VMSH: Hypervisor-agnostic Guest Overlays for VMs

Jörg Thalheim[1,2], Peter Okelmann[1], Harshavardhan Unnibhavi[1], Redha Gouicem[1], Pramod Bhatotia[1]
[1] Technical University of Munich, Germany
[2] The University of Edinburgh, UK

Abstract

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.

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 [1, 117]. To enable virtualisation, the Linux kernel-based virtual machine (KVM) [67] is the de facto mechanism in the cloud since it uses hardware acceleration to enforce compartmentalisation [41, 45, 82, 100].

With an increased demand to support performance-critical workloads, there is a significant thrust towards designing lightweight VM solutions to minimise the virtualisation overheads [1, 20, 75, 89]. These solutions provide reduced memory footprints and fast boot up times [78], which makes them suitable for increasingly popular deployment models, such as serverless [12, 109]. Furthermore, lightweight VMs improve dependability properties since they strive to minimise the trusted and reliable computing base [75].

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 [7, 37, 38, 77] and tracing libraries [4] which again counteract advantages of lightweight VMs such as their dependability properties (§ 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

EuroSys '22, April 05–08, 2022, Rennes, France

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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, performance, and effectiveness. Lastly, we evaluate three use-cases. For robustness, we run the `xfstests` [55] suite and show that VMSH's block device does not have any regressions compared to the QEMU implementation (§ 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 (§ 6.2). We measure performance with the Phoronix Test Suite [60] and `fio` [48], and find no slowdown of the guest while VMSH is attached (§ 6.3). For effectiveness, we strip popular Docker images on Docker Hub [25] from unused files such as user tools which might become obsolete through VMSH (§ 6.4). With an average size reduction of 60%, VMSH promises to ease adoption of lightweight VMs by allowing their on-demand attachment. Finally, we implement three real-world use-cases that make VMSH a key element in cloud infrastructures (§ 6.5).

Our contributions can be summarised as follows:

- We propose an abstraction which allows to extend lightweight VMs at run time independently of the guest and hypervisor (§ 2.2). This enables lighter VMs by removing tools from VM images while still being able to attach them back to the VM on-demand (§ 2.3).
- 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).
- We implement (§ 5) and evaluate (§ 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.

2 Background and Motivation

2.1 Background

We first provide a brief background on KVM and virtIO to understand the design details of VMSH.

Kernel-based virtual machine (KVM). Hardware-assisted virtualisation uses capabilities on host processors to enable efficient full virtualisation. Full virtualisation emulates the complete hardware environment to allow running an unmodified guest OS that uses the same instruction set as the host machine. Kernel-based virtual machine (KVM) is a kernel API for Linux (also FreeBSD and Illumos) that provides an abstraction layer on top of hardware-assisted virtualisation capabilities of different CPU architectures. KVM is the default virtualisation API used by major cloud providers [41, 45, 82, 100].

The actual program that runs the guest OS, called the hypervisor, is implemented in the userspace and uses KVM. KVM hypervisors include QEMU [88], Firecracker [1], Cloud Hypervisor [20] and crosvm [39]. The hypervisor sets up the initial CPU and memory and emulates the devices (*e.g.* block, console, NICs). VMSH attaches to VMs mainly targeting KVM.

VirtIO. Emulating physical hardware is slow and causes significant overheads compared to the native execution on a real hardware. Thus, most paravirtualised hypervisors rely on devices based on the VirtIO standard to improve the performance and simplify the interface between the hypervisor and the guest. VirtIO defines a common interface for VM-optimised device emulation (network devices, block devices, etc.) [94, 108]. Most hypervisors implement VirtIO devices and their guest drivers exist for all major OSes. Depending on the device type, VirtIO specifies a number of consumer/producer virtqueues in shared memory, which the device in the hypervisor and the driver in the guest use to exchange data. VirtIO has two major transport mechanisms, based on either memory mapped IO (MMIO) or on the PCI standard. In VMSH, we implement the MMIO variant, which is more widespread, especially in microVMs [1, 29, 52, 90].

2.2 Motivation: The Missing Abstraction in VMs

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.

2.3 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 § 6.5).

Dependability services. Cloud customers tend to have a wide range of distributions and versions installed [8]. 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 [24, 44].
- *Monitoring tools* are currently used to gather coarse-grained information about the resource usage of the entire guest [69]. 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 [5, 36, 46]. VMSH enables similar techniques to track and update packages in the VM space.

