Rethinking Virtual Machine Monitors for Better Security in Clouds

A K M Fazla Mehrab
Virginia Tech
Blacksburg, Virginia, USA
mehrab@vt.edu

Ruslan Nikolaev
Pennsylvania State University
University Park, Pennsylvania, USA
rnikola@psu.edu

Binoy Ravindran
Virginia Tech
Blacksburg, Virginia, USA
binoy@vt.edu

Abstract

Though virtual machine monitors (VMMs) ensure the isolation and security of co-existing virtual machines (VMs), the security of VMMs lacks sufficient attention. Many components in a general VMM architecture are prone to future vulnerabilities and may contribute to propagating and aggravating damage to cloud safety and reliability. Moreover, performance acceleration architectures, like DPDK, when incorporated with VMMs, can become a single point of catastrophic vulnerability with the potential to affect security of multiple VMs.

We propose that by modularizing and compartmentalizing the VMM and its associated accelerator components using hardware-assisted isolation techniques, we can limit attackers' freedom and provide better protection against future adversaries. Additionally, as each cloud workload has unique requirements, we suggest a per-VM specialization to reduce the attack surface without sacrificing any features or compatibility. Our prototype VMM, Redwood, offers high configurability, smaller VMM images, support for running regular cloud OSs and unikernels, and resistance against known attacks while preserving performance. The proposed isolation mechanism for DPDK demonstrates similar advantages.

CCS Concepts

• **Security and privacy → Virtualization and security**.

Keywords

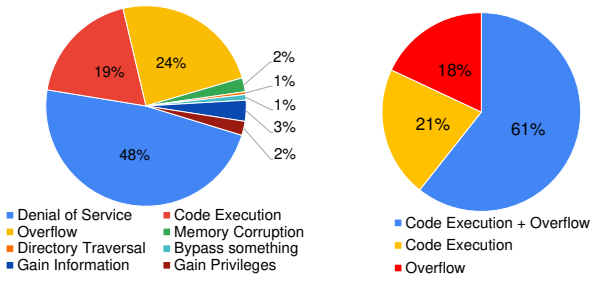
VMM, DPDK, Compartmentalization, Specialization

ACM Reference Format:
A K M Fazla Mehrab, Ruslan Nikolaev, and Binoy Ravindran. 2026. Rethinking Virtual Machine Monitors for Better Security in Clouds. In *27th International Middleware Conference (Middleware '26), December 14–18, 2026, Tarragona, Spain.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3801927.3858636

1 Introduction

Cloud infrastructures run thousands of server machines, each of which facilitates a multi-tenant environment so that different applications with varying resource requirements can run seamlessly and transparently without worrying about the underlying complexity. Such an environment should isolate tenants from one another to ensure data and execution integrity and confidence. Separate virtual machine (VMs), each equipped with the expected resources and capable of running the preferred operating system (OS), enable tenants to run their workloads securely with no modification. The ability to run many coexisting VMs simultaneously on the same server, with strong isolation between them, and easily manage resources have made virtualization technology the backbone of modern cloud infrastructures.

Hypervisors and virtual machine monitors (VMMs), often referred to interchangeably due to their overlapping and interdependent responsibilities, are essential components that enable virtualization. In this paper, we use the term *hypervisor* to refer to software that performs core operations such as system scheduling and physical resource allocation while running on bare metal (e.g., XEN [5], VMware ESX [59]) or that utilizes a host OS (e.g., KVM [30], VirtualBox [97]) for virtualization. We use the term *VMM* to refer to software (e.g., QEMU [19], Firecracker [76], Cloud Hypervisor [1]) that performs other essential virtualization operations with the help of a hypervisor. For example, QEMU with KVM can create a VM, allocate memory and virtual CPUs (vCPUs), help boot a guest OS, emulate virtual I/O devices, migrate VMs between servers, and stop VMs. VMMs play a crucial role in ensuring functionality, flexibility, performance, security, and reliability in virtualization environments. While VMMs are essential for VM-level isolation, each VMM runs as a userspace process in the host/administrative OS, contributing to the trusted computing base (TCB) size.



(a) All categories. **(b) Minimum score of 6.5.**

Figure 1: QEMU CVEs in categories from 2007 to 2022 [25].

VMM bugs are a major concern because they can undermine the main security benefits of virtualization. Numerous VMM vulnerabilities are reported annually, with QEMU being notably prominent. We analyzed and categorized 443 QEMU Common Vulnerabilities and Exposures (CVEs) reported from 2007 to 2022, as shown in Figure 1a. These vulnerabilities, including DoS, memory overflow,

This work is licensed under a Creative Commons Attribution 4.0 International License.
Middleware '26, Tarragona, Spain
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2621-7/2026/12
https://doi.org/10.1145/3801927.3858636

and arbitrary code execution attacks, represent various security flaws that require distinct mitigation measures.

Many bugs stem from improper memory handling, such as null pointer dereferencing, use-after-free, and out-of-bound access, all common in C due to its lack of safety features. Rust, with its type system and ownership model, can help prevent many of these vulnerabilities. However, no language can ensure complete safety, as human error remains a factor. For instance, despite Rust being newer, hundreds of CVEs have been reported for it [67]. Moreover, memory safety techniques alone are often insufficient to mitigate all vulnerabilities. An example is incomplete validation in the Rust-based VMM Firecracker, which led to CVE-2019-18960 [24], potentially allowing memory manipulation or remote code execution.

