark Handley. "Network Stack Specialization for Performance". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. 2014.

[176] Ilias Marinos et al. "Disk|Crypt|Net: Rethinking the Stack for High-Performance Video Streaming". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. 2017.

[177] A. Markettos et al. "Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals". In: Jan. 2019. DOI: 10.14722/ndss.2019.23194.

[178] M. Mesnier, G.R. Ganger, and E. Riedel. "Object-based storage". In: *IEEE Communications Magazine* 41.8 (2003).

[179] Microsoft. *CVE-2021-38647: Open Management Infrastructure Remote Code Execution Vulnerability*. https://msrc.microsoft.com/update-guide/vulnerability/CVE-2021-38647. 2021.

[180] Microsoft. *Open Management Infrastructure (OMI)*. https://github.com/microsoft/omi. 2021.

[181] Microsoft Azure. *Azure confidential computing*. https://azure.microsoft.com/en-us/solutions/confidential-compute/. Last accessed: Oct, 2020.

[182] Philipp Mieden and Philippe Partarrieu. *Performance analysis of KVM-based microVMs orchestrated by Firecracker and QEMU*. Tech. rep. University of Amsterdam, 2019.

[183] Richard Ta-Min, Lionel Litty, and David Lie. "Splitting Interfaces: Making Trust between Applications and Operating Systems Configurable". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. 2006.

[184] *mount(8) Linux User's Manual*. Sept. 2017.

[185] *Lightweight standard libc implementation*. `https://www.musl-libc.org/`. Last accessed: Oct, 2020.

[186] *MySQL*. `https://www.mysql.com/`. Last accessed: Oct, 2020. URL: `%5Curl%7Bhttps://www.mysql.com/%7D`.

[187] Mihir Nanavati, Jake Wires, and Andrew Warfield. "Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2017.

[188] *Nginx Web Server*. `https://www.nginx.com/`. Last accessed: Oct, 2020. URL: `%5Curl%7Bhttps://www.nginx.com/%7D`.

[189] Bogdan Nicolae et al. "Going Back and Forth: Efficient Multideployment and Multisnapshotting on Clouds". In: *Proceedings of the 20th International Symposium on High Performance Distributed Computing*. San Jose, California, USA: Association for Computing Machinery, 2011.

[190] *Nomad*. `https://www.nomadproject.io/`.

[191] *nsenter*. `https://github.com/jpetazzo/nsenter`.

[192] O. Purdila and L. A. Grijincu and N. Tapus. "LKL: The Linux kernel library". In: *9th RoEduNet IEEE International Conference*. 2010.

[193] *Off-CPU Flame Graphs*. Last accessed: Oct, 2020. URL: `%5Curl%7Bhttp://www.brendangregg.com/FlameGraphs/offcpuflamegraphs.html%7D`.

[194] Peter Okelmann and Jø"rg Thalheim. *lambda-pirate*. `https://github.com/pogobanane/lambda-pirate`. 2021.

[195] *OpenFaaS*. `https://www.openfaas.com/`.

[196] Oracle. *Oracle Virtualization*. `https://www.oracle.com/virtualization/`. 2021.

[197] Meni Orenbach et al. “Eleos: ExitLess OS services for SGX enclaves”. In: *Proceedings of the 12th ACM European ACM Conference in Computer Systems (EuroSys)*. 2017.

[198] *Our fork of Docker Slim used for evaluation*. `https://github.com/Mic92/docker-slim/tree/cntr-eval`.

[199] *Our fork the nix rust library*. `https://github.com/Mic92/cntr-nix`.

[200] Bryan D Payne et al. “Lares: An architecture for secure active monitoring using virtualization”. In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. Oakland, CA, USA: IEEE, 2008, pp. 233–247.

[201] Simon Peter et al. “Arrakis: The Operating System is the Control Plane”. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014.

[202] Jonas Pfoh, Christian Schneider, and Claudia Eckert. “A formal model for virtual machine introspection”. In: *Proceedings of the 1st ACM workshop on Virtual machine security*. New York, NY, USA: Association for Computing Machinery, 2009, pp. 1–10.

[203] Michael Larabel. *Homepage of Phoronix test suite*. `https://www.phoronix-test-suite.com/`. 2021.

[204] Rishabh Poddar et al. “SafeBricks: Shielding Network Functions in the Cloud”. In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2018.