De-bloat VM images. VMSH allows to build lightweight VMs [13, 92, 110, 113] 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.

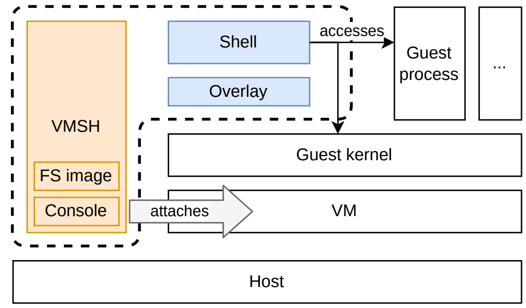


Figure 1. A user attaches a custom file system image to a VM and starts a shell from the image using VMSH. (*Orange refers to VMSH components running on the host and blue to the ones in the guest.*)

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 [6] and logging information from the application itself [3]. 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.

3 Overview

3.1 System Overview

To realise the *vm-exec* abstraction for VMs (Section 2.2), 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 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 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 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.

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 [87], 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 [93], as they contain complex device implementations that contribute to a relatively large attack surface. In Section 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 [10] and of the hypervisor [62] 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 § 4.5).

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 [16, 114]. 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 § 6).

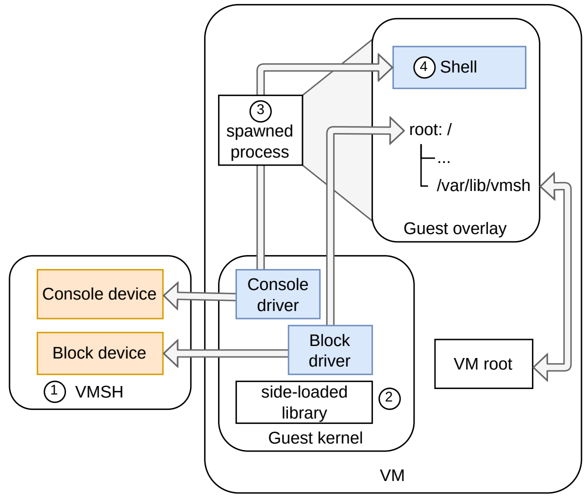


Figure 2. VMSH sets up its devices in the guest by side-loading a kernel library. The virtual block device backs the overlay's root. The virtual console handles console inputs/outputs of the spawned process.

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 § 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 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 [39] only has the former whereas Firecracker [1] and kvmtool [116] lack both. In other cases, such features, even when supported by hypervisors, are obscured by orchestration frameworks, such as OpenStack [31] or Containerd [21].

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 § 2.1) without any help from the hypervisor (see § 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 § 4.2).

#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 `VMEXITs` 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 [101]. The (2.) queues themselves are read from the hypervisor memory via system calls. We describe the design of our hypervisor-independent VirtIO devices in § 4.3.

4 Design

To address the design challenges, we next describe how we load kernel code into the guest VM (§ 4.1) and techniques to analyse the guest memory to enable VMSH to load the kernel library (§ 4.2). We describe mechanisms to serve VirtIO devices (§ 4.3). Then, we explain the layout of our container-based system overlay (§ 4.4), and finally discuss the security implications (§ 4.5).

4.1 Hypervisor-agnostic Side-loading for VMs

As described in § 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

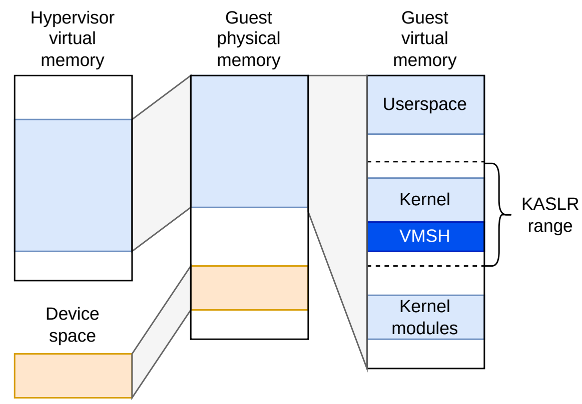


Figure 3. Address space mappings between hypervisor virtual memory and guest physical/virtual memory.

own address space (see Figure 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 § 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.

4.2 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 § 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 [47]. 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 [32]). 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 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 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.