As shown in Figure 1b, only 18% of 61 high impact QEMU CVEs can be mitigated with memory safety techniques that prevent memory overflow. Therefore, there is an ongoing effort to implement QEMU devices in rust [69, 70]. However, these techniques alone cannot address the remaining CVEs: 21% involve arbitrary code execution, and 61% involve both arbitrary code execution and memory overflow. Thus, enhanced security primitives are needed to prevent many severe vulnerabilities.

Undiscovered bugs necessitate preventive mechanisms to thwart adversaries from exploiting vulnerabilities into severe zero-day attacks [100–103]. Modularity in software design can reduce the risk of arbitrary code execution and data theft. By compartmentalizing unrelated modules with isolation mechanisms, unauthorized access and damage propagation can be prevented. CPU manufacturers and researchers introduce new isolation mechanisms to complement or replace traditional ones. This paper proposes modularization and isolation mechanisms for compartmentalizing VMMs.

Modern cloud applications have diverse safety and performance requirements, but a rigid VMM design or configuration for every VM may not provide optimal safety. For example, QEMU provides a floppy drive controller (FDC), although most workloads no longer use floppy drives. The VENOM vulnerability [6] in the FDC may allow a guest user to crash the VM or execute arbitrary code on the host with QEMU process privileges, regardless of whether the VM needs this device. Thus, a faulty implementation can threaten the virtualization ecosystem. Flexible per-VM VMM specialization is therefore crucial to minimize the attack surface. In Section 3, we compare strip-down and build-up approaches for flexible per-VM specialization, addressing modern cloud workload requirements while reducing the attack surface.

On the other hand, I/O acceleration architectures like DPDK [86] and SPDK [81] can enhance VM performance by moving the data plane from the VMM to a separate userspace application on the host. However, this requires the VMM to share sensitive VM information with the application, making it an attractive target for attackers. To protect VMs from being compromised if the accelerator is attacked, we propose isolating each VM's information within the application.

We developed methods for compartmentalizing and specializing VMMs to improve cloud security, incorporating these security primitives. Our prototype, Redwood, extends Cloud Hypervisor [1], an open-source VMM written in Rust, to run modern cloud workloads. We identified 13 crucial components of Cloud Hypervisor, such as virtual devices, and compartmentalized them using Intel memory protection keys (MPKs) [21] on x86 servers. This helps prevent unauthorized access to virtual devices and limits attackers' ability to exploit vulnerabilities, affecting other devices or executing arbitrary code. Moreover, Redwood transforms Cloud Hypervisor into a highly configurable and flexible VMM by breaking its components into fine-grained, independent modules. This allows only the necessary features to be included when running a VM, significantly reducing the VMM image size at build time.

The emergence of unikernel is popularizing the concept of minimalist single-purpose VMs to improve the security of cloud applications. To enable such VM-level specializations to benefit from VMM specialization, we extend Redwood's support for unikernels. Since many unikernels support only multiboot boot loading, we implement multiboot bootloader in Redwood. Therefore, unikernels can benefit from our specialized hardened VMMs and, altogether, can contribute to reducing the overall virtualization attack surface.

Our evaluation in Section 5 shows that the proposed security primitives can mitigate a class of vulnerabilities. Redwood reduces the image size and the number of return-oriented programming (ROP) gadgets, i.e., short instruction sequences that can be chained for code-reuse attacks, by up to 50% compared to Cloud Hypervisor. Although a reduction in ROP gadgets doesn't always guarantee a reduction in an attacker's exploit construction ability, an in-depth gadget set analysis [9] on Redwood indicates a decrease in the availability of special purpose gadgets without increasing the expressivity of the fundamental gadget set, thus limiting an attacker's flexibility. Performance evaluation through micro- and macro-benchmarks shows that Redwood performs similarly to Cloud Hypervisor, with little to no performance penalty for compartmentalization. This trend is also observed for vhost-user isolation in DPDK.

The contributions of this paper are as follows.

- We identify the security problem of over-featured and insufficiently compartmentalized cloud VMMs, including the often-overlooked risk that userspace accelerators such as OVS-DPDK can amplify cross-VM exposure by holding sensitive VM state in a shared process.
- We design a workload-aware VMM architecture with fine-grained build-time features, enabling per-VM specialization that removes unused functionality while preserving support for regular cloud OSs and unikernels.
- We introduce an intra-VMM compartmentalization mechanism that isolates unrelated VMM components and individual device instances. By using event-driven permission changes, the mechanism supports asynchronous virtual-device execution without expensive domain switches.
- We extend the isolation design to OVS-DPDK vhost-user by compartmentalizing each VM's memory inside the accelerator backend, reducing the blast radius of a backend compromise and preventing arbitrary access to other connected VMs' memory.
- We implement Redwood and isolated OVS-DPDK and evaluate them against Cloud Hypervisor and regular OVS-DPDK. Our evaluation shows that our work substantially reduces VMM image size and code-reuse gadget availability while preserving performance across network, storage, ballooning, and vhost-user workloads.

2 Background

Virtual I/O devices. A VMM's major role is managing I/O for VMs. Emulating traditional I/O devices is inefficient due to significant performance penalties [10]. To improve performance, VMMs offer special paravirtualized (PV) I/O devices. Virtio [66, 88, 93] is a specification for PV devices that provides an interface for software to manage and exchange information. A *virtio device* runs on the host and translates signals between the *virtio driver* and the virtio data plane. The virtio driver in the VM communicates with the virtio device according to the virtio specification [93].