[205] Donald E. Porter et al. “Rethinking the Library OS from the Top Down”. In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2011.

[206] Mary C Potter et al. “Detecting meaning in RSVP at 13 ms per picture”. In: *Attention, Perception, & Psychophysics* 76.2 (2014), pp. 270–279.

[207] C. Priebe, K. Vaswani, and M. Costa. “EnclaveDB: A Secure Database using SGX (S&P)”. In: *IEEE Symposium on Security and Privacy*. 2018.

[208] Christian Priebe et al. *SGX-LKL: Securing the Host OS Interface for Trusted Execution*. 2019. eprint: `arXiv:1908.11143`.

[209] Project Zero. *An EPYC escape: Case-study of a KVM breakout*. `https://googleprojectzero.blogspot.com/2021/06/an-epyc-escape-case-study-of-kvm.html`. 2021.

[210] Michael Larabel. *Wiki page for the Phoronix disk test suite*. `https://openbenchmarking.org/suite/pts/disk`. 2021.

[211] Qemu maintainers. *QEMU - 'microvm' virtual platform (microvm)*. https://qemu.readthedocs.io/en/latest/system/i386/microvm.html. 2021.

[212] Qemu maintainers. *Homepage of qemu*. https://www.qemu.org/. 2021.

[213] Qemu wiki authors. *Documentation 9psetup*. https://wiki.qemu.org/Documentation/9psetup. 2021.

[214] Qemu maintainers. *QEMU version 4.2.0 released*. https://www.qemu.org/2019/12/13/qemu-4-2-0/. 2021.

[215] D. P. Quigley et al. "UnionFS: User- and Community-oriented Development of a Unification Filesystem". In: *Proceedings of the 2006 Linux Symposium (OLS)*. 2006.

[216] Avi Qumranet et al. "KVM: The Linux virtual machine monitor". In: *Proceedings Linux Symposium* 15 (2007).

[217] *RancherOS*. https://rancher.com/rancher-os/.

[218] *Raw benchmark report generated by phoronix test suite*. https://openbenchmarking.org/result/1802024-AL-CNTREVALU05.

[219] Red Hat Customer Portal. *CVE-2015-3456*. https://access.redhat.com/security/cve/CVE-2015-3456. 2021.

[220] *Redfish standard page*. Last accessed: Dec, 2021. URL: %5Curl%7Bhttps://www.dmtf.org/standards/redfish%7D.

[221] *Redis*. https://redis.io/. Last accessed: Oct, 2020. URL: %5Curl%7Bhttps://redis.io/%7D.

[222] RISC-V. *Keystone Open-source Secure Hardware Enclave*. https://keystone-enclave.org/. Last accessed: Oct, 2020. URL: https://keystone-enclave.org/.

[223] Luigi Rizzo. "Revisiting network I/O APIs: The Netmap Framework". In: *Communications of the ACM* (2012).

[224] *Root cause analysis in unprivileged nspawn container with cntr*. https://github.com/systemd/systemd/issues/6244#issuecomment-356029742.

[225] Rusty Russell. "Virtio: Towards a de-Facto Standard for Virtual I/O Devices". In: *SIGOPS Oper. Syst. Rev.* 42.5 (July 2008), pp. 95–103. ISSN: 0163-5980. DOI: 10.1145/1400097.1400108. URL: https://doi.org/10.1145/1400097.1400108.

[226] *Rust library for filesystems in userspace*. `https://github.com/zargony/rust-fuse`.

[227] *Rust library that wraps around the Linux/Posix API*. `https://github.com/nix-rust/nix`.

[228] Raja R Sambasivan et al. *So, you want to trace your distributed system? Key design insights from years of practical experience*. Tech. rep. Carnegie Mellon University, 2014.

[229] Nuno Santos, Krishna P. Gummadi, and Rodrigo Rodrigues. "Towards Trusted Cloud Computing". In: *Proceedings of the 1st USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. 2009.

[230] Dan Schatzberg et al. "EbbRT: A Framework for Building Per-Application Library Operating Systems". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016.

[231] Felix Schuster et al. "VC3 : Trustworthy Data Analytics in the Cloud using SGX". In: *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*. 2015.

[232] *SCONE File Protection*. `https://sconedocs.github.io/SCONE_Fileshield/`. Last accessed: Oct, 2020.

[233] *Serverless Architectures*. `https://martinfowler.com/articles/serverless.html#ReducedOperationalCost`.