4.3 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. 4/1.). VMSH's block device processes the request and enqueues the response into the other virtqueue (Fig. 4/2.). To indicate new requests in the queue, the guest driver also notifies the block device by writing to an MMIO register (Fig. 4/3.). 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 § 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 Container-based System Overlay

After setting up the devices, the kernel library spawns a userspace process in the guest (see Figure 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

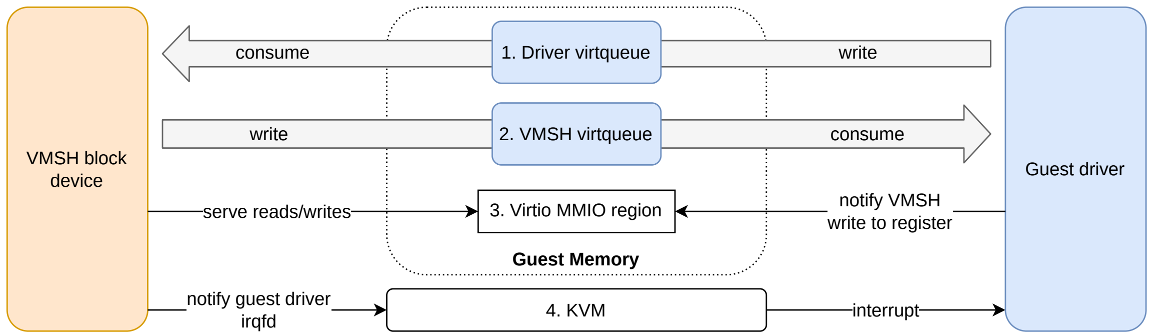


Figure 4. VMSH communication infrastructure based on the VirtIO protocol. Guest and host components share data through virtqueues (1, 2). Notification is performed through MMIO regions (3) and KVM (4).

process (UID, GID, Apparmor/Selinux profiles, namespaces, cgroups, capabilities) and applies the context to the newly established interactive shell.

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 § 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, because we do not require network access from the guest network, which mitigates remote code execution bugs [76].

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 § 4.1. 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 [111] 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 § 4.3. 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 § 6.3.

As outlined in § 4.3, VMSH traps accesses to MMIO addresses for device initialisation and driver updates (also see Figure 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 [101], 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 § 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*. Once started, the guest program will then setup the container-based system overlay that is described in § 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 [23]. 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.

6 Evaluation

We evaluate VMSH across the following dimensions: robustness (§ 6.1), generality (§ 6.2), performance (§ 6.3), and effectiveness (§ 6.4). Lastly, we evaluate three use-cases (§ 6.5).

Experiment setup. We perform our experiments on a machine with an Intel Core i9-9900K CPU with 8 cores (16 hyperthreads, 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.

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 [55] and block devices [56]. `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 1. Hypervisor and kernel support.

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 1).

Hypervisors. We develop VMSH using QEMU as the primary target. However, we expand our scope to the following KVM-based hypervisors: QEMU [88], kvmtool [116], Firecracker [1], crosvm [39], and Cloud Hypervisor [20].

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 [54]. 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 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.

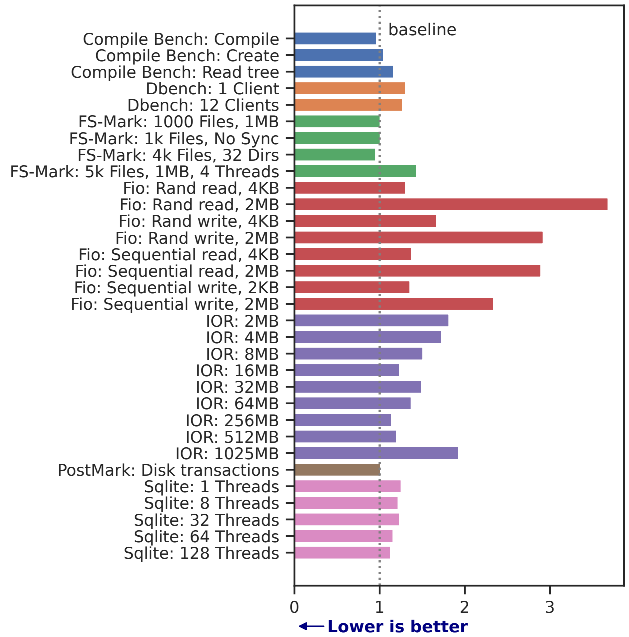


Figure 5. Relative performance of vmsh-blk for the Phoronix Test Suite compared to qemu-blk.