Virtio devices operate in several ways, depending on the location of their data plane. When the VMM handles both data and control plane operations, it is referred to as a *virtio* device, such as *virtio-net* for networking. If the VMM delegates the data plane to the kernel, it is called a *vhost* device, such as *vhost-net* for networking. When the VMM offloads the data plane to a separate user-space process on the host, it is called a *vhost-user* device, such as *vhost-user-net* for networking. The discussion of the pros and cons of each variation is beyond the scope of this paper. A modern VMM should support all variations so workloads can choose the best fit for their needs.

OVS-DPDK Vhost User. Open Virtual Switch (OVS) [62] is a userspace software switch for virtualized environments. OVS can forward network packets between two userspace applications without incurring the overhead of context switching between the kernel and userspace. DPDK [86] is a library that accelerates packet processing in data plane applications. It allocates resources to dedicated logical processing cores before the application is called. In contrast to the Linux kernel, which uses a scheduler and interrupts to switch between processes, DPDK accesses devices through constant polling. Its poll mode drivers (PMD) allow packets to be transferred directly between user space and physical interfaces, completely bypassing the kernel network stack. Kernel bypass and the elimination of interrupt handling and context switching overheads boost performance.

An OVS application connects with DPDK, allowing a VM to connect to an OVS-DPDK application running in the host's userspace through the vhost-user interface of the DPDK library. For each guest VM created on the host, another vhost-user backend is instantiated to communicate with the guest's virtio driver. This enables OVS to forward packets between VMs on the same machine without the overhead of context switching. Furthermore, OVS-DPDK with PMD further enhances performance.

Memory-based Isolation. We explore memory-based isolation techniques for compartmentalizing VMM components. Software fault isolation (SFI) [95] uses memory-bound checks to prevent components from indirectly accessing each other's memory, requiring compile-time instrumentation or binary rewriting. Its execution overhead is typically up to 42% [40, 74]. Some approaches use hardware page protection for memory isolation, where access permissions are checked during address translation with no added execution overhead. However, switching between components requires changing the extended page table (EPT) base, incurring expensive context switches. Although light-weight context switching approaches [7, 17, 33, 45, 46] reduce overhead, the toll is still significant, where the upper limit varies from 10% to 65% [45, 89], depending on the application and approach.

Unikernels. Unikernels [8, 36, 38, 41, 42, 48, 50, 51] are a type of lightweight OS that is optimized for use in cloud systems and run atop a hypervisor in a separate virtual machine. They are designed to run a single application and are created by statically compiling the application with the minimum necessary kernel code and libraries into a single-address-space image. This reduces the code size and memory footprint, which reduces the attack surface, and eliminates the need for context switching, allowing system calls to be made as ordinary function calls, which can improve performance. Despite their minimalistic nature, unikernels are run by full-featured VMMs like QEMU. To embrace minimalism, our prototype minimal VMM includes support for unikernels.

3 Design

3.1 Design Principles

This paper aims to reduce the VMM attack surface and contain future exploits without sacrificing compatibility, performance, or deployability. We propose the following design principles for hardening cloud VMMs. We use *instance* to refer to a runtime occurrence of a VMM component, such as each virtual device created for a VM.

(P1) ***VMM design should be modular and highly configurable.*** A VMM should not include any default feature or implementation that cannot be removed when it is not needed.

(P2) ***A VMM should not include unused components.*** Each VMM should be built to satisfy the needs of a specific VM, rather than being a full-featured generic VMM for all workloads.

(P3) ***Unrelated VMM components should be placed in different compartments.*** This limits an attacker's ability to use a vulnerability in one component to access or corrupt another.

(P4) ***Each instance of the same component type should be isolated.*** Keeping two instances in the same compartment may allow one compromised instance to affect another.

(P5) ***Security should not come at the cost of performance.*** The isolation mechanism should therefore be lightweight enough to incur little to no measurable overhead.

(P6) ***Compatibility with existing VMs should not be compromised.*** The introduced security primitives should be transparent to the VMs running on the VMM.

3.2 Challenges

Achieving a VMM with these principle poses following challenges.

C1: Specializing a VMM without breaking compatibility. A natural way to reduce the VMM attack surface is to remove unused functionality. However, existing VMMs are not organized as fine-grained, independently selectable components. The challenge is to specialize a production VMM for each VM without removing required functionality or modifying guest OSs. Using an existing lightweight VMM with limited functionality or developing a new VMM from scratch often sacrifice compatibility, feature coverage, and long-term maintainability.

We present Redwood, a workload-specialized VMM that refactors Cloud Hypervisor into fine-grained build-time features and adds multiboot support for both cloud OSs and unikernels. As a result, redwood preserves compatibility with existing workloads while reducing each VMM to only the functionality required by its VM, without requiring guest modifications.

C2: Isolating VMM components at the right granularity. Running each VM in a separate VMM process isolates VMs, but not mutually distrusting components within a VMM. This is particularly problematic for virtual I/O devices, which maintain sensitive state, access guest memory, and execute asynchronously through event handlers. Existing library-oriented isolation models do not fit this structure well because devices are long-lived, stateful instances rather than simple callee libraries. The challenge is to isolate each device instance while preserving efficient asynchronous I/O.