[234] Monirul I Sharif et al. "Secure in-vm monitoring using hardware virtualization". In: *Proceedings of the 16th ACM conference on Computer and communications security*. New York, NY, USA: Association for Computing Machinery, 2009, pp. 477–487.

[235] Prateek Sharma et al. "Containers and Virtual Machines at Scale: A Comparative Study". In: *Proceedings of the 17th International Middleware Conference*. Middleware '16. Trento, Italy: Association for Computing Machinery, 2016.

[236] Prateek Sharma et al. "Containers and Virtual Machines at Scale: A Comparative Study". In: *Proceedings of the 17th International Middleware Conference (Middleware)*. 2016.

[237] Shweta Shinde et al. "PANOPLY: Low-TCB Linux Applications with SGX Enclaves". In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2017.

[238] Benjamin H. Sigelman et al. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. Google, Inc., 2010.

[239] Simon Sharwood. *AWS adopts home-brewed KVM as new hypervisor*. https://www.theregister.com/2017/11/07/aws_writes_new_kvm_based_hypervisor_to_make_its_cloud_go_faster/. 2021.

[240] Sergei Skorobogatov. *Low temperature data remanence in static RAM*. Tech. rep. UCAM-CL-TR-536. University of Cambridge, Computer Laboratory, June 2002. DOI: 10.48456/tr-536. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-536.pdf.

[241] Livio Soares and Michael Stumm. "FlexSC: Flexible System Call Scheduling with Exception-less System Calls". In: *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2010.

[242] *speedtest*. https://www.sqlite.org/speed.html. Last accessed: Oct, 2020.

[243] SQLite Consortium. *SQLite*. https://www.sqlite.org/. Last accessed: Oct, 2020. 2021. URL: %5Curl%7Bhttps://www.sqlite.org/%7D.

[244] Stefan Hajnoczi. *Proposal for MMIO/PIO dispatch file descriptors*. https://www.spinics.net/lists/kvm/msg208139.html. 2020.

[245] *sysbench*. https://github.com/akopytov/sysbench. Last accessed: Oct, 2020.

[246] Taesoo Kim. "SGX 101: The very first place to study Intel SGX." In: (2019).

[247] Tarik Taleb et al. "On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration". In: *IEEE Communications Surveys Tutorials* 19.3 (2017), pp. 1657–1681. DOI: 10.1109/COMST.2017.2705720.

[248] Jianfeng Tan et al. "VIRTIO-USER: A new versatile channel for kernel-bypass networks". In: *Proceedings of the Workshop on Kernel-Bypass Networks*. New York, NY, USA: Association for Computing Machinery, 2017, pp. 13–18.

[249] J. Thalheim, P. Bhatotia, and C. Fetzer. "INSPECTOR: Data Provenance Using Intel Processor Trace (PT)". In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. 2016.

[250] Jörg Thalheim. *Project site of rkt-io*. https://github.com/Mic92/rkt-io. 2021.

[251] Jörg Thalheim et al. "Cntr: Lightweight OS Containers". In: *2022 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2022, pp. 199–212.

[252] Jörg Thalheim et al. “Rkt-IO: A Direct I/O Stack for Shielded Execution”. In: *Proceedings of the Sixteenth European Conference on Computer Systems*. 2021.

[253] Jörg Thalheim et al. “Sieve: Actionable Insights from Monitored Metrics in Distributed Systems”. In: *Proceedings of Middleware Conference (Middleware)*. 2017.

[254] Jörg Thalheim et al. “Sieve: Actionable insights from monitored metrics in distributed systems”. In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. New York, NY, USA: Association for Computing Machinery, 2017, pp. 14–27.

[255] *Toolbox - is a small script that launches a container to let you bring in your favorite debugging or admin tools.* `https://github.com/coreos/toolbox`.

[256] Bohdan Trach et al. “ShieldBox: Secure Middleboxes using Shielded Execution”. In: *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*. 2018.

[257] Trusted Computing Group. *TPM Main Specification*. `https://trustedcomputinggroup.org/tpm-main-specification`. Last accessed: Oct, 2020. 2011. URL: `https://trustedcomputinggroup.org/tpm-main-specification/`.

[258] Chia-Che Tsai, Donald E Porter, and Mona Vij. “Graphene-SGX: A practical library OS for unmodified applications on SGX”. In: *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. 2017.

