Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

MASTER'S THESIS IN INFORMATICS

Methods and Technologies for Building Lightweight VMs for Function as a Service

Peter Okelmann

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

Master's Thesis in Informatics

Methods and Technologies for Building Lightweight VMs for Function as a Service

Methoden und Technologien zum Bauen von leichtgewichtigen VMs für Function as a Service

Author:	Peter Okelmann
Supervisor:	Prof. Dr.-Ing. Pramod Bhatotia
Advisor:	Dipl.-Ing. Jörg Thalheim
Date:	October 15, 2021

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Garching, October 15, 2021

Location, Date

Signature

Abstract

Hosting providers for online services offer deployment models such as Function as a Service (FaaS), enabling developers to build scalable and dependable micro-services. For security reasons, FaaS providers run lambda functions in virtual machines (VMs) which are combined with a multitude of abstraction layers to expose high-level, container-like interfaces. Methods to access lambda functions are therefore individual and highly specific to each virtualisation software stack. Bad accessibility makes debugging or inspection of lambda functions time-consuming and drives development time, as well as cost.

The active VM introspection tool Vmsh approaches the accessibility gap by injecting code into VM guests with minimal assumptions about the virtualisation software stack. We build λ-Pirate which integrates Vmsh into the lambda environment vHive, allowing developers to attach debugging or inspection environments to lambda functions on-demand, just when access is needed. Additionally, we extend Vmsh by a block-device, enabling user-supplied, big debug environments to be attached to VMs and by a console-device to provide user-interactivity.

Our evaluation of λ-Pirate finds that it maintains generality through its compatibility to four major VM hypervisors and that it does not add performance overheads to the attached lambda functions. With λ-Pirate we show how to reduce the accessibility gap of lambda functions while also minimising lambda environment specific code.

Thanks for Helma for being the best secretary. Thanks to Jörg Thalheim for driving vmsh development and the low level kernel hacking. Thanks to the University of Edinburgh for (involuntarily?) borrowing us CPU time. And thanks to Mom and Dad.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Highly scalable and dependable systems are essential for today's requirements towards online services. Increasingly popular areas such as mobile and web apps are prime examples for where online services are heavily used. To tackle the requirements for scalability and dependability, the industry came up with service architectures such as micro-services. Micro-services enable web service providers to build decentralised systems in which every instance can easily be duplicated to increase request throughput. Thus instances are also easily replaceable which assists with error recovery and is advantageous for small downtime. While those architectures were originally developed for backing big websites, the trend of building even native apps with web frameworks (electron [1], [2], fluttr [3]) further advances their adoption even in native apps.

Micro-service architectures pose new challenges as well as opportunities to hosting infrastructure providers. As even services from different developers often have similar needs and as they often have small public interfaces, micro-services are typically easy to manage. Hosting providers thus offer infrastructure abstractions such as the serverless cloud, to gain more freedom over where customer software containers are deployed as well as over optimisations the provider can employ. Serverless is also advantageous for the customer as he receives cheaper and more easy to use infrastructure, enabling him to build scalable and dependable consumer services. One of the most important efficiency optimisations, responsible for shaping today's clouds, is that infrastructure providers consolidate customers on physical machines (nodes) [4]. To provide secure isolation between individual customers, virtualisation is employed as containerisation is considered insufficient [5]. This leads to the first problem of modern cloud infrastructures:

Debugging and inspection of Virtual Machine (VM)s has bigger hurdles compared to containers.

Lambda functions drive the development of serverless even further by offloading even more software responsibilities to the infrastructure provider. While for serverless the developer submits software containers which solely interface via network, for lambda functions he submits the code for each web-endpoint itself. The lambda cloud provider now has more control as he even controls the application runtime, however the interface of the lambda function to the providers runtime grows in complexity. An increased interface size induces more dependencies on the API counterparts of cloud providers. In many cases the other end for the interfaces are not publicly available though which prevents developers from setting up a local testing environment. This leads to the second problem of modern lambda environments:

The lack of local reproducibility of lambda function infrastructure severely hinders developers to debug their functions.

The first step towards solving those two problems is proposed by VMSH [6]. VMSH is a tool to spawn debugging shells into VMs which can be used for tasks such as inspection or debugging. Traits important to its design are that it is independent of the hypervisor as well as independent of the VM userspace – not requiring agents to be running the VM guest. The spawned shell can be used for tasks such as inspection or debugging. By using VMSH in lambda environments, capabilities for interactive debugging can be provided in a non-intrusive way and with very little assumptions about the infrastructure. Therefore, we build λ-Pirate which integrates VMSH into a serverless stack for lambda functions called vHive. Specifically, lambda environments pose some unique challenges for debugging: λ-Pirate needs to automatically select the VM of the first lambda instance in which an error occurs next in order to attach VMSH to it. Furthermore, λ-Pirate prevents destruction of the lambda VMs through the autoscaler.

With our work on λ-Pirate we bring forward the following contributions:

- We build λ-Pirate to advance the accessibility of serverless platforms for developers. λ-Pirate provides an abstraction enabling debugging VMs independent of the hypervisor or guest and reduces virtualisation platform specific code to a minimum.
- We show that VMSH can be implemented for and integrated into existing serverless platforms with ease. To show its non-intrusiveness, λ-Pirate integrates VMSH into the lambda environment vHive.

- We show that λ-Pirate – in particular VMSH-blk – does not impose performance overheads onto the existing hypervisor components.

In Chapter 2 we describe the big variety of container runtimes, hypervisors and how they can be plugged together. We explain mechanisms important to understand the VMSH-blk device such from KVM or the VirtIO protocol. Finally, we introduce vHive which we integrate VMSH into. The motivation in Chapter 3 lays out use cases for λ-Pirate and discusses their existing solutions involving fat VM images, guest agents as well as agent-less approaches. Chapter 4 introduces the design of VMSH, VMSH-blk and -console and finally how it plugs together and integrates into vHive with λ-Pirate. λ-Pirates components and VMSH-blk are described in detail in Chapter 5. We evaluate λ-Pirate across three dimensions in Chapter 6: We show the correctness of VMSH-blk with xfstests [7]. We test its generality by using it with 4 industry leading KVM based hypervisors. Finally, we benchmark the VMSH-blk and -console devices regarding their impact on the guest and their IO performance.

Chapter 2

Background and Related Work

2.1 Abstractions for MicroVMs and Containers

The market of virtualisation and containerisation products is constantly evolving. Hypervisors are built with different specialisations to achieve full system virtualisation whereas container runtimes are used for lightweight compartmentalisation. On the other side, there are high-level orchestration systems such as openstack [8], docker [9] or kubernetes [10] which are used as a frontend for service developers or Function as a Service (FaaS) frameworks. In between the hypervisors and container runtimes on the one side, and the orchestration frameworks on the other, there exists a vast variety of intermediate layers, abstractions and interface standards. We depict a non-exhaustive set of intermediate layers in Figure 2.1. Some of which are deprecated or have been replaced with other solutions.

The intermediate layers are hard to grasp due to their complexity: Openstack can for example be used to manage and provision VMs in datacenters [8]. It uses libvirt [11] as an abstraction layer for its runtimes such as the qemu hypervisor [12] or lxc [13] containers. Openstack also supported intel clearcontainers, which however has been reaplced by kata [14] containers now [15]. Kata in turn can be used through the containerd interface [16]–[18]. This interface is supported by docker [9], [19] and kubernetes [10]. The firecracker hypervisor [4] has the firecracker-containerd adaptor [20] available, to connect to containerd compatible interfaces as well. However, firecracker can also be managed by kata [21] which in turn can be used through containerd, too. Furthermore, kata can be used through frakti [22] and CRI-O [17] which are implementations of CRI [23] (alongside rktlet which is for the rkt containers [24] and cri-containerd which adapts for containerd). Since kubernetes also supports the CRI interface [25], it can

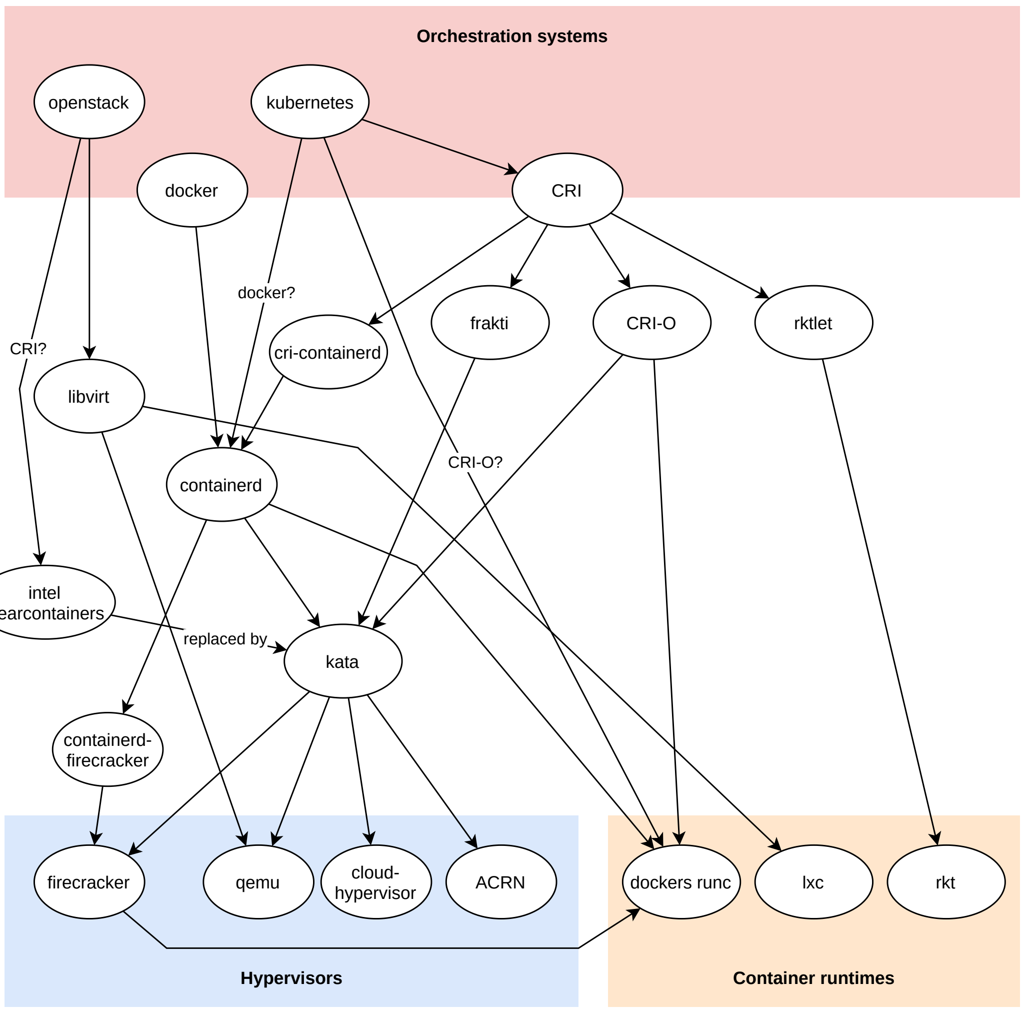


Figure 2.1: A non-exhaustive set of orchestration systems on the one hand – and hypervisors and container runtimes on the other. In between there is a complex interplay of intermediate layers.

manage for example dockers runc runtime via either containerd or CRI-O [23]. However, kubernetes can manage hypervisors such as firecracker, qemu, cloud-hypervisor [26] and ACRN [27] through kata as well [21], [28]

Because it is difficult to introduce new container tools compatible with this big variety of software, those tools bring the risk of being either hard to maintain due to their multitude of dependencies to all the intermediate layers – or they lock the user into a specific software stack.