We design an hardware-based intra-VMM compartmentalization mechanism that isolates each virtual-device instance and grants access only during its legitimate I/O handling. Thus, a compromised device is confined to its own compartment and cannot freely access other device state or protected VMM memory, limiting its impact without disrupting normal I/O processing.

C3: Hardening accelerated vhost-user backends. I/O accelerators such as DPDK improve VM networking performance by moving the data plane outside the VMM. However, vhost-user requires the VMM to share guest memory with the backend, allowing a compromised OVS-DPDK process to access the memory of multiple connected VMs. Isolating DPDK as a single compartment is insufficient because that compartment would still contain all guest mappings. The challenge is to enforce per-VM memory isolation within DPDK while preserving its high-throughput data path.

We extend compartmentalization to OVS-DPDK by isolating each VM's memory and granting access only during the corresponding vhost-user operation. Therefore, a compromised backend is restricted from freely accessing other VMs' memory while preserving the existing high-performance vhost-user data path.

3.3 Achieving Per-VM Specialization

We discuss various options for creating a highly configurable VMM (**P1**) that offers a minimal attack surface (**P2**) while still maintaining the needed compatibility (**P6**).

VMMs have several **core** responsibilities including creating, allocating resources for, launching, and destroying VMs. However, these tasks are not sufficient for running an operating system with workloads inside a VM. As a result, a VMM may need to provide **additional** support, such as firmware like BIOS and UEFI, interrupt controllers like APIC, MSI, and MSI-X, configuration and power interfaces like ACPI, I/O devices, and VM migration mechanisms.

Any feature not needed by a workload only contributes to the attack surface. The uKVM [99] approach combines unikernels with relevant VMM components and replaces standard PV devices with low-level interfaces to create a unified executable. However, this approach is designed specifically for and requires modifications to unikernels, which violates the principle of backward compatibility (**P6**) and may not be suitable for all VMs. In addition, running a user-provided executable directly on the host degrades security.

The virtualization community often introduces new software features, such as vDPA [90] and VFIO [92], and hardware features, such as Intel SGX [23] and AMD SEV [77], to improve performance and security. Full-featured VMMs like QEMU are often the first to test these features due to their widespread market share. However, build-from-scratch approaches like uKVM suffer from a lack of

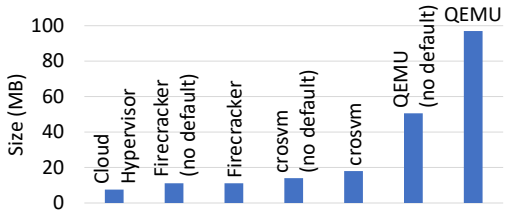


Figure 2: VMM image size comparison.

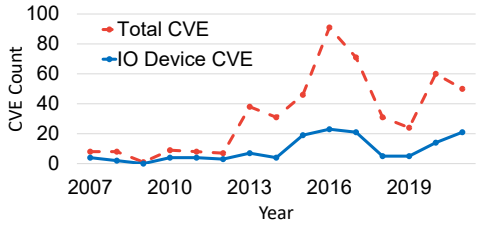


Figure 3: QEMU CVE trend from 2007 to 2022.

support and infrequent updates. Therefore, even if a workload desires, it may not use minimal VMMs because of missing features.

In contrast, QEMU typically has hundreds of features implemented and uses Kconfig files [37]. Firecracker [76], crosvm [79], and Cloud Hypervisor [1] selectively implement fewer features as they aim only to support microVM, ChromeOS [80], as well as cloud OSs, respectively, and uses `features` [27] in cargo [11]. We built these VMMs for x86_64 ISA with both default features enabled and disabled, except for Cloud Hypervisor. Disabling its default features using the cargo option `no-default-features` resulted in build errors. Figure 2 compares the sizes of the produced images. As we can see these VMMs are insufficiently configurable as most of their features are default and non-removable.

Not all workloads have the same requirements. To create a VMM with a minimal attack surface, it is necessary to make each VMM highly customizable to suit the specific needs of workloads. Achieving a minimal VMM from an existing one without losing any necessary features or compatibility requires fine-grained configurability, a feature not offered by any current VMMs. It demands nontrivial refactoring and modularization of the current implementation. Next, we discuss features that should be configurable at build time.

Bootloading. VMMs offer several ways to boot a VM. One method is the traditional boot sequence: (1) the firmware locates the Master Boot Record (MBR) on the VM image, (2) the bootloader loads the kernel, and (3) the kernel starts the init process. Another method is for the VMM to directly load the GRUB bootloader from the host using the VM's configuration, with GRUB then loading the kernel and starting the init process. A third method involves directly supplying a kernel and init disk to the VMM, allowing it to skip many normal boot steps and start the init process from the supplied disk. While there are other methods to boot a VM's OS, each VM only needs one. Thus, we minimize the VMM by keeping only the necessary method.

I/O Device. A VMM facilitates I/O operations by a PV, a fully emulated, or a physical device. For example, a VM can use an emulated

e1000 NIC, a virtio network device, or a physical NIC accessed with PCI-passthrough. Each device type may have further variations. For instance, a virtio block device may support different virtual disk formats, such as raw, qcow, vdi, and vhdx. Thus, a full-featured VMM supports many types of I/O. In contrast, a VM only requires a small subset of these implementations to support its workloads.