6.3 Performance

We evaluate VMSH's performance using a range of workloads: (A) the Phoronix test suite [60], (B) impact of attaching VMSH on the guest application's performance, (C) fio [48], 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 [61] of the Phoronix Test Suite [60]. This suite consists of Compile bench [19], DBENCH [9], fs_mark [115], Flexible I/O Tester [48], IOR [80], PostMark [53] and Sqlite [22]. 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 5. On average, VMSH is 1.5×±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× 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× slowdown compared to qemu-blk. In practice, the guest 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 [48] 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 small block sizes (4 KiB), thereby maximising per-access software overheads, and sequential accesses, to avoid hardware bottlenecks.

Figure 6a shows the throughput results of these experiments while Figure 6b 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 § 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 [91]). Results are also shown in Figure 6. `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.

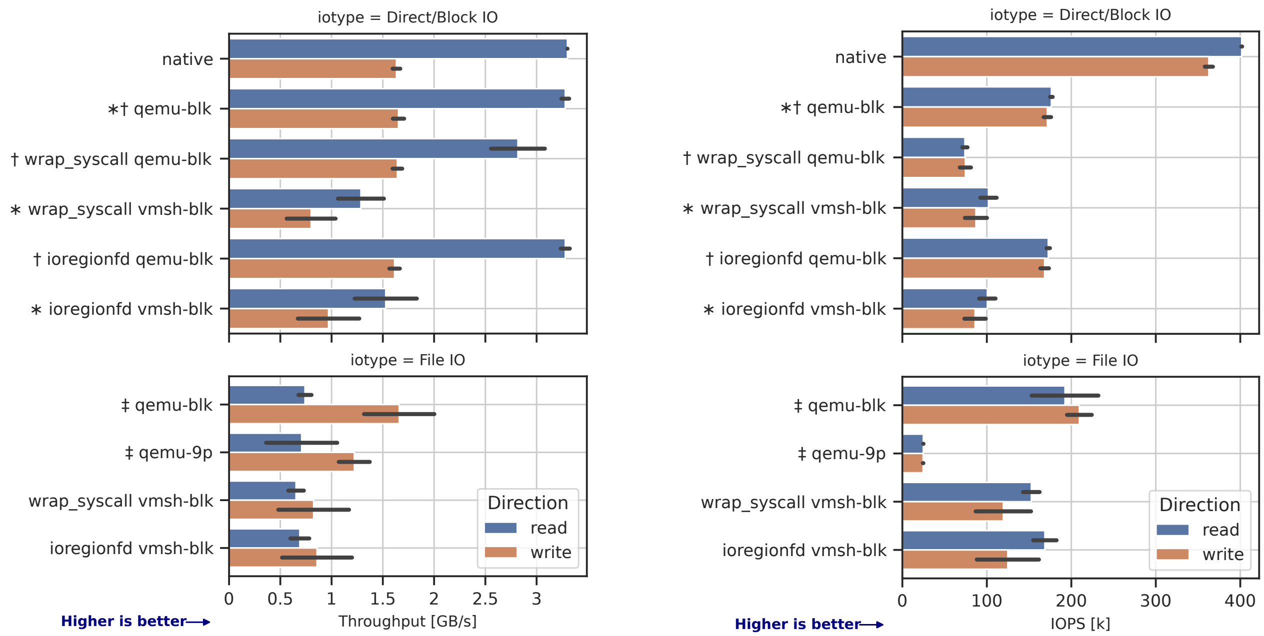
As for vmsh-blk, throughput and IOPS are halved compared to qemu-blk, indifferent to the used implementation (see results tagged with ∗). This degradation is expected since VMSH triggers more context switches than qemu-blk. IO operations are cooperatively handled by the guest driver running in the VM process and the VMSH virtual device running on the host (see Figure 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 § 6.5 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"* [86].

We measure the round-trip of a shell input by connecting one end of a pseudo-terminal seat (`pts`) [68] to a shell. We then use the other end to submit an echo command to the



(a) IO bandwidth/throughput. Best-case scenario.

(b) IO operations per second (IOPS). Worst case scenario.

Figure 6. `fio` with different configurations featuring qemu-blk and vmsh-blk with direct IO, and file IO with qemu-9p.