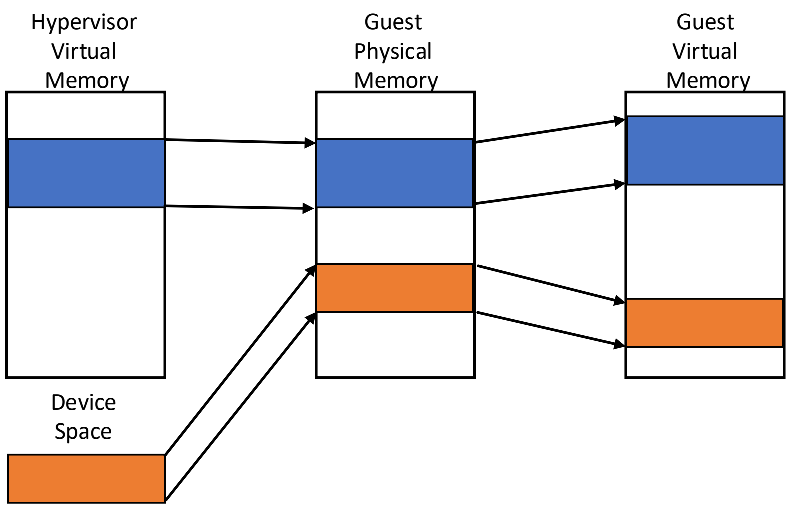


Figure 2.2: When working with VM memory, there are three relevant address spaces involved. VirtIO devices additionally use MMIO for registers called device space.

2.2 Kernel Virtual Machine (KVM)

Hardware-assisted virtualisation uses hardware 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, 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 [5], [29]–[31].

The actual program that runs the guest OS, called the hypervisor, is implemented in the userspace, and it uses KVM. KVM specific hypervisor implementations include qemu [12], Firecracker [4] and cloud-hypervisor [26]. The hypervisor sets up the initial CPU and memory state and emulates the IO devices (e.g., block, console, NICs) for the guest OS. After the setup, the hypervisor issues a `KVM_RUN ioctl` system call per virtual CPU to run the guest. This `ioctl` blocks the hypervisor thread until the virtual CPU causes a `vmexit` which occurs, for example, when accessing an IO device. Guests have their own virtual memory mappings based on their own guest physical memory (see Figure 2.2). The guest physical memory is based on virtual memory in the hypervisor process.

To emulate devices, it is additionally required to emulate device registers and memory. When the guest tries to access guest-physical device addresses, the CPU causes an interrupt that in turn causes the `KVM_RUN ioctl` to return control to the hypervisor. Thereafter, the hypervisor emulates the device response based on the interrupt. VirtIO devices use this interrupt mechanism to emulate devices that are made available to the guest.

2.3 Virtual Devices with VirtIO

Emulating physical hardware is slow and causes significant overheads compared to the native execution on 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 the virtual machine optimised device emulation for network devices, block devices, etc. Most hypervisors implement VirtIO devices and their guest drivers exists for all major OSes. [32], [33]

The VirtIO protocol defines a handshake and feature negotiation mechanism between the guest driver and the virtual device. This handshake follows the design of hardware interfaces heavily depending on MMIO accesses. Afterwards a device specific handshake may follow. When the device is initialised, the virtio protocol provides a transport layer over which device specific protocols are spoken. Depending on the device type, VirtIO specifies a number of consumer/producer virtqueues in shared memory, which the device code in the hypervisor and the driver in the guest use to exchange the requests and responses. The queue operations don't require MMIO in principle, however most device protocols rely on doorbell notifications which are implemented by VirtIO with MMIO.

VirtIO has two major transport mechanisms which differ in their MMIO memory: one based on either pure memory mapped IO (MMIO) or on the PCI standard. We implement the MMIO variant, which is more widespread, especially in microVMs [4], [34].

2.4 vHive Architecture around Firecracker

vHive [35] makes available web-endpoints by managing lambda-functions deployed with firecracker-containerd micro-VMs. For out-of-the box FaaS setups, OpenFaas [36] is commonly used. vHives purpose on the other hand is to provide a platform for 'serverless experimentation' [35]. It is comparable to industry FaaS infrastructures as it is built 'to be representative of the leading FaaS providers, integrating the same production-grade components used by providers, including the AWS Firecracker hypervisor [4], Cloud Native Computing Foundation's containerd [16], and kubernetes [10].' [35]. For firecracker-containerd, vHive implements snapshotting mechanisms to reduce cold-start delays but also supports other runtimes such as plain containerd/knative and gVisor [37].

vHive uses kubernetes as deployment system which allows to define deployments through configuration files. While vHive incorporates many standard infrastructure services for example from knative serving [38], it replaces the container engine i.e. docker with itself. vHive conforms to containerd for its public API to connect itself to kubernetes. Instead of implementing the entire underlying container engine anew, it offloads many tasks directly to firecracker-containerd [20] which already implements the containerd interface. It thereby wraps around firecracker-containerd which is a system of many components by itself and exposes a container-like API for firecracker-based micro-VMs [4].

Firecracker-containerd uses a 'runtime shim' [39] which communicates with the firecracker API to configure and run a VM. The VM then boots into a Linux kernel with a default image containing the VM-internal 'agent shim'. The guest agent shim then communicates with the runtime shim over a vsocket [40]. The workload, i.e. lambda function container, is deployed into the VM in the form of an OCI container [41]. The container is handled and run by *runc* which in turn is controlled by the agent shim.

Chapter 3

Motivation

This section shall give a notion about λ-Pirate's goals of use-cases in Section 3.1 and discuss the capabilities and implications current solutions have in Section 3.2.

3.1 Use-Cases

There are many use-cases for which VMs in the cloud have to communicate with the outside world. While their main workload usually simply communicates over http or gRPC, other scenarios involve more steps due to their different goals.

3.1.1 Debugging

Developers often desire to have the program of interest on their own host and within their personal development environment. Thereby they are able to use all their familiar, locally installed tools and configurations and are not limited in the way they can interface with the program of interest. With cloud infrastructure this becomes a problem since applications and services are increasingly containerised, virtualised and distributed over physical machines.

Particularly with lambda functions in serverless environments, the container/VM and the function runtime are controlled by the provider and thus the API interface between the provider and the developer's workload is significant in size. Reproducing the runtime locally is therefore not easy which makes local debugging difficult. Since the developer has no point of interaction with the function, except its source code, approaches such as print debugging are used instead. This is however inefficient for lambda environments as for each change the function has to be redeployed. Thus, programs are hard to access and debug which can result in a frustrating development experience [42].

3.1.2 Tracing

Cloud engineers want to detect anomalies or performance bottlenecks and therefore need to measure the timings of requests to their services. Trends like micro-services however increase the complexity of cloud infrastructures. Dependencies between services can become complex, and their performance characteristics depend on the interplay of the individual micro-services' behaviour. Therefore, cloud engineers require distributed tools to measure interactions and their timings between micro-services. Importantly this process needs to be non-intrusive as not to disturb the measurements and not to crush the hardware in production deployments.

3.1.3 Lightweight VMs

Lightweight VMs are commonly used to reduce memory footprints and boot-up times of VMs. Cloud providers as well as their customers use them to improve the scalability and efficiency of their infrastructure. With lambda functions, the provider gains the ability to control most of the therefore relevant components. They optimise boot time through miniaturisation of kernel configurations, hypervisors and init systems. Another crucial component of lightweight VMs is their small image size which is achieved by stripping images from scarcely used software like monitoring and debugging tools, speeding up the deployment process.

3.1.4 Security Inspection

Security inspection of VMs is often done on a regular basis, particularly for long-living monolithic VMs. Admins want tools in the VMs userspace to monitor and check other parts of it [43]. Those tools depend on certain parts of the VM to remain trusted though. Therefore, VM introspection tools are desirable. They run on the host, are isolated from the VM itself with the goal of making them independent of directly attackable software like the VM guest.

3.1.5 Inspection Services

Providers want to integrate their customers VMs into their own services to gain advantages over their competitors. Depending on the type of VM the services could differ: Customers of Lightweight VMs typically have lots of them. Therefore, aggregating status dashboards, corresponding monitoring tools, and data collectors are of interest for customers.

3.2 Approaches

All the use-cases from Section 3.1 have already been solved. They are approached by packing plenty of tools into the VM images, using agents to access the VMs, and some specific use-cases are even solved with agent-less methods. The goal of this work is to present a solution to the use-cases while avoiding issues of existing approaches.

3.2.1 Fat Lambda VMs

The selling point of serverless lambda environments is their cost-effectiveness. Providers manage the lambda function runtimes and have thus the ability and are incentivised to optimise the function runtimes to maximise efficiency. The advantages of lightweight VMs (see Section 3.1.3) are however compromising on feature richness and the options to access, inspect and debug them. Therefore, customers are incentivised to add all tools again they might potentially need in the future. Adding software to cloud VMs retroactively is often not supported. Instead, developers have to rebuild the base VM image and redeploy the VM which leads to disruptions. During debugging this means that system state which might be of interest for a developer is lost. In the example of debugging (as in Section 3.1.1), this is erroneous program state. For distributed tracing (see Section 3.1.2), the state which leads to anomalies could get lost.

Those issues lead to big VM images which already contain as many programs as possible, all of which are predicted to be required at some point in the future. This principle not only applies not only to VM image sizes, but also other VM characteristics like interfaces, devices and features implemented by the hypervisor.

This work strives to enable developers to keep their VMs lightweight.

3.2.2 Agent-based

Many use-cases from Section 3.1 have already been solved with agents. A developer interactively debugs a VM (as in Section 3.1.1) by accessing it via ssh [44] and by using tools such as gdb [45] installed in it. When a developer requires additional software on demand, he relies on a package manager to be available and working. The ssh server, the tools and package manager qualify as agents in this scenario. Cloud providers try to simplify many of those steps by building specialised agents to manage software updates, user accounts and configurations in client VMs [46]–[51]. However, none of them are complete and some even complement each other [51]. In the area of distributed tracing, there are a multitude of agent based solutions [52]–[56] each of which have their own unique selling points [57]. The corresponding agents for lambda environments take the form of libraries [58].

Each of those agents increase deployment complexity for the customer. He has to configure them, install and update them. Last, but not least, agents pose additional attack surface, because they are continuously exposed to the network [59]. For developers, agents pose challenges as well because they require a large stack of components to be operational. Networking devices and driver need to exist and the network as well as the guest configuration must be working. Ssh identities, user accounts and authorisation policies must be in place. To summarise, agent induce lots of complexity in order to fulfil their purpose.

This work shall require no agent to be installed in the VM guest.

3.2.3 Agent-less

There are tasks for which agents are insufficient. Hypervisor specific APIs offer additional capabilities for such situations. qemu [12] provides a debugger interface that can be used for debugging the guest kernel, while also allowing one to attach disks at run time. Crosvm [60], on the other hand, only has a debugging interface similar to the one provided by qemu. Other hypervisors such as Firecracker [4] and kvmtool [61] lack both these features. In other cases, such features, even when supported by hypervisors, are tightly integrated with orchestration frameworks such as OpenStack [8] or Containerd [16], which do not expose an API to interact with the underlying hypervisor.

Such interfaces have many applications. The guest kernel can be debugged (see Section 3.1.1). Additional data can be made available without interfering with the guest userland, keeping VMs lightweight (see Section 3.1.3). Providers can collect 'coarse grained information about the resource usage of the entire guest [62]' [6] (see use-case from Section 3.1.5). VM introspection tools targeting security (see Section 3.1.4) also commonly use such APIs. Such tools running on the host conduct binary analysis of guest memory to bridge the semantic gap [63] – enabling users to extract high level OS information such as running processes [64], [65].

While those approaches are specialised for their scope, they are highly specific i.e. require specific hypervisor interfaces. Their adoption in cloud environments is therefore difficult, because there the selection of hypervisors follows other needs first.

This work shall not rely on hypervisor-specific APIs in order to be hypervisor-agnostic.