Figure 3 shows the number of I/O device-related and total QEMU CVEs reported from 2007 to 2021. I/O devices account for about one-third of the total CVEs, suggesting that I/O device implementations may be more vulnerable than other parts of a VMM. Reducing the number of I/O implementations by retaining only those needed by a particular VM significantly reduces the VMM attack surface.

Migration. VM migration between hosts is essential for cloud operations like load balancing, fault management, and system upgrades. A VMM may implement various migration techniques, such as checkpoint-restart, post-copy, pre-copy, and hybrid migration. Each technique has its pros and cons, but a VM only needs one when migration is necessary. Some short-lived VMs, like those providing function as a service (FaaS), may not need migration at all, as they boot quickly and shut down after serving the requested function. Thus, reducing or eliminating migration-related implementations help reduce the attack surface.

After identifying all additional components, we refactor codes to ensure that each fine-grained component can function independently along with the core parts. Finally, we create a configuration space with one configuration per component, allowing users to choose which components to include at build time. We achieve this using platform-specific tools or techniques such as the Kconfig language [37], the *define* keyword in C, or Rust's *features* feature [27].

3.4 Deploying Specialized VMMs in Cloud

In practice, users request VMs with specific CPU, memory, and device configurations. Rather than using the same general-purpose VMM image for all VMs, or receiving VMM images from clients, the cloud provider may create and manage specialized VMM images for different configurations. The VM orchestration system then can maps each requested VM configuration to a corresponding VMM image whose enabled features match the VM's requirements. For common configurations, providers can maintain a small repository of pre-built VMM images and launch the corresponding image directly, similar to existing VM image and template management.

Uncommon configurations do not require a new manual deployment path. An automated build service can generate the required VMM image from the requested feature set, cache it, and reuse it for future VMs with the same configuration. In our prototype, building such an image takes less than 27 seconds and uses up to 670 MB of memory on an Intel Xeon Silver server; this is a one-time cost per unique configuration. Moreover, our prototype reduces the VMM executable size by at least 50%, making the additional disk space needed to store multiple specialized executables negligible. We expect fewer than 100 pre-built images to cover common configurations, requiring less than 800 MB of storage in total. Since cloud providers already use shared storage to distribute images across servers, this storage cost and the occasional build-time memory overhead are small relative to the terabytes of memory and storage available in modern cloud deployments.

3.5 Establishing Intra-VMM Isolation

VMMs run as userspace processes in the host OS, contributing to the TCB, and increasing the attack surface. While reducing the VMM size can significantly reduce this attack surface, vulnerabilities in the remaining implementation can still pose serious security risks. For example, if a VM requires multiple I/O devices and one of them has a bug that allows an attacker to gain the privilege of the VMM process, the attacker may disrupt the execution flow or steal data from this device or other parts of the VMM. Isolation can prevent an untrusted component from directly accessing the private memory of other components. Partitioning sensitive data and code into isolated components within the VMM help limit the severity of an attack and preserve the confidentiality and integrity of other components' data.

What to isolate? To compartmentalize a VMM, we first identify which of its components we should isolate. For instance, virtual I/O devices are a major part of any VMM and often have vulnerabilities, as shown in Figure 3. Therefore, virtual devices are potential candidates for compartmentalization. According to principle **P3**, each I/O device should be isolated as they operate independently. For instance, a block device should be in a different compartment than a network device. According to principle **P4**, there should be isolation between multiple instances of the same type of device. For instance, if a VM needs two network devices, both instances should be isolated in the VMM to prevent a vulnerability in one network device from compromising another.

How to isolate? Prior protection key-based compartmentalizations [31, 44, 82, 89, 96] isolate libraries and the application code. For instance, there can be multiple compartments in FlexOS [44], each containing one or more libraries. When a compartment needs to execute a function in another compartment, the caller compartment's memory is disabled, and the callee compartment's memory is enabled. Upon return, memory permission is changed in the reverse order. Devices do not get invoked like library functions. Unlike functions performing an operation on the supplied arguments, each device instance has its own memory with sensitive information, which needs to be secured. Therefore, instead of passing a device, represented as a structure in procedural programming or as an object in object-oriented programming, from one compartment to another, the device should have its own isolated compartment.

Moreover, I/O devices perform asynchronous operations. Thus, disabling other I/O devices, i.e., their corresponding compartments, when one device is operating is not a viable option. This scenario demands fine-grained isolation. For example, when a device is instantiated, a VMM may create a unique protection key to label every memory location that belongs to and represents the device instance, essentially creating an isolated memory compartment for the instance. Any access to these memory locations will require a proper permission change; otherwise, it will be considered illegal.

Hardware extensions such as Intel MPK [21] and ARM MTE [3] can be utilized to implement such an isolation model. Hardware capabilities, such as CHERI ISA's capability [98], can also be used to compartmentalize the VMM, e.g., by assigning different capabilities to different VMM components. However, TrustZone [47], AMD SEV [77], and EPT/VM all utilize compartments as separate systems, communicating through RPCs and shared memory. Therefore,

using them to isolate VMM components may result in significant performance penalties, which goes against the principle **P5**. Additionally, Intel SGX is not capable of performing I/O, so isolating device drivers in this environment is not feasible.

When to enable/disable a device? Our goal is to keep memory in a VMM compartment disabled when not in use, preventing vulnerabilities in other compartments from propagating and executing I/O operations arbitrarily. Since I/O events are asynchronous, we make use of dedicated I/O event handlers to enable memory access for the relevant compartment only when an event is triggered, and then disable it immediately after performing the I/O operation.