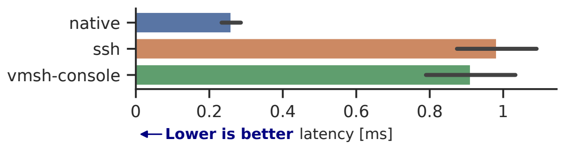


Figure 7. VMSH-console responsiveness compared to SSH.

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 7). The latency of the VMSH console is an order of magnitude faster than the capabilities of the human eye [86], making it sufficient for real life use cases.

6.4 Effectiveness

We evaluate the effectiveness of VMSH for building lightweight VMs by quantifying the reduction in VM image sizes when the virtualised infrastructure provider deploys only the core application—additional tools in the VM images are not necessary and can be loaded on demand thanks to our overlays.

Dataset: Docker Hub. We analyse the top 40 most downloaded official container images from Docker Hub [25].

Methodology. Using the Docker Hub dataset, we build lightweight VMs by identifying the files that are strictly required for the application, while removing the unnecessary files, *e.g.,* additional tools or services. In particular, we run each container image in a QEMU-based hypervisor [84]. In the guest's initial ramdisk, before the application starts, we add a custom system call tracer based on `sysdig` to record all paths opened by the VM [49]. Using this, we build a new minimal VM image containing only these files and check that the application still works.

Results. Figure 8 shows the distribution of the reduction in VM image sizes to build lightweight VMs. Image sizes are reduced by between 50% and 97%, on average by 60%. When analysing the files removed in the process, we observe that a number of tools are installed, including package managers, coreutils and shells. Only 3 of the 40 containers are reduced by less than 10%. We find that these containers are using a single statically linked Go executable instead of depending on OS images.

Note that since we are analysing container images instead of VM images, these results are conservative. In general, VM images package a higher number of tools that are unrelated to the application, compared to containers. This is due to the fact that VMs are harder to inspect, making it important to have tools that are pre-built into the image. We believe that with VMSH, we can enable an ecosystem where these tools could be pruned from the VM images and attached on demand at run time, thereby promoting lightweight VMs.

6.5 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

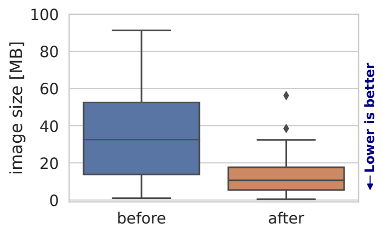


Figure 8. VM size reduction for the top-40 Docker images (average reduction: 60%).

their dependability properties [81]. 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 [95]. To help developers debug FaaS deployments, we provide them with an interactive shell in lambda-function instances. In particular, we integrate VMSH into vHive [109], a *knative*-compliant stack running serverless workloads in slim Firecracker-containerd VMs [1, 28]. 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 [24, 44]. 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` [97] 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 [5, 36, 46]. 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 [2] of known security vulnerabilities and report them.

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 [70] is typically used for interactive debugging. For tasks like automated management of updates, user accounts or configurations, cloud providers offer a multitude of agents [7, 37, 38, 40, 72, 77]. Agents are also used for distributed tracing in serverless environments [17, 30, 51, 99, 106]. According to Sambasivan et al., this variety is justified, as one size does not fit all use-cases [95]. 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 from the guests.

Virtualisation. Chen and Noble describe the problem of recovering high-level OS state from guest memory [18]. This 'semantic gap' has since been approached [35] and formalised [85]. Executing code inside a guest has been done by reusing userspace execution contexts [27, 42] or by injecting kernel modules [83, 98, 114], akin to VMSH's sideloader. Introspection usually aims at stealthiness and erases proofs of tampering the guest's execution [16, 34, 114].

To keep the host isolated from the guest, additional VMs are proposed to contain the inspection tool [33, 83]. 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 [105] 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 [58] can be used to deploy software, *e.g.,* an interactive debugging environment, into another pod. This approach is locked in to Kubernetes. Systemd-sysext [74] overlays file systems with extension images using overlayfs [71]. 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 [14, 15, 96, 107] reduce VM size by merging the kernel and user application into a single binary. Unikraft [59] combines the aspects of micro-library OSes and unikernels [64–66] 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 [11, 43, 57]. VMSH follows similar principles by offering essential functionality in a separate host process and guest overlay, acting upon IPC.