Chapter 4

Design

4.1 Overview

We introduce λ-Pirate, a tool which helps developers debug their lambda functions. λ-Pirate's high-level mode of operation is depicted in Figure 4.1. As a lambda function stack we use vHive (see Section 2.4) which is provisioned using kubernetes. A developer may now use λ-Pirate in this vHive setup, to debug or inspect a lambda function. λ-Pirate therefore subscribes to the microVM manager (vHive) to read all lambda function logs. On certain events, such as errors being printed, λ-Pirate attaches an external debug environment to the faulty microVM.

To inject code into the VM to start a debug environment, we use Vmsh whose design we describe in Section 4.2. It works by injecting code into a VM to spawn a debug container which does not interfere with the guest userland. As shown in Figure 4.2 the container is able to access the VMs file system root through a subfolder as well as its own file system root. It is connected to the world outside the VM through two virtual devices which we extend Vmsh by. I build Vmsh-blk (see Section 4.3), a block device to attach external program files to a VM – and Vmsh-console (see Section 4.4) to make the injected program available to the user i.e. through an interactive shell. Thereby λ-Pirate can bring the users interactive debug environment into a microVM. Section 4.5 describes the design of λ-Pirate and how it integrates Vmsh into vHive.

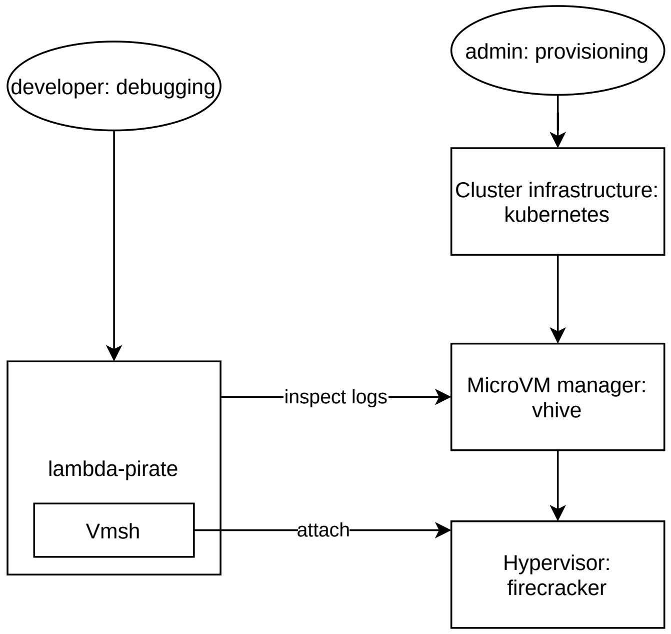


Figure 4.1: An admin provisions a vHive [35] lambda environment and a developer uses λ-Pirate for convenient and efficient debugging.

4.2 Vmsh

λ-Pirate wants to keep its dependencies small and tries to avoid being locked in to specific products such as hypervisor implementations. Therefore, λ-Pirate shares many design goals with Vmsh:

- Non-cooperativeness: λ-Pirate should not rely on agents running in the guest's userspace.
- Generality: λ-Pirate shall be agnostic to hypervisors and 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.

Thus, we base λ-Pirate on Vmsh and extend it by a block (see Section 4.3) and console device (see Section 4.4). Vmsh's initialisation process splits in three steps: Injecting code into the VM, preparing the guest kernel and finally executing an userspace program to set up the guest overlay container. In this Section we describe the design of Vmsh while the implementation was described by Thalheim et al. [6].

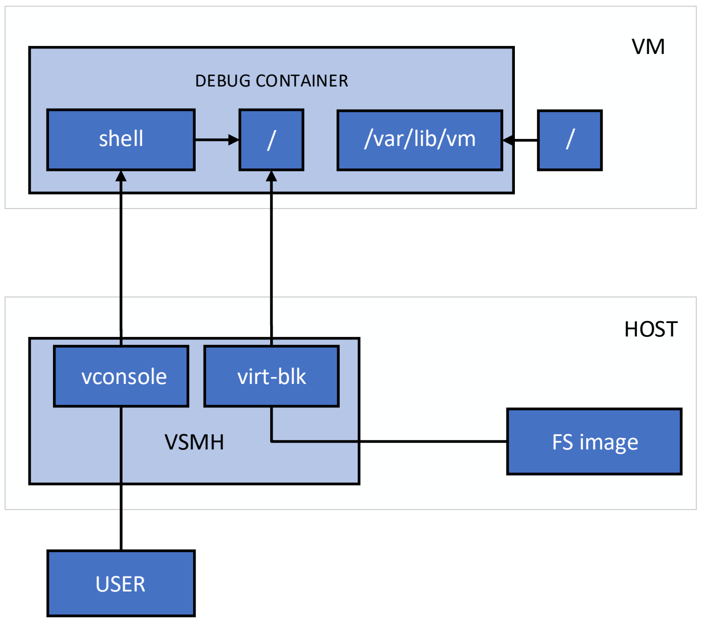


FIGURE 4.2: VMSH components after setup. The user enters the VM with a debug container which mounts a debug environment over the VMSH-blk device as well as the VMs filesystems. The debug container is accessed via console through the VirtIO-console device.

4.2.1 Hypervisor-agnostic side-loading for VMs

First, VMSH uploads a kernel module to mount devices and to start the userspace process that creates the guest overlay container. However, this has to occur without any help from a guest agent or the hypervisor in order to meet our design goals.

To upload code into the guest OS we need the following information about the guest:

- The number of vCPUs in the guest and the file descriptors they are mapped to.
- Physical memory layout of the VM and its mapping in the virtual memory of the hypervisor.
- Location of the guest kernel in the guest physical address space and the exact addresses of the kernel functions.

First, we observe that the hypervisor process has the guest's physical memory, mapped into its own address space as shown in Fig 2.2. However, VMSH needs to know exactly where in hypervisor memory the guest physical memory is mapped to. With this information, VMSH loads and maps the additional kernel code that is required for mounting devices and starting the guest userspace process. To perform this operation in a generic manner, we cannot rely on the hypervisor as it provides no APIs to perform these low-level operations.

To access the above information Vmsh injects syscalls into the hypervisor process. This is required as the OS only allows to manipulate the guest from the hypervisor process.

To be able to inject syscalls into the hypervisor process we rely on the debugging and system call tracing tools 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 targeting and interacting with the low-level kernel API, KVM in our case, directly. Using this, Vmsh first stops or interrupts the hypervisor process. This also interrupts the guest VM's execution as interrupting the hypervisor threads interrupts the vCPUs of the guest. Next, Vmsh prepares the syscall arguments of the injected syscall, by updating the CPU registers according to a specific syscall ABI. In cases where syscalls may 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 syscalls.

Through the execution of the injected syscalls, Vmsh gathers the following information about the guest VM: number of vCPUs in the guest, mapping of the guest physical into its virtual memory, and the location and layout of the guest physical memory in the hypervisor process.

More specifically, using the low-level hypervisor API, KVM in our case, Vmsh first figures the number of vCPUS and dumps the register state of a vCPU to reveal page table location, CR3 register on x86 and TTBR0 on arm64, which provides information about all the virtual memory mappings of the guest VM. Finally, Vmsh acquires information about the mappings of the guest into the hypervisor's address space, using a small eBPF code that is injected into the host kernel since the low-level hypervisor API does not expose this information.

Using this information, Vmsh uploads the kernel code into the guest VM. It then uses the low-level hypervisor API to update the guest instruction pointer register to run from the uploaded code. However, modern operating systems employ hardening techniques such as KASLR that maps the kernel into random locations in memory on every boot. This makes it hard for Vmsh to inject the kernel code into the correct location. In the next section, we describe Vmsh's binary analysis techniques that enable it to gather information about the location of the kernel and its functions in memory. [6]

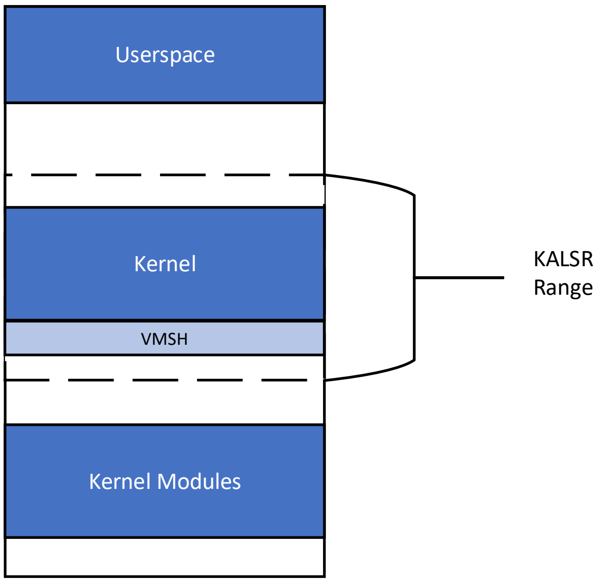


FIGURE 4.3: The virtual memory layout of the guest kernel. VMSH finds a safe spot to map itself to right behind the kernel.

4.2.2 Kernel-agnostic library

As described in the previous section, VMSH uploads a kernel module into the guest that enables it to mount devices and start the guest userspace process. Due to KASLR, a hardening technique that is used by modern OSes to map the kernel in a random location in virtual memory on every boot, loading the kernel module 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 module.

Although KASLR randomises the location of the kernel, the kernel itself is placed into a fixed number of slots in memory located in a fixed address range [66]. Hence, VMSH can locate the kernel by simply iterating over the guest VM's page tables.

VMSH also searches for the location of the function name section in the guest OS, for example located at *.ksymtab_strings* in Linux with similar mechanisms in other OSes [67]. The actual function addresses are stored in a different data structure (*.ksymtab*), whose size is not known. 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 update kernel module, containing references to kernel functions, that is side-loaded into the guest via Vmsh's custom binary loader.

Once the kernel module is ready to be side-loaded, Vmsh uses the information about the kernel ranges to side-load it into the guest OS by writing to the hypervisor memory. To load it in a manner such 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. This part of the memory is commonly not used as the hypervisors, in practice, incorrectly advertise the available physical addresses to the guest VM in order to allow VM migration in heterogeneous clusters.

Once the kernel module has been side-loaded, it needs to be mapped into guest virtual memory, so that it can be run from within the guest VM. The module 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 virtual memory, right after the kernel as shown in Figure 4.3, for the purposes of running the side-loaded kernel module.

Now that the module has been loaded into the guest VM and can be run, Vmsh modifies the instruction pointer of the guest VM's vCPU, via the low-level hypervisor API to run the module's code. To synchronise events between Vmsh running on the host and the side-loaded module running in the guest, we use a shared memory region that the guest polls for updates from Vmsh and vice versa. [6]

4.2.3 Container-based system overlay

After setting up the devices in the previous step, our kernel code starts an userspace process in the guest (see Figure 4.2). To bring in systems and additional devices, Vmsh needs to separate the environment using containerisation techniques. For example our design avoids restricting how the filesystem image by Vmsh or the guest filesystem is structured. On both filesystems, programs might rely on absolute paths being present that would conflict with each other. To resolve this conflict, Vmsh employs mount namespaces. The filesystem on the block devices of Vmsh will be mounted as root filesystem in a newly created mount namespace. All old mountpoints of the guest will be moved to a subdirectory (*/var/lib/vmsh*). Using a mount namespace ensures that these events are not propagated to any existing guest process except the ones started by Vmsh. To enable interactive shells, Vmsh will also set standard input/output file descriptors to its own console device. As virtual machines are more and more used to run container