3.6 Securing OVS-DPDK Vhost User

The vhost-user model allows offloading data plane from the VMM to a userspace application running on the host. Moving the backend to a separate process increases security by providing process-level isolation. Upon integrating the vhost-user with performance acceleration platforms such as DPDK [86], performance increases significantly. However, it creates an opportunity for a greater threat than the security gain from process-level isolation.

OVS-DPDK creates vhost-user instances for each guest VM connected to the application. To establish direct memory access between the vhost-user backend and the virtio frontend, the VMM shares the whole VM's memory with OVS-DPDK. This means that OVS-DPDK has direct access to the memory of all VMs connected to it, which makes OVS-DPDK an attractive target for intruders to manipulate attacks and disrupt VMs' execution or steal their data.

Any vulnerability in the host OS, VMM, OVS, or DPDK that gives an attacker full or partial control over DPDK can expose the memory of all VMs connected through the vhost-user interface. It may allow one VM user to read another VM's memory because DPDK already has access to the memory of all connected VMs. We propose restricting OVS-DPDK's access to guest memory by placing each VM's memory in a separate compartment and enabling access to that compartment only for the corresponding vhost-user instance. Any unauthorized access to a compartment is identified as an attack. For example, accessing another VM's memory through a buffer overflow or arbitrary code execution would trigger a protection fault or application reset, preventing the attacker from proceeding further. Isolation mechanisms such as tagged memory and capabilities can be used to create and enforce such compartmentalization.

3.7 Threat Model and Trusted Computing Base

In our threat model, any VM running on the host is considered untrusted by all other VMM processes because VM users can be malicious actors. Although some languages are considered safer than others, none prevent all exploits. For instance, Rust is more memory-safe than C. However, even code written in *safe* Rust can be vulnerable to future exploits [67]. Therefore, we do not consider any component of the VMM as part of the TCB unless it is formally verifiable such as seL4 [39] or certikOS [29]. The VMM's **core** components are relatively small and potentially formally verifiable. In contrast, we consider all **additional** components to be mutually distrusting, as each may have potential vulnerabilities that an attacker could exploit to attack other parts of the VMM.

The virtualization stack has several components that are critical to any VMM's functional correctness, including the host OS, hypervisor, compiler, linker, and loader. We consider these components to be trusted, as compromise of them can compromise the correctness of a VMM. The prototype presented in this paper leverages the Intel MPK [21] feature for compartmentalization. We trust the MPK APIs for enabling and disabling protection keys and assume that they work as intended.

The proposed security primitives are complementary to Trusted Execution Environments (TEEs) such as SGX [23], TDX [84], and SEV [77]. TEEs protect guests, enclaves, or trusted domains from an untrusted platform, whereas our approach hardens the VMM and accelerator components themselves by reducing unused functionality and compartmentalizing sensitive state. Thus, our approach targets the attack surface of regular VM deployments rather than replacing TEE-based protection.

TEE-based VMs can incur substantial overheads, reported to be up to 50% in some settings [12, 20, 43], and therefore require workload-specific trade-offs. For workloads that do not require TEE protection, our approach provides lower-cost hardening for regular VMs. For workloads that require protection from an untrusted host or hypervisor, TEE support can be combined with our mechanisms for defense-in-depth.

4 Prototype Implementation

Although any new or existing VMM, including QEMU, can benefit from the proposed design principles with some effort, our prototype, Redwood, is built on Cloud Hypervisor [1], which is an open-source VMM that runs on KVM [87] and Hyper-V [22]. Cloud Hypervisor is suitable for our target problem space because it is designed to run modern cloud workloads. It runs cloud OSs with most I/O handled by virtio devices and 64-bit CPUs. We chose this VMM over QEMU as it is fully written in Rust, which has better memory safety than C. In contrast, the effort to add the rust code to QEMU is relatively new and incomplete [69, 70]. Although Firecracker is also written in Rust, we chose Cloud Hypervisor for its broader support for general cloud OSs, rather than specific configurations like micro-VMs.

In this section, we discuss Redwood's implementation details for achieving a minimal VMM with hardware-based isolation. We also discuss the vhost-user device and VM memory isolation in OVS-DPDK. Our implementation effort encompasses 3672 lines of code, where isolation, modularization, multiboot, and DPDK changes contribute 508, 1894, 445, and 825 SLOC, respectively.

4.1 Workload-aware Redwood

Rust's package manager, *Cargo* [11], provides a mechanism called *features* for conditional compilation and managing optional dependencies. As discussed in Section 3.3, we identify the core and additional components in Cloud Hypervisor and refactor all additional components into named *features* in Redwood. Thus, users can enable and disable components at build time by choosing what to keep in the Redwood image.

Figure 4 shows Redwood's components. Green boxes denote core components required by all VMs, including CPU, memory, device managers, and VM-, ISA-, and hypervisor-level APIs. These managers create and manage VCPUs, memory, and devices for a

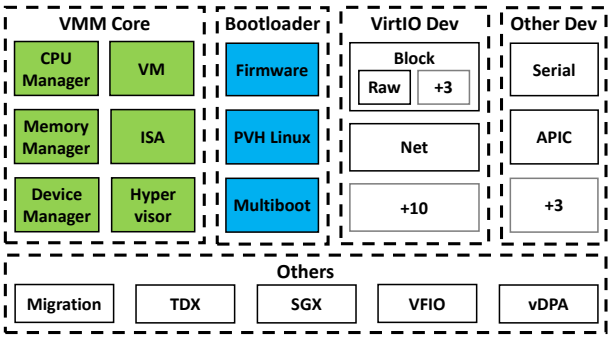


Figure 4: Core and additional components of Redwood.