New hypervisors are built [1, 20, 39, 89], smaller and less complex, to reduce overheads [75, 78, 90] and attack surface. Many of them are written in memory-safe languages [1, 20, 39, 112], while others are formally verified [63]. Their miniaturisation is also advanced, as virtual devices are extracted into separate processes with vhost-user [73, 102, 118]. While vhost still requires modifications on the hypervisor side, VMSH does not and operates non-cooperatively.

8 Conclusion

In this work, we present 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.

Artifact. VMSH is publicly available as an open-source project along with the complete evaluation setup [50].

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434.

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

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

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

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

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

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

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

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

[10] Andy Honig and Nelly Porter. 2021. 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.

[11] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. 2016. 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. Association for Computing Machinery, New York, NY, USA, 189–203.

[12] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. 2017. Serverless computing: Current trends and open problems. In *Research advances in cloud computing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20.

[13] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 164–177.

[14] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R Lorch, Barry Bond, Reuben Olinsky, and Galen C Hunt. 2013. Composing OS extensions safely and efficiently with Bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems*. Association for Computing Machinery, New York, NY, USA, 239–252.

[15] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 335–348.

[16] Martim Carbone, Matthew Conover, Bruce Montague, and Wenke Lee. 2012. Secure and robust monitoring of virtual machines through guest-assisted introspection. In *International workshop on recent advances in intrusion detection*. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 22–41.

[17] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. 2002. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings International Conference on Dependable Systems and Networks*. IEEE, Washington, DC, USA, 595–604.

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

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

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

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

[22] SQLite Consortium. 2021. Homepage of SQLite. http://sqlite.org/.

[23] Jonathan Corbet. 2017. *Linux Kernel Development Report*. Technical Report. Linux foundation.

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

[25] Docker. 2021. Explore official Docker images. https://hub.docker.com/search?q=&type=image&image_filter=official.

[26] Docker. 2022. Docker homepage. https://www.docker.com/.

[27] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. 2011. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *2011 IEEE symposium on security and privacy*. IEEE, Oakland, California, 297–312.

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

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

[30] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. 2007. X-trace: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*. USENIX Association, USA, 20.

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

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

[33] Yangchun Fu and Zhiqiang Lin. 2013. Exterior: Using a dual-vm based external shell for guest-os introspection, configuration, and recovery. *Acm Sigplan Notices* 48, 7 (2013), 97–110.

[34] Yangchun Fu, Junyuan Zeng, and Zhiqiang Lin. 2014. HYPERSHELL: A Practical Hypervisor Layer Guest OS Shell for Automated In-VM Management. In *USENIX Annual Technical Conference (USENIX ATC)*. USENIX, Philadelphia, PA, 85–96.

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

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

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

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

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

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

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

[42] Zhongshu Gu, Zhui Deng, Dongyan Xu, and Xuxian Jiang. 2011. Process implanting: A new active introspection framework for virtualization. In *2011 IEEE 30th International Symposium on Reliable Distributed Systems*. IEEE, Madrid, Spain, 147–156.

[43] Gernot Heiser and Kevin Elphinstone. 2016. L4 microkernels: The lessons from 20 years of research and deployment. *ACM Transactions on Computer Systems (TOCS)* 34, 1 (2016), 1–29.

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

[45] IBM. 2021. 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.

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

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

[48] Jens Axboe. 2021. Flexible I/O Tester. https://github.com/axboe/fio.

[49] Jörg Thalheim. 2021. Runq fork with our modifications. https://github.com/Mic92/runq/commits/vmsh.

[50] Peter Okelmann Jörg Thalheim. 2022. Project page of vmsh. https://github.com/Mic92/vmsh.

[51] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. 2017. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 34–50.

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

[53] Jeffrey Katcher. 1997. *PostMark: A New File System Benchmark*. Technical Report. Network Appliance Inc.

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

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

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

[57] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. Association for Computing Machinery, New York, NY, USA, 207–220.

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

[59] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, et al. 2021. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*. Association for Computing Machinery, New York, NY, USA, 376–394.

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

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

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

[63] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. A Secure and Formally Verified Linux KVM Hypervisor. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 1782–1799.

[64] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. 2015. Jitsu: Just-in-time summoning of unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, USA, 559–573.

[65] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 461–472.

[66] Anil Madhavapeddy and David J Scott. 2014. Unikernels: the rise of the virtual library operating system. *Commun. ACM* 57, 1 (2014), 61–69.