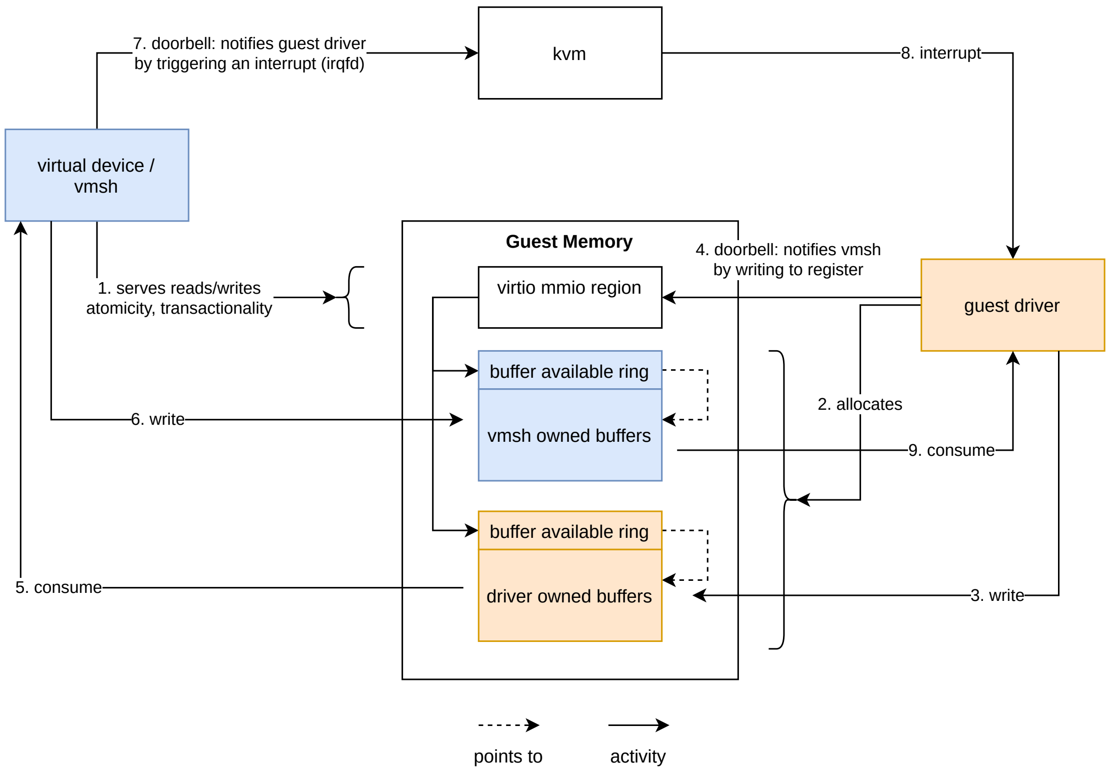


FIGURE 4.4: VMSH-blk device operation following the VirtIO specification. First (1-2) the VirtIO handshake is completed. Then (3-5) the guest driver may send buffers to the virtual device which are responded to by the device (6-9).

workloads, VMSH can also attach to containers rather than the root namespace. VMSH is not tied to a specific container engine here (i.e. docker, lxc, containerd, podman). Instead, it takes the process id of a containerised process. It scrapes all its process context (UID, GID, Apparmor/Selinux profiles, namespaces, cgroups, capabilities) from procfs and applies it to the new started shell. [6]

4.3 VMSH-BLK

The uploaded kernel code will then register VMSH's VirtIO devices. These devices need to be run in a hypervisor agnostic manner, and hence must run in a process external to the hypervisor. Hence, we design a VirtIO block device that run inside the VMSH process. VMSH uses the block device to serve the filesystem image containing applications. We next describe how we serve VMSH's block device according to the VirtIO protocol. VirtIO uses a MMIO region for driver updates and the device initialisation protocol. Therefore, the MMIO accesses are served by VMSH (Fig. 4.4/1.). During device initialisation, the guest driver allocates buffers (Fig. 4.4/2.) which are organised with thread-safe ring-queues. The ring-queues are used to manage and point to buffers which contain the transferred data. Two rings-queues make up one VirtIO virtqueue where the ring owned by the device is used to make buffers available to the driver and the ring owned by the driver to return them back to the device. This way the driver can enqueue buffers (Fig. 4.4/3.) and notify the device that there are some available by writing to a MMIO register (4.). Since those MMIO addresses are not mapped by physical memory the write causes a `VMEXIT` (see figure fig. 4.4/1). The exit is usually handled by the hypervisor i.e. by processing VirtIO queues to respond to outstanding requests. In Vmsh we need to inhibit this access and handle it in our own device. Finally, after the driver has announced available buffers to the device, the device can read the buffer by reading from hypervisor memory (Fig. 4.4/5.) and return its ownership back to the driver. Now, that the guest driver can send buffers to the Vmsh device, the other direction is easier: The guest produces buffers (Fig. 4.4/6.) and requests KVM (7.) to inject an interrupt into the guest (8.). Now the guest driver is notified and can consume the buffers (9.) again. The actual driver requests and device responses are stored in VirtIO queues that is also mapped in hypervisor process memory. Vmsh writes/reads those using inter-process memory access to handle drivers VirtIO requests.

4.4 Vmsh-console

I design a VirtIO console for Vmsh to forward the standard IO of the injected guest overlay and its spawned user payload to the user of λ-Pirate. Similar to virtio-blk devices, virtio-console devices are supported by most operating systems.

While Vmsh-console works similarly to the block device, there are some subtle differences. A plain simple VirtIO block device has only one virtqueue which handles the bidirectional communication protocol. The VirtIO console device on the other hand uses two virtqueues per console device, one for input and one for output. The VirtIO standard has an optional extension for adding additional virtqueue pairs to a single console device to allow it to support multiple consoles – to reduce implementation effort we rather attach multiple instances of the console device.

4.5 λ-Pirate

With Vmsh we have the capability to attach a debug environment to a micro-VM. λ-Pirate is now responsible for providing an interface to the user taking care of adjacent tasks in a serverless lambda environment. λ-Pirate thereby integrates Vmsh into vHive by providing the following functionality: (1) λ-Pirate listens to events on which a developer may want to interact with a lambda function. It (2) prevents scale-down of the function during the interaction with the user. And most importantly, λ-Pirate (3) finds the process id corresponding to the hypervisor running the respective lambda function.

1. Listen to Events: Serverless environments such as vHive are commonly used by customers to host micro-service oriented applications. They are based on the concept to subdivide an application into many small, independently scalable parts. This design favours deployments with many instances (replicas) of the same service or lambda function. The requests are then distributed between the replicas by a load balancer. Different balancing policies exist: Requests may be assigned randomly but uniformly over all replicas for high quality of service. Alternatively requests may be assigned based on different metrics such as packet headers or session id to improve spacial locality of related requests. It is therefore hard to predict where a packet will be routed to. Developers, who want to debug a lambda, therefore may not know where an issue will arise. Thus, we design λ-Pirate to listen to certain events and then attach to the replica in which the event occurred.

In micro-service architectures events are often transported via calls to REST web-endpoints or via gRPC. To execute those calls, the callee i.e. the lambda function has to be modified. The changes can be intrusive since REST or gRPC have a significant dependency footprint. While kubernetes already has its own eventing mechanism, triggering them requires changes to the kubernetes infrastructure as well. All of those approaches contradict our goal of generality and non-intrusiveness. Instead, we choose console output as our source of events. Console output is already used by most programming language environments to communicate errors which λ-Pirate can use as a trigger. To give an example: 96% of lambda functions, hosted by a leading cloud provider, run python, javascript or java code [68]. All of which print error messages for fatal errors. Particularly for debugging, the main use-case of λ-Pirate, console output is therefore a powerful source of events. This channel is usually available across serverless lambda infrastructures since without λ-Pirate it is the only universal mean of interaction between the developer and the function. We think that monitoring the console output of lambda functions is a simple, reliable and intuitive to use design choice.

2. Prevent Scale-down: Serverless environments such as vHive are commonly used by customers to host micro-service oriented applications. They are based on the concept to subdivide an application into many small, independently scalable parts. This concept favours deployments where each lambda function is automatically instantiated and replicated (scaled-up) in order to cope with the amount of web-requests. Consequently, functions are also scaled-down when the load is small. While a user is attached to a lambda-VM for debugging, it is however desirable that no other component changes the state of the VM or completely destroys it. For this duration it is therefore important to prevent automatic scale-down i.e. removal of the lambda function and its VM. Just as a load balancer could be configured stop forwarding request to the VM, an autoscaler could be configured not to remove a specific VM instance. While scale-down can be prevented via configuration in vHive, scaling is done in many ways by different companies and projects [35], [36], [69]. An option of interest may therefore not be implemented or exposed to the user. Thus, some options may require to modify the respective components. To maintain generality, we choose to restrict ourselves to using vHives configuration parameters to show off the ease of integration achieved by λ-Pirate.

3. Finding the Hypervisors Process-Id: When the lambda instance to be attached to has been identified, λ-Pirate resolves the instance id to the corresponding hypervisors process id. This process id is then used by VMSH to identify the target hypervisor to attach the debugging environment to. Similarly, as with preventing scale-down this pid can in some environments be found using existing tools whereas in other cases modifications to the involved code may be required. The vHive environment does not expose the pid via any API. To maintain generality we build λ-Pirate by choosing a non-invasive approach based on debug information.

As described in the previous three paragraphs, our design decisions orient themselves on the goal of generality. While VMSH reaches a high level of generality, λ-Pirate does have some functional requirements towards the serverless platform it operates in. From our experience with plugging VMSH into vHive, we learn that for the features which are not available, non-invasive workarounds exists or the changes to be made are small.

Chapter 5

Implementation

λ-Pirate integrates Vmsh into a serverless lambda environment called vHive as depicted in Figure 5.1. The vHive executable runs on the host and orchestrates instances of the firecracker hypervisor. After booting, the firecracker-containerd VM spawns a container running the workload i.e. lambda function. When λ-Pirate attaches Vmsh to the VM (see Section 5.1), it spawns an additional container which uses a busybox filesystem image as root. Since Vmsh is aware of the workload running in a container as well, it takes care of making its content available to the busybox container. The busybox image is served to the VM by Vmsh through Vmsh-blk which we describe in Section 5.2. The busybox container can now be used by a developer to inspect and debug the workload container.

5.1 λ-Pirate

We implement λ-Pirate in three steps:

1. We complete and upstream vHives implementation of output forwarding [70] in order to parse the output to find events.
2. We implement the hypervisor pid lookup from high-level workload deployment identifiers. This functionality is incorporated into the container-pid [71].
3. We implement λ-Pirate [72] to get events from workloads output and attach Vmsh to the matched workload.
4. Furthermore λ-Pirate prevents scale-down through vHives configuration. We offer extensive automation to reproduce this setup on NixOS.

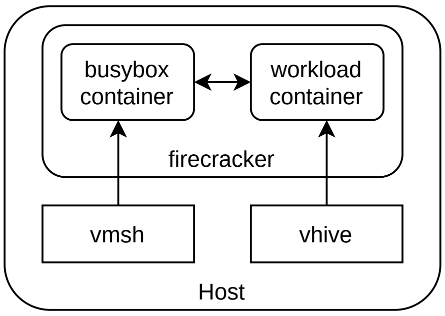


Figure 5.1: Container nesting with Vmsh and vHive: Both run containers within the firecracker VM.

5.1.1 Output Forwarding

λ-Pirate uses the workload containers standard output as a source of events which trigger an attachment. We ensure that the output can reach us by identifying its travel route: The stack of software required to move the output to λ-Pirate is depicted in Figure 5.2. The workload application is executed in its container by the runc container runtime inside the VM. Runc then passes the output on to the agent shim (still inside the VM) which passes it further to the runtime shim over a vsocket – thus making the jump from inside, to outside the VM. The runtime shim is controlled by firecracker-containerd (the containerd abstraction around firecracker). Container runtimes conforming to the OCI runtime specification [73] such as containerd [16] are designed to forward the workloads output to the host. Implementations such as firecracker-containerd [20] support this feature as well. Since vHive has full access to the containerd API of firecracker-containerd, it can thus also receive the workloads output. While vHive does connect the workloads output to its own output in principle, it makes no distinction between the individual sources – nor is the workloads output distinguishable from vHives own output. We therefore implement a standard output parser and forwarder which encodes it into machine-readable messages conforming to the logfmt style [74] and bring those changes upstream. Those logfmt messages are printed by vHive and contain the output line itself as well as the containerd id corresponding to the container which the output originates from. λ-Pirate can now monitor vHives output and parse the logfmt messages to search for error keywords. A match then triggers the lookup to find the hypervisor process belonging to the container associated with the message.