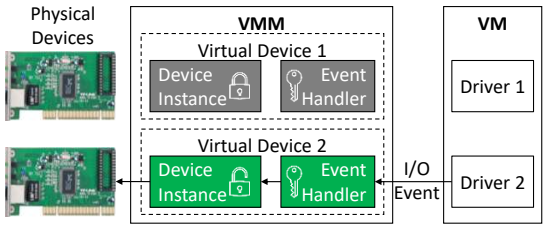


Figure 5: Intra-VMM compartmentalization. Each runtime virtual-device instance forms a separate compartment protected by a distinct pkey. The VMM core creates instances and manages permissions, while each event handler enables only its own compartment during I/O processing.

VM. The VM APIs perform VM creation, launching, destruction, etc. ISA (x86-64 and aarch64) and hypervisor (KVM and Hyper-V) supports were already configurable in Cloud Hypervisor.

Cloud Hypervisor supports booting OSs using firmwares, such as OVMF [60] and Rust Hypervisor Firmware [68], and direct booting of uncompressed Linux in 64-bit mode. Redwood adds multiboot supports for booting unikernels. As discussed in Section 3.3, a VM needs only one booting mechanism. By separating these three bootloaders into *features* (blue boxes), Redwood provides the flexibility for including only the required mechanism at build time.

Cloud Hypervisor implements 18 non-configurable I/O devices: including virtio devices, ACPI, IO APIC, and legacy devices such as RTC/CMOS, GPIO, and serial. Redwood resolves interdependencies, if any, between these I/O devices and makes all of them configurable at build-time. We also make disk implementations configurable, allowing a VMM to have specific types needed for a particular VM. Supported formats include raw, qcow, VHD, and VHDX.

Cloud hypervisor supports security features, like Intel SGX [23] and TDX [84], and device passthrough mechanisms like VFIO and vDPA. TDX and VFIO are configurable. It also supports live migration by default. Redwood separates the VM SGX, vDPA, and migration-related sources and makes them optional features. All additional components in Redwood are marked with white boxes.

4.2 Isolation Inside Redwood

Cloud Hypervisor implements 13 devices: virtio devices for network, block, console, balloon, memory, persistent memory, random number generator, watchdog, IOMMU, and socket, and vhost devices for network, block, and file systems. Redwood isolates all of these devices at runtime instance granularity. In our architecture, an *instance* is a concrete device object created for a VM; for example, two block devices attached to the same VM are two separate block-device instances. A *compartment* is the protected memory domain assigned to one such instance, not to an entire device class or the whole device manager. Therefore, a VM configured with two block devices and one network device results in three separate compartments, one per device instance.

Each device instance maintains sensitive private state that is crucial for correct I/O execution, including device features, queue metadata, event and interrupt state, driver state, backend configuration, guest-memory references, and handles to the underlying device. Each device type has additional device-specific information. For example, a block-device compartment contains the device ID, underlying disk information, and block-device configuration.

The device in the VMM and the driver in the VM communicate through asynchronous I/O operations. Therefore, the VMM creates a distinct event-handler thread for each device instance to process the corresponding I/O events. Since this handler accesses sensitive state, including virtqueues, guest-memory references, interrupts, events, and backend configuration, a compromised instance could otherwise give an attacker significant flexibility to disrupt execution or leak data. To limit this exposure, Redwood places each device instance's private state and event handler in a dedicated compartment by labeling the corresponding memory pages with a unique protection key during device instantiation.

The VMM core, including the CPU, memory, and device managers, remains outside these compartments and is responsible for creating device instances, assigning protection keys, and mediating permission changes.

Redwood utilizes Intel's MPK feature [27] to label memory pages and access control rights with protection keys *instance*. For every access to a tagged memory page, an additional permission check is performed using the PKRU register, which has access-disable and write-disable bits. System calls such as `pkey_alloc` and `pkey_free` are used to allocate and free *pkeys*, respectively. The `pkey_mprotect` system call is used to label a page with a key. In contrast, userspace functions `pkey_get` and `pkey_set` for getting and setting permissions, respectively, have a low overhead as they execute in a few cycles, requiring no context switching.

In the last step of device instantiation, Cloud Hypervisor creates an event handling thread, which waits for the I/O events from the associated driver in the VM. For such events, the handler performs the corresponding I/O operations and goes to the wait state after completion. Redwood disables access to all tagged memory locations by modifying the permissions using the corresponding `pkey` before the handler enters the wait state. As a result, no one can perform any I/O operations on the device in this state, which is illustrated by Virtual Device 1 in Figure 5. Any I/O event for the VM triggers the event handler, which then grants access to the memory locations of device instances using the `pkey`, as shown by Virtual Device 2 in Figure 5. This process enables the virtual devices to perform I/O. After completing I/O, and before returning to the wait

state, the event handler revokes memory access, reverting to the initial state akin to that of Virtual Device 1. This method effectively reduces the potential for an attacker to exploit the device's memory.