[259] Chia-Che Tsai et al. “Cooperation and Security Isolation of Library OSes for Multi-process Applications”. In: *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*. 2014.

[260] Chia-Che Tsai et al. “Cooperation and security isolation of library OSes for multi-process applications”. In: *Proceedings of the Ninth European Conference on Computer Systems*. New York, NY, USA: Association for Computing Machinery, 2014, pp. 1–14.

[261] Michael S. Tsirkin and Cornelia Huck. “Virtual I/O Device (VIRTIO) Version 1.1”. In: *OASIS Committee Specification 01* 1.1 (2019). Latest version: `https://docs.oasis-open.org/virtio/virtio/v1.1/cs01/virtio-v1.1-cs01.html`, p. 1.

[262] *Twelve-Factor App - Manifesto mainained by Heroku on how to build modern web-application*. `https://12factor.net/config`.

[263] The Regents of the University of California. *Homepage of IOR*. https://ior.readthedocs.io/en/latest/. 2021.

[264] Dmitrii Ustiugov et al. "Benchmarking, Analysis, and Optimization of Serverless Function Snapshots". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*. New York, NY, USA: ACM, 2021, pp. 559–572. DOI: 10.1145/3445814.3446714.

[265] Jo Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*. 2018.

[266] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. "To FUSE or Not to FUSE: Performance of User-Space File Systems". In: *15th USENIX Conference on File and Storage Technologies (FAST 17)*. 2017.

[267] Arjan van de Ven. *An introduction to Clear Containers*. https://lwn.net/Articles/644675/. 2015.

[268] Vlad Krasnov. *How "expensive" is crypto anyway*. https://blog.cloudflare.com/how-expensive-is-crypto-anyway. Last accessed: Oct, 2020. 2017.

[269] VMware. *VMware ESXi: The Purpose-Built Bare Metal Hypervisor*. https://www.vmware.com/products/esxi-and-esx.html. 2021.

[270] Sebastian Vogl et al. "X-tier: Kernel module injection". In: *International Conference on Network and System Security*. Springer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 192–205.

[271] *Website of the lxc container engine*. https://linuxcontainers.org/.

[272] Weichbrodt, Nico and Kurmus, Anil and Pietzuch, Peter and Kapitza, Rüdiger. "AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves". In: *Computer Security – ESORICS*. 2016.

[273] *What Technology Change Enables 1 Terabyte (TB) Enclave Page Cache (EPC) size in 3rd Generation Intel® Xeon® Scalable Processor Platforms?* https://www.intel.com/content/www/us/en/support/articles/000059614/software/intel-security-products.html.

[274] Ric Wheeler. *Homepage of fs_mark*. https://sourceforge.net/projects/fsmark/. 2021.

[275] Will Deacon. *Homepage of kvmtool*. https://github.com/kvmtool/kvmtool. 2021.

[276] WMware. *VMware vCloud*. `https://www.vmware.com/products/vrealize-suite-vcloud-suite.html`. 2021.

[277] *wrk*. `https://github.com/wg/wrk`. Last accessed: Oct, 2020.

[278] Kernel maintainers. *What is xfstests?* `https://kernel.googlesource.com/pub/scm/fs/ext2/xfstests-bld/+/HEAD/Documentation/what-is-xfstests.md`. 2021.

[279] Yuping Xing and Yongzhao Zhan. "Virtualization and cloud computing". In: *Future Wireless Networks and Information Systems*. Springer, 2012, pp. 305–312.

[280] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-channel attacks: Deterministic side channels for untrusted operating systems". In: *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*. 2015.

[281] Ziye Yang et al. "Spdk vhost-nvme: Accelerating i/os in virtual machines on nvme ssds via user space vhost target". In: *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*. Paris, France: IEEE, 2018, pp. 67–76.

[282] Kenichi Yasukata et al. "StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs". In: *2016 USENIX Annual Technical Conference (USENIX ATC)*. 2016.

[283] Tom Yates. *Linux kernel: Introduction of hybrid polling in the blk-mq subsystem*. `https://lwn.net/Articles/735275`. Last accessed: Oct, 2020. 2017.

[284] Irene Zhang et al. "I'm Not Dead Yet! The Role of the Operating System in a Kernel-Bypass Era". In: *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*. 2019.

[285] Wenting Zheng et al. "Opaque: An Oblivious and Encrypted Distributed Analytics Platform". In: *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2017.