5.1.2 Finding the Hypervisor Pid

Resolving a workload instance to a hypervisor pid involves a chain of id lookups. We implement this pid lookup for VMs by extending the container-pid project which does the same for container engines [71]. Generally, multiple layers of abstractions have to be traversed to extract the hypervisor pid. The frontend for the user who deploys the

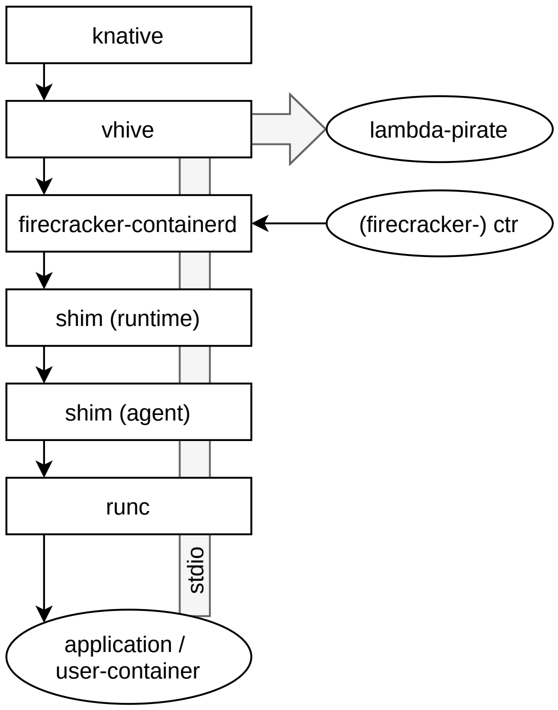


Figure 5.2: vHives stack of components which are required to forward console output from the application (lambda function) to vHive and thus λ-Pirate.

workload is the kubernetes API i.e. kubectl. Here, the workloads are referenced by three names: The workload is a container which has a unique name within its pod. Pods in turn have a unique name within in their namespace. The combination of all three names uniquely identifies the workload container a user may want to debug. With the help of the kubernetes API, this identifier can be used to retrieve the id assigned to the container by firecracker-containerd ('containerd id'). When the firecracker VM is started by vHive, vHive is aware of both, the containerd id and the corresponding VM id. We preserve this knowledge so that at a later point in time the VM id can still be resolved. The firecracker VM id can now finally be used to find the pid of the firecracker VM instance in question. In our specific use case, we already get the VM id of the hypervisor which printed the message. Thus, we can take a shortcut and skip the first two resolution steps. When we now receive an event from a lambda-function which makes us want to attach to it for debugging, we can now use our extension of container-pid to resolve the pid of the hypervisor. This pid is now passed to Vmsh to attach the debug environment.

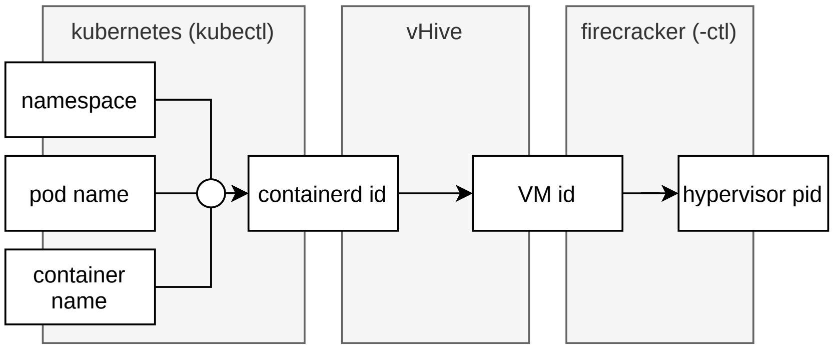


Figure 5.3: The hypervisor has different identifiers on different layers of abstraction. Containerpid [71] translates those identifiers to process ids.

5.1.3 vHive Configuration

When a developer wants to debug a workload, λ-Pirate will wait for an event to attach to the VM. Afterwards some more time will pass until the developer notices that Vmsh is now attached and until he starts debugging. Given the highly scalable nature of lambda environments, it is likely that within that time the autoscaler has already scaled down the number of replicas of the respective function.

To give an example, we assume that the user runs λ-Pirate on a function in which he sees occasional failures. At some point a request will trigger an error in the function which triggers the attachment of Vmsh. Now the developer can for example attach a debugger to the workload to find out why it failed. Often developers send test requests manually. When the developer starts debugging, the requests stop and thus the function will be scaled down. Furthermore, debugging through a human may very likely take longer than the timeout of the failed request.

This poses the following problems: The function, the developer is attached to, may get destroyed by the lambda environment since the lack of requests causes the autoscaler to scale down unnecessary replicas of the function. Additionally, when a lambda function failed once or the request timed out, the load balancer may assume that it is available again for new requests. Newly incoming requests may however destroy the state of the failed one. Those problems require changes in the load balancer and autoscaler which can be implemented in different ways.

vHive uses a fork of knative serving for load-balancing and autoscaling [38]. While knative serving offers multiple autosaling systems [69], without external programs none of them supports fine-grained configuration which λ-Pirate needs [75]. We consider changing the scaler and load-balancer code as too intrusive. Another approach would change vHive instead. It has to relay all scale-down commands on to firecracker-containerd.

The containerd shim API version 2 has explicit commands for stopping a VM [76]. A λ-Pirate-aware vHive could therefore delay its relay of the stop command until the developer has detached Vmsh. We find a even less intrusive solution though: vHive (as well as knative autoscalers [75]) can be configured to maintain a minimum number of replicas of each function. By choosing this number sufficiently high, the debugged lambda function is not scaled down while accessed by a developer and thus the problem is solved.

5.2 Vmsh-blk

To provide a highly correct implementation of our devices, we base our work on existing libraries written for VirtIO(-blk) devices. The libraries are shared among multiple hypervisors such as firecracker [4], rust-vmm/vmm-reference [77] and crosvm [60]. We adapt them to our purposes and plug them back together. To implement a VirtIO device, one has to fulfil the following requirements for the core components:

The VirtIO MMIO memory region (or at least parts of it) must be served by the virtual device **atomically** and **transactionally**. The requirement of atomicity arises because of fields such as *Status* (0x070), which are written by both the driver and the device, possibly at the same time. Further, accesses must be transactional to implement for example the *DeviceFeatures* register (0x010) which returns a different device feature page depending on the value last written by the driver to *DeviceFeaturesSel* (0x014). Usually, mechanisms that provide transactionality can also provide atomicity because they block one party from accessing the memory at certain times. Nevertheless, we lay out our design for both, because satisfying atomicity can present significantly less performance overhead.

In this section we investigate transport options, mapped and unmapped (MMIO) memory, their capabilities and how they can be used to fulfil the core requirements.

5.2.1 VirtIO Transport Options

VirtIO can use different transport systems out of which we choose to use MMIO [78]. If we reference specific device registers, we mean the variant as specified for VirtIO over MMIO.

PCI: Using the PCI/PCIe bus for VirtIO allows the driver to benefit from features implemented by PCI such as a dynamic device tree which supports automatic device discovery and hotplug [79]. It also allows the device to announce a location for the common device config (comparable to MMIO device config) - but a separate address can be announced for the *DeviceNotify* register [80]. The disadvantage of this transport is the additional implementation effort on the hypervisor/device as well as the driver side. Especially hypervisors trimmed for micro-VMs deliberately do not support PCI for performance reasons (i.e. qemu microvm [81][34] or firecracker[4]). Therefore, we choose the MMIO transport instead.

MMIO: MMIO has the broadest support in hypervisors. Compared to PCI transport, VirtIO over MMIO drops all wrappers around the device config space and assumes that whether and where the config space lies in memory is already negotiated. The Device Notify register is not configurable and has a fixed location within the MMIO device config space.

Channel I/O: This transport is specific to S/390 based virtual machines which support neither PCI nor MMIO [82]. We do not support that architecture and thus ignore this option.

5.2.2 Foreign Mapped Memory

This section deals with memory mapped into the VM. That is a memory page is allocated in the virtual address space of the process which spawned the KVM (the hypervisor process). Vmsh does not spawn the KVM. Thus, the allocated page is foreign to Vmsh and must be accessed remotely. The page is then attached into the physical memory space of the VM. Therefore, when we refer to mapped memory in this section, we do not mean memory mapped into the virtual address space of some guest kernel or process, but memory mapped into the physical address space of the VM.

Atomicity: Generally speaking, the VirtIO standard requires 32bit wide accesses to be atomic, but for wider accesses (in the device-specific configuration space) it states that the ConfigGeneration atomicity mechanism 'SHOULD' be used by the driver to guarantee atomicity.

In usual applications atomic memory accesses are guaranteed by the hardware architectures load/store operation to which the access is translated to. This requires the memory to be mapped into the address space of the application. Sharing memory between the hypervisor and Vmsh can achieve this, however the virtqueues are allocated by the driver and are thus located on memory pages the guest is initially booted with. Because we cannot share a page after the fact, **shared memory** cannot be used to provide atomic operations on mapped guest memory.

The trivial approach to atomicity is stopping the hypervisor/guest with ptrace, apply the operation and resume again. The data processing path of VirtIO contains atomic operations though (for each push and pop of a virtqueue buffer) which makes them very frequent. Thus using **ptrace** to make operations seem atomic to the guest has a very high performance impact.

The syscalls *process_vm_writev* and ptrace peek/poke do not guarantee atomicity. They dynamically copy user data of arbitrary length. On x86 that requires copying atomic chunks as small as 1 byte. Usually, conditionals are added to *memcpy()* to detect longer copies so that bigger chunks can be used. This optimisation is likely to be done on all reasonably implemented architectures, because it adds a constant per-memcopy overhead while reducing the per-byte overhead (added $\mathcal{O}(1)$ overhead is outweighed by $\mathcal{O}(n)$ benefit). We find that x86 copies atomically in 1 or, when possible, in 8 byte chunks [83]. That means we can guarantee 32bit (4 bytes) wide atomic memory copies on x86 if the copy starts 8 byte aligned and has a length of at least 8 bytes. Similarly, a lower bound for copy-length can be found on arm64 for which chunks of 4 bytes are copied atomically [84]. Therefore, **if Vmsh has exclusive write access** to the entire aligned 8 byte chunk enclosing the 32bit field, **then process_vm_writev can be used** for atomic accesses. Since this approach promises to provide the best performance, we use it for our implementation.

Transactionality: Physical device registers are typically backed by hardware which guarantees that the response to every single read can be individualised. This behaviour is atypical for normal mapped memory. Here multiple reads and/or writes can occur before the backing memory can be updated by a virtual device. The VirtIO specification has been written with the behaviour of physical device registers in mind and thus requires transactionality.

One effect of this approach is the design choice of making the doorbell event mechanism from driver to device a `write(virtqueue_id)` to the QueueNotify register (0x050). A VirtIO device must therefore be able to listen to writes to this register. Combine this capability with a way to pause execution of the guest when such a write occurs, and you are able to update the backing memory atomically according to that write without creating a race condition with the guest. Thereby transactionality can be provided as long as consecutive reads on a register may return the same value which holds true at least for MMIO over VirtIO. The other way around that means, that transactionality also allows us to implement the doorbell mechanism.

Classically, GDB can achieve transactionality with **memory watchpoints** [85]. It can do this in software, which is ‘very slow, since GDB needs to single-step the program’ [86].