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

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

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

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

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

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

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

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

[75] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 218–233.

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

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

[78] Philipp Mieden and Philippe Partarrieu. 2019. *Performance analysis of KVM-based microVMs orchestrated by Firecracker and QEMU*. Technical Report. University of Amsterdam.

[79] Maintainers of nix. 2022. Homepage of nix . https://nixos.org/download.html.

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

[81] Peter Okelmann and Jörg Thalheim. 2021. lambda-pirate. https://github.com/pogobanane/lambda-pirate.

[82] Oracle. 2021. Oracle Virtualization. https://www.oracle.com/virtualization/.

[83] Bryan D Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. 2008. Lares: An architecture for secure active monitoring using virtualization. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, Oakland, CA, USA, 233–247.

[84] Peter Morjan. 2021. Runq - a hypervisor-based Docker runtime. https://github.com/gotoz/runq.

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

[86] Mary C Potter, Brad Wyble, Carl Erick Hagmann, and Emily S McCourt. 2014. Detecting meaning in RSVP at 13 ms per picture. *Attention, Perception, & Psychophysics* 76, 2 (2014), 270–279.

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

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

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

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

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

[92] Avi Qumranet, Yaniv Qumranet, Dor Qumranet, Uri Qumranet, and Anthony Liguori. 2007. KVM: The Linux virtual machine monitor. *Proceedings Linux Symposium* 15 (2007).

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

[94] Rusty Russell. 2008. Virtio: Towards a de-Facto Standard for Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.* 42, 5 (July 2008), 95–103. https://doi.org/10.1145/1400097.1400108

[95] Raja R Sambasivan, Rodrigo Fonseca, Ilari Shafer, and Gregory R Ganger. 2014. *So, you want to trace your distributed system? Key design insights from years of practical experience.* Technical Report. Carnegie Mellon University.

[96] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. 2016. Ebbrt: A framework for building per-application library operating systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (SDI 16)*. USENIX Association, USA, 671–688.

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

[98] Monirul I Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. 2009. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM conference on Computer and communications security*. Association for Computing Machinery, New York, NY, USA, 477–487.

[99] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Technical Report. Google, Inc.

[100] Simon Sharwood. 2021. 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/.

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

[102] Jianfeng Tan, Cunming Liang, Huawei Xie, Qian Xu, Jiayu Hu, Heqing Zhu, and Yuanhan Liu. 2017. VIRTIO-USER: A new versatile channel for kernel-bypass networks. In *Proceedings of the Workshop on Kernel-Bypass Networks*. Association for Computing Machinery, New York, NY, USA, 13–18.

[103] Jörg Thalheim. 2022. Maintained fork of Linux for Ioregionfd patch. https://github.com/Mic92/linux/tree/ioregion-5.14.

[104] Jörg Thalheim. 2022. Run the evaluation. https://github.com/Mic92/vmsh/blob/main/EVALUATION.md.

[105] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. 2018. Cntr: Lightweight OS Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 199–212.

[106] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. 2017. Sieve: Actionable insights from monitored metrics in distributed systems. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. Association for Computing Machinery, New York, NY, USA, 14–27.

[107] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. 2014. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*. Association for Computing Machinery, New York, NY, USA, 1–14.

[108] Michael S. Tsirkin and Cornelia Huck. 11 April 2019. Virtual I/O Device (VIRTIO) Version 1.1. *OASIS Committee Specification 01* 1.1 (11 April 2019), 1. Latest version: https://docs.oasis-open.org/virtio/virtio/v1.1/cs01/virtio-v1.1-cs01.html.

[109] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. 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)*. ACM, New York, NY, USA, 559–572. https://doi.org/10.1145/3445814.3446714

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

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

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

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

[114] Sebastian Vogl, Fatih Kilic, Christian Schneider, and Claudia Eckert. 2013. X-tier: Kernel module injection. In *International Conference on Network and System Security*. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 192–205.

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

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

[117] Yuping Xing and Yongzhao Zhan. 2012. Virtualization and cloud computing. In *Future Wireless Networks and Information Systems*. Springer, Berlin, Heidelberg, 305–312.

[118] Ziye Yang, Changpeng Liu, Yanbo Zhou, Xiaodong Liu, and Gang Cao. 2018. 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)*. IEEE, Paris, France, 67–76.