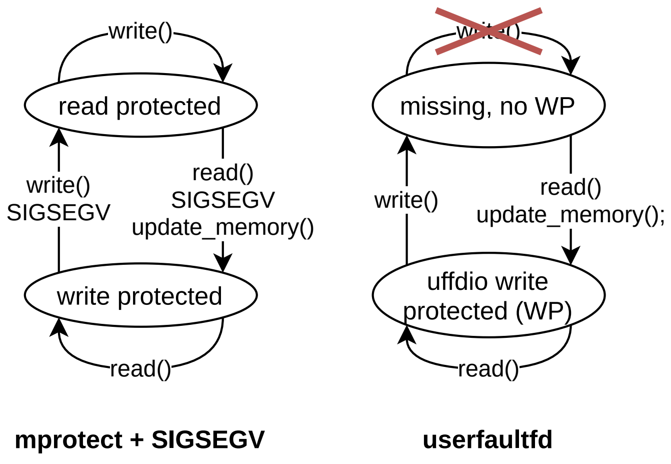


FIGURE 5.4: The states of a memory page which provides transactionality through mprotect+SIGSEGV and through userfaultfd. Userfaultfd ulitmately doesn't work because writes can't be served in the 'missing' state.

An efficient alternative on x86 is to use the available hardware acceleration: Up to 4 CPU registers can be used to monitor up to 4 consecutive bytes each for change [86]. In our scenario the watched physical addresses are accessed from within a VCPU though and naturally KVM clears and restores those registers before and after executing the VCPU. Memory watchpoints are therefore only accessible for the guest kernel which however does not know the host-physical address to use for the hardware registers. Watchpoints are therefore a dead end for transporting events out of the guest kernel.

There is another hardware assisted mechanism to track guest kernel writes to mapped memory: Pages can be **read- or write-protected** via the `mprotect` syscall. The corresponding memory access will then produce a SIGSEGV signal. The signal is accompanied by a `siginfo_t` which contains the `void *si_addr` which caused the fault. This signal can be caught with ptrace and handled (see Figure 5.4): As long as a page is read-protected, an arbitrary amount of writes can occur. The intermediate states are not urgently relevant for the state of the virtual device. Only when a read happens, a SIGSEGV is generated and caught by ptrace. The CPU is now already stopped (traced) and the device state and memory values are brought into a consistent and up-to-date state. Then the page permissions are changed from read- to write-protect, the VCPUs instruction pointer decremented to repeat the read, and the hypervisor continued. In this new write-protected state, consecutive reads will not change the state, and we must only interfere again when the first write arrives. In that case we catch the SIGSEGV again, switch page permissions, continue the hypervisor and the cycle begins anew. This way one could satisfy VirtIO's requirement for transactionality.

However, when code runs in a VCPU, page faults because of page permissions are handled in a special manner: For memory that is attached to a KVM, it is assumed that it can be used to emulate physical memory. That means read or write-protected pages are not quite expected. For Vmsh that means there is no SIGSEGV to catch in the hypervisor, because the VCPU exits with EFAULT instead which could potentially have other reasons as well. This approach to transactionality can be summarised to suffer from performance overheads and would further require research into how the guest's page fault ends up in a EFAULT VCPU exit.

Assuming we can make the kernel drop a page and mark it as missing/absent, as it would happen with a swapped page, we can implement transactionality with **userfaultfd**. Similar to the mprotect approach, the page alternates between two states: Missing without write protection and userfaultfd-write-protected. Traditionally, an access to a missing page can now be handled from userspace by restoring it (since Linux 4.10). This feature finds application for incrementally restoring the memory of a snapshotted VM into memory. For incremental snapshot creation, there also exists an userfaultfd-specific write protection for pages (since Linux 5.7), which, other than mprotect-write-protect, does work for VCPUs. Similarly to the mprotect+SIGSEGV approach, we can alternate a page between those two userfaultfd states (see Figure 5.4). When the page is missing and any access occurs, the page is allocated and copied from the buffer maintained by Vmsh. By passing a dontwake flag, we can further delay the accessing process to be resumed. Now the memory can be brought into a consistent and up-to-date state. Finally, the process is woken up via the userfaultfd. When a write occurs in write-protected state, the accessing process is stopped, the write-protection is removed and the process is woken up to finish the write. While this gives us a hook for pre-write, userfaultfd does not provide a hook for post-write. That means the write protection can not be turned on again in a non-racy manner. The read protection available with mprotect does not exist with userfaultfd and can only be replaced by dropping a page and marking it as missing. But in that state writes cannot be served. This challenge could possibly be overcome by inspecting VCPU registers and memory to determine the address and value to be written, however in combination with the high memory bandwidth overhead of copying the entire page for every state transition, we deem this approach as not feasible.

We conclude, that memory mapped into the VM can be accessed atomically, but not be served transactionally. Because of its good fit for data transfer, we choose this approach for handling the VirtIO virtqueues. Other aspects of VirtIO require transactionality though. Therefore, we next explore MMIO memory.

5.2.3 Unmapped/Mmio Guest Memory

Traditionally, the VirtIO device region is implemented via unmapped/MMIO memory. By design, accesses to such memory are atomic (for 32bit) and transactional in accordance to the VirtIO standard.

vhost-user: There is an existing mechanism to serve VirtIO devices from another process. Vhost-user serves the VirtIO MMIO memory region and in fact handles the entire VirtIO device from the hypervisors process to comply with the VirtIO specification. It then cooperatively makes available an abstraction to another process, allowing it to handle the virtqueues and receive and send doorbell notifications. Vhost-user therefore requires implementation in the hypervisor and further cooperation with the device-implementing process – in our case Vmsh. This contradicts our design goals because we target KVM as least common denominator and expect no cooperation. [87], [88]

KvmRunWrapper: What Vmsh essentially wants to do, is intercept MMIO exits affecting Vmsh's VirtIO memory region. Precisely that is possible using ptrace to interrupt the VCPU (a hypervisor thread), each time it enters or exits a syscall. Specifically the CPU registered of the traced hypervisor thread must be inspected to find and match the syscall id to ioctl, the first ioctl argument to KVM_RUN, and the return code to 0x0 (success). When that is the case, the kvm_run struct mapped to the VCPU file descriptor can be inspected to check the KVM exit reason. When that matches MMIO_READ or _WRITE, it further contains the accessed address (and optionally the value to write). If that matches an address range served by Vmsh, then all *preconditions* are fulfilled. The MMIO access must now be modified, a write registered or a read answered using the kvm_run struct.

To continue the guest's operation, the instruction pointer of the VCPU handling hypervisor thread could be decremented by one to restart the KVM_RUN ioctl. Thereby we make the MMIO exit completely invisible to the hypervisor. That is however not a necessity, because qemu plainly answers all unhandled MMIO reads with 0 and ignores such writes. Firecracker additionally prints a message with log level warn for it. Therefore, Vmsh simply rewrites reads to become a write before continuing the guest. That means the hypervisor interprets the kvm_run.data field as a write, which it ignores, while the guest/VCPU still interprets the data as response. Writes are also just passed through after registering it in Vmsh for its device.

Checking Preconditions: The checks to find MMIO exits among the syscalls start off with inspecting the syscall parameters. This can be done by getting the CPU registers of the traced VCPU thread via ptrace and applying knowledge about the architecture specific syscall convention. Alternatively ptrace has an abstraction (PTRACE_GET_SYSCALL_INFO) since Linux 5.3 to apply that knowledge by itself and return syscall information and parameters in a structured manner.

There is room for optimisation left: With seccomp one can load BPF code into ptrace context. That can be used to move some checks into the kernel. This could improve performance, because for every event filtered in the kernel we save two context switches. However, BPF can only operate on the input vector it receives from the kernel. It cannot access any outside memory because it cannot dereference pointers [89]. That means BPF can only be used for precondition checks on syscall context and parameters. Checking the kvm_run struct for the exit reason and the accessed MMIO address, and thus context-switching to userspace is still often necessary.

We count how often each check is hit: While applying load on the block device attached to the guest via hdparm we measure that 2/3 of the checks could be prevented with BPF. This ratio is expected to decrease though when more MMIO backed devices are in use. Because of that and the additional complexity, we do not employ this optimisation.

We conclude that KvmRunWrapper is able to serve MMIO memory atomically and transactionally. Thereby we implement it to serve the VirtIO MMIO region of each device.

IoRegionFd: KvmRunWrapper introduces overheads for all VMEXITs, propagating the overhead of Vmsh-blk to all other VirtIO devices. Every exit causes a interrupt of the hypervisor thread and a context switch to Vmsh for further exit handling. As an alternative to KvmRunWrapper with better performance we implement IoRegionFd [90]. IoRegionFd is a new KVM feature implemented in the kernel. It was developed under the leadership of RedHat and is under review on the KVM mailing list.

IoRegionFd is aimed at improving hypervisor capabilities to decouple hypervisor and virtual devices or to implement out-of-process devices. It provides an API to register sockets on VM memory. Arbitrary physical address ranges as seen by the VM can be registered to it. The memory ranges can then be used as MMIO memory. Accesses to those regions are then communicated over the sockets i.e. to another thread or process. The receiving end of the socket, such as another thread or process and in our case Vmsh, can then register the access address and optionally a written value. Additionally, IoRegionFd has a blocking mode and a post mode. In blocking mode, the VCPU waits for the reply to each MMIO access. Only when the access has been acknowledged or the read value has been returned over the other socket, the VCPU continues. Thereby it replaces the VMEXIT mechanism with one that better fits the multi-threaded execution models of today's KVM hypervisor implementations. In post mode, the VCPU may continue immediately after sending a write into the socket. This mode is aimed at use cases such as doorbell notifications which are today implemented using IoEventFd. In such use cases transactionality is not required since the asynchronous write is only used to notify another asynchronous task.

Compared to KvmRunWraper, IoRegionFd induces fewer overheads as context switches are saved. On the other hand it currently requires to run a custom Linux kernel build [91]. We implement Vmsh-blk to detect IoRegionFd compatibility of KVM at runtime and offer a command line option to switch between KvmRunWrapper and IoRegionFd.

5.2.4 passing doorbell notifications

For passing doorbell events from the virtual device to the guest driver, traditionally a KVM irqfd is used. It injects an interrupt into the guest which is very similar to what would happen on real hardware. This is also what Vmsh uses.

Unmapped/MMIO memory: For unmapped MMIO memory, passing doorbell events from the guest driver to the virtual device is essentially trivial because of the transactionality of the memory. We optimise this however:

IoEventFd is a KVM feature which allows the hypervisor to register posted writes to an address in unmapped MMIO memory when IoRegionFd is absent. That means writes to that address do not cause an MMIO exit but instead the write is posted to an event fd where the event can be asynchronously polled. Since eventfds cannot transport the value written but only the existence of the event, multiple IoEventFds can be defined with a datamatch each to be fired when the written data value matches it. Due to those properties ioeventfd is traditionally used to implement the doorbell mechanism from the guest driver to the virtual device.

IoRegionFd could replace IoEventFds datamatch because it can indeed transport the written value via its fds. It also has a posted writes mode that could be used and in which MMIO exits are avoided.

Mapped memory: We also consider mapped memory to serve a VirtIO MMIO region. For example the doorbell mechanism can be built when transactionality is given. Busy polling however is no alternative because it cannot reliably detect repeated writes to the doorbell register. That is because the value written to the register is the id of the next ready queue. This value never changes when there is only one queue. Nevertheless, to comply with the VirtIO standard such a detection is required. The Linux implementation of VirtIO drivers block when they request to be notified about new buffer, but are not notified. Finally, the Linux KVM API has the capability to monitor writes to memory regions by tracking dirtied pages (KVM_CAP_DIRTY_LOG_RING) [92]. The dirty status is polled asynchronously, whereby a doorbell write can be detected. Looking back at Section 5.2.2, this tracking cannot be used however to implement transactionality for mapped memory because an immediate VCPU pause is impossible due to the dirty rings asynchronous properties.

Chapter 6

Evaluation

6.1 Experiment Setup

We perform our experiments with machine that have an Intel Core i9-9900K CPU with 8 cores (16 hyper-threads), 64 GiB of memory (caches: 32KiB L1; 256 KiB L2; 16 MiB L3). All disk benchmarks are run on a dedicated P4600 NVMe 2TB drive. The host OS is Linux version 5.12.14. For better reproducibility we pinned the benchmarks to cores and disabled Intel turbo-boost. Before each I/O related benchmark we discarded all NVME data with TRIM. For performance related benchmarks we used qemu with KVM as a hypervisor and gave the VM 8GB of RAM and 4 VCPUs.

6.2 Correctness

6.2.1 Xfstests

To ensure a correct implementation of the hypervisor tracer, virtio-block and -console devices, we conduct stress tests.

Group	System	Total	Ok	Skipped	Failed Vmsh-blk	Failed reference
3*quick	generic	516	407	109	0	0
	shared	2	1	1	0	0
	xfs	297	208	86	3	3
sum		815	616	196	3	3

Table 6.1: Xfstests: Vmsh-blk is implemented correctly because it does not fail more tests than the qemu-blk reference system.

Benchmark: xfstests: For fuzzing and regression testing of file systems and block devices, kernel.org has a test suite called xfstests [93][7]. It is a collection of over a thousand test scripts, some of which apply to block devices and some to file system specific features.

Methodology: We select the 'quick' test-group which contains the majority of tests, but excludes fuzzing tests. From this group we run the biggest test categories, generic and xfs, by provisioning a physical block device with two xfs partitions to be supplied as test- and scratch-partition. We run the tests natively on the host device, virtualised on the qemu virtio-blk device and in qemu on our Vmsh-blk device. We assume the native and the qemu block device to be correct and define failure of this benchmark as failing a test which passes natively or on the qemu block device.

Results: A fourth of the tests are automatically skipped by xfstests. Reasons are that the tests require other xfs build flags or versions. In some cases, tools (like man or dm* tools, fsverify) are missing or a SCRATCH_LOGDEV needs to be supplied. Those make up the vast majority of skipped tests.

All other tests succeed, except three. They succeed natively but fail on both the qemu-blk and the Vmsh-blk device. All of them are related to block device quota reporting i.e. reporting file system statistics. Other ones which test the enforcement of the quotas succeed though. Since Vmsh-blk passes all tests which are passed by known-good devices, we conclude that the Vmsh-blk device has no regressions in regard to qemu-blk.

To make up for the missing sustained load tests, we attach a 4GB OS image with Vmsh and calculate a sha-256 checksum over it. The checksum matches the one calculated natively on the host and thus succeeds the test.

6.3 Generality

We test the generality of our approach by testing my block device across multiple hypervisors:

- Qemu
- Firecracker
- Kvmtool
- Crosvm

Qemu, Firecracker as well as kvmtool are compatible with Vmsh-blk. Cloud-hypervisor is the only one not working with Vmsh. That is because other than most KVM based hypervisors it does not use legacy interrupts but instead PCI based Message Signalled Interrupts (MSI). While this can be beneficial from a design perspective, it is inherently incompatible with my Vmsh devices since they don't implement the PCI protocol as VirtIO transport.

6.4 Performance

We evaluate λ-Pirates performance regarding its IO speed. We measure the Vmsh-blk device, its console device and the impact of λ-Pirate on other VM devices while Vmsh is attached.

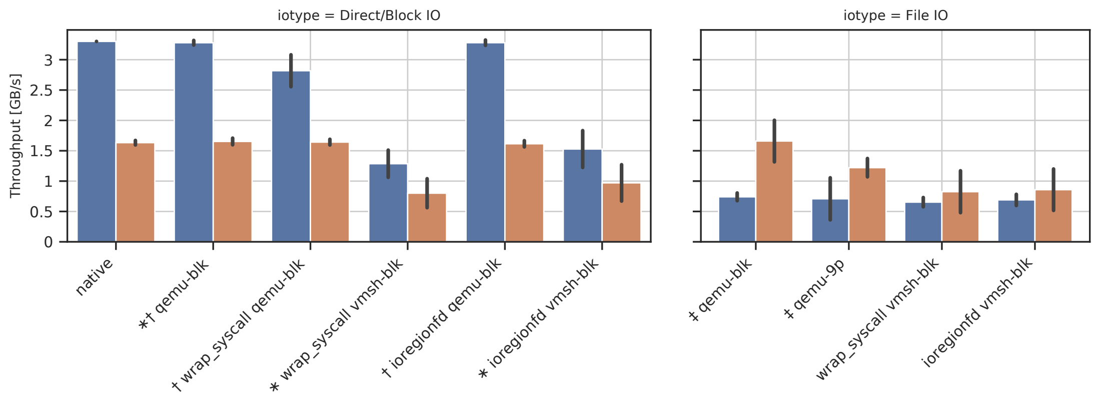
6.4.1 Vmsh-blk Performance

For a thorough understanding of the Vmsh-blk device, we conduct synthetic benchmarks, trying to isolate the results from as many external factors as possible. We use the feature rich IO workload simulator fio [94] with its libaio backend.

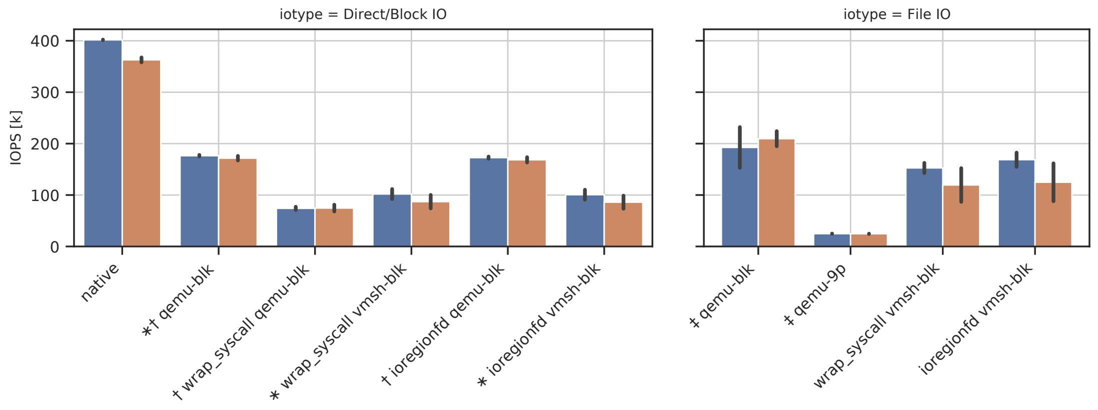
Methodology: To measure the overhead induced by Vmsh, we compare qemus block device with Vmsh-blk by running fio on it (direct/block IO). Additionally, we evaluate the choice of attaching a block device (virtio-blk) to the VM instead of sharing a view of the host file system with the guest (virtio-9p). Therefore, we add file IO based tests comparing Vmsh-blk with qemu-9p.

Using fios libaio backend, we measure the bandwidth and IO operations per second (IOPS) for reads and writes, separately. Bandwidth is measured in the best conditions possible to find the highest achievable one. That means big block sizes (256k) and sequential access. This test is usually bound by the hardware, because the number of IO operations is small and thus are also the per-operation software overheads. On the other hand the goal of our IOPS measurements is to maximise the software overheads while avoiding hardware bottlenecks. Therefore, we choose small block sizes (4k) to maximise per-access overheads. But we use sequential accesses, which does not affect the software components, but reduces the relative load on the hardware. Otherwise, our parameters stick closely to the examples given by fio.

Results: First we observe, that the host natively matches qemu-blk in bandwidth, but is 2x faster in IOPS (see Figure 6.1 $*$).



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



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

Figure 6.1: `fio` with different configurations featuring qemu-blk and Vmsh-blk with direct IO, and file IO with qemu-9p.

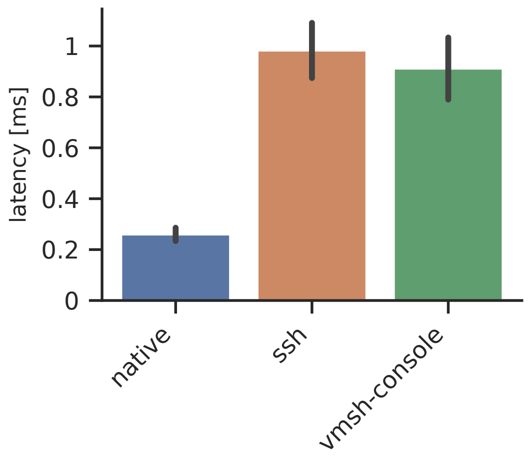


Figure 6.2: Vmsh-console responsiveness compared to SSH.

Compared to qemu-blk, Vmsh-blk introduces overheads of about 1.5-2.5x (see Figure 6.1 †). A factor, contributing to the higher standard deviations of Vmsh-blk is that contrary to qemu, it maps the block device into memory. This leads to the page caching of the host kernel to take effect. Because we limit the amount of page cache available to less than a tenth of the sequentially accessed storage, the bandwidth measurements only suffer in deviation but hold true on average. Vmsh-blk is therefore significantly slower than qemu-blk but in the same order of magnitude, which makes it a *good fit for debugging tasks*.

For file IO, the standard deviation is even bigger compared with direct IO, because additionally to Vmshs page caches, the caches of the guest file system also apply. We see that read bandwidth is about the same for all tested devices (see Figure 6.1 ‡). For writes, qemu-9p lies in between qemu-blk and Vmsh-blk which has a 2x overhead. IOPS on the other hand show a slowdown for Vmsh-blk of around 1.5x but for qemu-9p one of around 8x. Even though qemu-9p does not include any Vmsh overheads yet, it does not pose significant performance advantages. Therefore, we conclude, that basing Vmsh on virtio-blk instead of file based protocols like virtio-9p is *beneficial from a performance standpoint*.

6.4.2 Vmsh-console Performance

For a console, throughput is of lesser relevance. Instead, we compare its latency to ssh and 'the minimum viewing time needed [by a human] for visual comprehension' [95].

To measure the round-trip of a shell input, we connect one end of a pseudo-terminal seat (pts) [96] to a shell. Then we use the other end to submit an echo command to the shell and measure the duration until the echo response arrives. The measurement is run with the pts being connected to (1) a shell on the host, (2) ssh running a remote shell on the guest and (3) Vmsh running a shell on the guest.

Our measurements show that, with around 0.9ms, the latency of the Vmsh console is very similar to the one of ssh (see Figure 6.2). Research on the capabilities of the human eye shows that 'even durations as short as 13 ms are clearly sufficient, [...] to drive conscious detection [and] identification' [95] of a word displayed on a screen. This duration correlates to the one of a single frame on a 75Hz monitor which is common in office applications. The latency of the Vmsh-console is an order of magnitude faster than that which we think is sufficient.

6.4.3 Guest Performance under Vmsh

Using the same synthetic fio benchmarks from Figure 6.1, we measure the impact specifically of Vmsh-blk on the performance of other devices attached to the guest.

Exemplary we measure the impact on qemus block device. Therefore, we attach Vmsh(-blk) to the guest, but run our fio benchmarks on the qemu-blk instead of the Vmsh device. We repeat the tests once with wrap_syscall and once with ioregionfd and compare the results to baseline qemu-blk without Vmsh attached.

While qemu-blk with Vmsh-ioregionfd is as fast as without Vmsh, its performance drops with Vmsh-wrap_syscall (see Figure 6.1). Wrap_syscall produces read bandwidth overheads of 1.5x and IOPS overheads of 6x. Reason for this overhead is that wrap_syscall adds overhead for every syscall done by qemu and its devices. For every KVM MMIO exit, even Vmsh-unrelated ones, overhead is added to check relevance of the MMIO access to Vmsh. Those overheads do not occur with ioregionfd, because KVM already filters MMIO accesses for a Vmsh-defined MMIO region in the kernel.

Wrap_syscalls overheads violate Vmshs goal of non-intrusiveness. On the other hand it is measurably faster than ioregionfd. However, we conclude that the significantly faster qemu devices of ioregionfd outweigh the slight performance advantage of wrap_syscall.

Chapter 7

Conclusion

With λ-Pirate we integrate VMSH into the lambda environment vHive, empowering developers to attach debugging or inspection environments to lambda functions on-demand. We extend VMSH by a block- and console-device which are hypervisor agnostic and do not slow down the guest.

As future work, the generality of λ-Pirate could be advanced: An API framework could be built so that λ-Pirate can be integrated easily even into custom-built autoscalers and load balancer configurations. While container-pid already supports orchestration systems such as kubernetes, docker and vHive, it could additionally receive support for openstack.

With VMSH, active VM introspection becomes more feasible and easy to use which raises security concerns. A solution for customers to protect their VMs against introspection could involve memory encryption [97]. Nevertheless, the customer may want to grant outside processes access to the VM memory in some situations. Further research could explore ways, how λ-Pirate can cooperate with hypervisors to grant such access while maintaining the security guarantees for the VM.

During the process of developing λ-Pirate for vHive we learn that most other lambda environments likely will not require significant implementation beyond what we have already done. Combined with VMSH, which is hypervisor- and VM-userspace-agnostic, our system promises to be a versatile tool for many different lambda environments.

Chapter A

Appendix

This is the appendix. Remove the appendix if it is empty.

A.1 Appendix section

The appendix can contain different sections.

A.2 List of acronyms

VM	Virtual Machine
MMIO	Memory Mapped IO
Vmsh	VM Shell
λ-Pirate	Lambda Pirate
vHive	Open-Source Framework for Serverless Experimentation
FaaS	Function as a Service

Literatur

[1] Github, *Electron: Build cross-platform desktop apps with JavaScript, HTML, and CSS*, `https://www.electronjs.org/`, 2021.

[2] Google, *Google Trend: Electron (Software Framework)*, `https://trends.google.com/trends/explore?date=all&q=%2Fg%2F11bw_559wr`, 2021.

[3] Google, *Fluttr: Beautiful native apps in record time*, `https://flutter.dev/`, 2021.

[4] A. Agache, M. Brooker, A. Iordache u. a., "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA: USENIX Association, 2020, S. 419–434.

[5] 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.

[6] J. Thalheim, P. Okelmann, H. Unnibhavi, R. Gouicem und P. Bhatotia, "Vmsh: Hypervisor-agnostic Guest Overlays for VMs," 2021.

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

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

[9] Docker Inc., *Docker: Empowering App Development for Developers*, `https://www.docker.com/`, 2021.

[10] Kubernetes authors, *Kubernetes: Production-Grade Container Orchestration*, `https://kubernetes.io/`, 2021.

[11] Libvirt maintainers, *libvirt: The virtualization API*, `https://libvirt.org/`, 2021.

[12] Qemu maintainers, *Homepage of qemu*, `https://www.qemu.org/`, 2021.

[13] Linux maintainers, *Linux Containers*, `https://linuxcontainers.org/`, 2021.

[14] Kata maintainers, *Kata Containers: The speed of containers, the security of VMs*, `https://katacontainers.io/`, 2021.

[15] Kata maintainers, *OpenStack Zun DevStack working with Kata Containers*, `https://github.com/kata-containers/kata-containers/blob/main/docs/use-cases/zun_kata.md`, 2021.

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

[17] Kata maintainers, *Run Kata Containers with Kubernetes*, `https://github.com/kata-containers/kata-containers/blob/main/docs/how-to/run-kata-with-k8s.md`, 2021.

[18] Kata maintainers, *How to use Kata Containers and CRI (containerd plugin) with Kubernetes*, `https://github.com/kata-containers/kata-containers/blob/main/docs/how-to/how-to-use-k8s-with-cri-containerd-and-kata.md`, 2021.

[19] X.-L. Xie, P. Wang und Q. Wang, "The Performance Comparison of Native and Containers for the Cloud," in *2018 International Conference on Smart Grid and Electrical Automation (ICSGEA)*, IEEE, 2018, S. 378–381.

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

[21] Kata maintainers, *Kata Containers with Firecracker hypervisor*, `https://github.com/kata-containers/documentation/wiki/Initial-release-of-Kata-Containers-with-Firecracker-support`, 2019.

[22] Kubernetes maintainers, *Frakti: The hypervisor-based container runtime for Kubernetes*, `https://github.com/kubernetes-retired/frakti`, 2018.

[23] Kubernetes maintainers, *CRI: the Container Runtime Interface*, `https://github.com/kubernetes/community/blob/master/contributors/devel/sig-node/container-runtime-interface.md`, 2021.

[24] rkt maintainers, *rkt: a pod-native container engine*, `https://github.com/rkt/rkt`, 2019.

[25] Kubernetes maintainers, *Kubernetes: Container runtimes*, `https://kubernetes.io/docs/setup/production-environment/container-runtimes`, 2021.

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

[27] Project Acrn, *ACRN: A Big Little Hypervisor for IoT Development*, `https://projectacrn.org/`, 2021.

[28] Kata maintainers, *Kata: Hypervisors*, `https://github.com/kata-containers/kata-containers/blob/main/docs/hypervisors.md`, 2021.

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

[30] 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.

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

[32] R. Russell, "Virtio: Towards a de-Facto Standard for Virtual I/O Devices," *SIGOPS Oper. Syst. Rev.*, Jg. 42, Nr. 5, 95–103, Juli 2008, ISSN: 0163-5980. DOI: `10.1145/1400097.1400108`. Adresse: `https://doi.org/10.1145/1400097.1400108`.

[33] M. S. Tsirkin und C. Huck, "Virtual I/O Device (VIRTIO) Version 1.1," *OASIS Committee Specification 01*, Jg. 1.1, 1, 11 April 2019, Latest version: `https://docs.oasis-open.org/virtio/virtio/v1.1/cs01/virtio-v1.1-cs01.html`.

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

[35] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion und B. Grot, "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, 559–572. DOI: `10.1145/3445814.3446714`.

[36] OpenFaaS, *Serverless Functions, Made Simple*, `https://www.openfaas.com/`, 2021.

[37] gVisor contributors, *gVisor is an application kernel for containers that provides efficient defense-in-depth anywhere.* `https://gvisor.dev/`, 2021.

[38] Knative und V. maintainers, *Knative Serving: Vhive fork.* `https://github.com/ease-lab/serving`, 2021.

[39] Firecracker contributors, *Firecracker-containerd architecture*, `https://github.com/firecracker-microvm/firecracker-containerd/blob/main/docs/architecture.md`, 2021.

[40] Linux maintainers, *vsock(7) Linux Programmer's Manual*, Linux foundation, 2021.

[41] Open Container Initiative, *OCI Image Format Specification*, `https://github.com/opencontainers/image-spec`, 2021.

[42] Tdfirth, *Does anyone else find the AWS Lambda developer experience frustrating?* `https://news.ycombinator.com/item?id=26855037`, 2021.

[43] Cisco Systems, *Snort IPS*, `https://www.snort.org/`, 2021.

[44] OpenBSD maintainers, *OpenSSH remote login client*, OpenBSD, 2021.

[45] Linux maintainers, *gdb(1) Linux man page*, Linux foundation, 2021.

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

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

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

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

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

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

[52] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox und E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Proceedings International Conference on Dependable Systems and Networks*, 2002.

[53] R. Fonseca, G. Porter, R. H. Katz und S. Shenker, "X-trace: A pervasive network tracing framework," in *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*, USA: USENIX Association, 2007, S. 20.

[54] J. Kaldor, J. Mace, M. Bejda u. a., "Canopy: An end-to-end performance tracing and analysis system," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[55] B. H. Sigelman, L. A. Barroso, M. Burrows u. a., "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure," Google, Inc., Techn. Ber., 2010.

[56] J. Thalheim, A. Rodrigues, I. E. Akkus u. a., "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, S. 14–27.

[57] R. R. Sambasivan, R. Fonseca, I. Shafer und G. R. Ganger, "So, you want to trace your distributed system? Key design insights from years of practical experience," Carnegie Mellon University, Techn. Ber., 2014.

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

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

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

[61] Will Deacon, *Homepage of kvmtool*, `https://github.com/kvmtool/kvmtool`, 2021.

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

[63] P. M. Chen und B. 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, S. 133–138.

[64] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin und W. Lee, “Virtuoso: Narrowing the semantic gap in virtual machine introspection,” in *2011 IEEE symposium on security and privacy*, 2011.

[65] J. Pfoh, C. Schneider und C. 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, S. 1–10.

[66] Y. Jang, S. Lee und T. 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, 380–392.

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

[68] G. Rama, *Report: AWS Lambda Popular Among Enterprises, Container Users*, https://awsinsider.net/articles/2020/02/04/aws-lambda-usage-profile.aspx, 2020.

[69] Knative, *Supported Autoscaler Types*, https://knative.dev/docs/serving/autoscaling/autoscaler-types/, 2021.

[70] P. Okelmann, *Vhive pull request: wrap stdio of workload containers*, https://github.com/ease-lab/vhive/pull/326, 2021.

[71] J. Thalheim und P. Okelmann, *container-pid*, https://github.com/Mic92/container-pid, 2021.

[72] P. Okelmann und J. Thalheim, *lambda-pirate*, https://github.com/pogobanane/lambda-pirate, 2021.

[73] Open Container Initiative, *OCI Runtime Specification*, https://github.com/opencontainers/runtime-spec, 2021.

[74] Logfmt maintainers, *Parsing the logfmt style*, https://github.com/kr/logfmt, 2021.

[75] Knative serving maintainers, *Serving API: Autoscaling*, https://github.com/ease-lab/serving/blob/main/docs/serving-api.md#autoscaling.internal.knative.dev/v1alpha1.PodAutoscalerSpec, 2021.

[76] containerd maintainers, *Shim API v2*, https://github.com/containerd/containerd/issues/2426, 2021.

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

[78] M. S. Tsirkin and C. Huck, *Virtual I/O Device (VIRTIO) Version 1.1: Virtio Over MMIO*, https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.html#x1-1440002, 11 April 2019.

[79] ——, *Virtual I/O Device (VIRTIO) Version 1.1: Driver Requirements: PCI Device Discovery*, https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.html#x1-1020002, 11 April 2019.

[80] ——, *Virtual I/O Device (VIRTIO) Version 1.1: Virtio Structure PCI Capabilities*, https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.html#x1-1090004, 11 April 2019.

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

[82] M. S. Tsirkin and C. Huck, *Virtual I/O Device (VIRTIO) Version 1.1: Virtio Over Channel I/O*, https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.html#x1-1570003, 11 April 2019.

[83] Linux kernel maintainers, *process_vm_writev memcopy implementation for x86: copy_ user_generic_unrolled*, https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/arch/x86/lib/copy_user_64.S, 2021.

[84] L. kernel maintainers, *process_vm_writev memcopy implementation for arm64*, https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/arch/arm64/lib/copy_template.S, 2021.

[85] GDB maintainers, *GDB wiki: Internals Watchpoints*, https://www.sourceware.org/gdb/wiki/Internals%20Watchpoints, 2021.

[86] ——, *GDB manual: Setting Watchpoints*, https://sourceware.org/gdb/onlinedocs/gdb/Set-Watchpoints.html, 2021.

[87] E. P. Martín, *Deep dive into Virtio-networking and vhost-net*, https://www.redhat.com/en/blog/deep-dive-virtio-networking-and-vhost-net, 12 September 2019.

[88] Qemu maintainers, *Qemu spec about Vhost-user Protocol*, https://qemu.readthedocs.io/en/latest/interop/vhost-user.html, 2021.

[89] N. 'skepticfx', *Deep argument inspection in seccomp filters*, https://blog.skepticfx.com/post/seccomp-pointers/, 2021.

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

[91] IoRegionFd authors, *Linux kernel with IoRegionFd patches*, https://github.com/Mic92/linux, 2021.

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

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

[94] Jens Axboe, *Flexible I/O Tester*, `https://github.com/axboe/fio`, 2021.

[95] M. C. Potter, B. Wyble, C. E. Hagmann und E. S. McCourt, “Detecting meaning in RSVP at 13 ms per picture,” *Attention, Perception, & Psychophysics*, Jg. 76, Nr. 2, S. 270–279, 2014.

[96] Linux maintainers, *pts(4) Linux Programmer’s Manual*, Linux foundation, 2021.

[97] D. Kaplan, J. Powell und T. Woller, “AMD memory encryption,” *White paper*, 2016.