Any part of the VMM can modify the PKRU register's value. Prior works prevents such unauthorized writes through runtime checks [31] and static analysis [89]. Since Rust builds everything statically, except for glibc, no code is loaded after compilation in Redwood. Therefore, static binary analysis, coupled with strict W⊕X enforcement, where memory is either writable or executable but not both, is sufficient.

4.3 Unikernel Support for Redwood

Unikernels are workload-specific OSs that include only the components required by an application, reducing software footprint and attack surface. Redwood applies the same principle to VMMs by retaining only the components needed by a VM. Combining the two can therefore yield a highly specialized VM–VMM stack with an even smaller overall attack surface. Since many unikernels rely on Multiboot, which Cloud Hypervisor does not natively support, we add Multiboot support to enable them to benefit from Redwood's VMM specialization and Cloud Hypervisor's modern features.

Redwood first checks if the provided OS image supports the multiboot protocol by reading the OS image. Then, it loads the kernel's text, data, and bss sections into the guest VM's memory from the image. Next, it writes multiboot information (MBI) in VM's memory so that the OS can retrieve the MBI during execution. The bootloader also sets the GDT and IDT tables with the segment registers according to the multiboot specification. Finally, it configures other VCPU registers with multiboot magic, kernel entry point, MBI's start address, and flags. Redwood avoids any BIOS interaction to simplify and accelerate the mutiboot boot process.

4.4 Isolating Vhost User in OVS-DPDK

We can compare the vhost-user communication to the client-server model, where VMs are clients and OVS-DPDK is the server containing multiple clients' information. OVS-DPDK stores all the information of the corresponding drivers and the memory of the VMs. Fully or partially isolating DPDK components in a separate compartment using inter-domain isolation [31, 44, 82, 89, 94] is insufficient, because the compartment would still retain access to all VMs' memory. Thus, we isolate each VM's memory inside DPDK.

We tag each VM's memory with distinct *pkeys* and disable the access permission on all VM's memory by default. Access to a VM's memory is enabled only when a vhost-user needs to operate for a specific client and is disabled immediately after the operation is completed. As a result, an attacker cannot perform an arbitrary read/write on a guest's memory even if it gets access to OVS-DPDK. We implement the permission change operations as inline functions to avoid the overhead of a runtime abstraction interface.

4.5 Scalability and Extension to Other VMMs and Hardwares

The design principles are general, not Cloud Hypervisor-specific; Redwood is a prototype demonstrating their application to Cloud Hypervisor. Applying them to an existing VMM may require refactoring to identify core and optional components and expose suitable specialization and isolation boundaries, as done for Redwood. The concrete compartmentalization mechanism may vary with the ISA, hardware platform, and implementation language, while the higher-level principles—what to isolate, where to place boundaries, and when to enable or disable compartments—remain unchanged. Rust-based VMMs such as crosvm and Firecracker are particularly amenable to these techniques, while recent Rust integration efforts in QEMU [69, 70] may enable similar compartmentalization there, including for use cases involving CPU emulation. Implementing multiple prototypes is beyond the scope of this paper.

ARM's Memory Tagging Extension (MTE) [3] offers functionality similar to MPK, using tags associated with memory regions and pointers. On memory access, the CPU checks whether the pointer tag matches the memory tag. This enables Redwood to be extended to ARM CPUs with MTE support. However, we are not aware of any AMD CPUs supporting memory protection at such fine granularity.

In cloud environments, many VMs run on a single host OS, with one VMM process per VM. Therefore, Redwood does not face scalability issues with respect to the number of VMs or VMMs. Due to MPK limitations, each Redwood VMM supports up to 16 compartments, i.e., isolation for up to 16 I/O devices. If a VM requires more compartments, which is rare in typical cloud deployments, techniques such as pkey virtualization [61] could be used.

Table 1: Hardware configuration.

	Server
CPU	Intel Xeon Silver 4110 2.10GHz
Cores	32 (w/ HyperThreading)
L1/L2	32/1024 KB per core
L3	22528 KB
Memory	192 GB
Network	Intel X722 10-Gigabit
Storage	INTEL SSDSC2KB480G7 ATA 500GB SSD

5 Evaluation

We assess Redwood's security and performance. The security part includes VMM's CVE and attack surface analysis. The performance part identifies any VMM execution overheads due to our changes. We compare Redwood to the original Cloud Hypervisor VMM running Linux VMs on the KVM hypervisor. Our evaluation demonstrates that Redwood, enforcing isolation, matches the Cloud Hypervisor's performance for most cases. Runtime control flow integrity (CFI) techniques can prevent unauthorized permission changes on pkeys. These techniques are orthogonal to our work. Although we did not integrate them into Redwood, minimal performance impact is expected if integrated, as existing works [31, 89, 94] show negligible performance penalties.

Like Cloud Hypervisor, Redwood supports two hypervisors: KVM [87] and MSHV [54]. In our experimental setup, a server machine runs the KVM hypervisor that hosts guest VMs. The server machine runs Debian 13 with kernel 6.12.90+deb13-amd64. The guest VM is assigned 4GB of memory, 4 vCPUs and runs Ubuntu 24.04.4 LTS with kernel Linux cloud 6.12.8+. Table 1 shows the hardware configuration.

5.1 Security Analysis

CVE Analysis. Common mistakes, such as improper input validation, can introduce vulnerabilities like buffer overflows and enable attacks such as arbitrary code execution. The window between bug discovery and mitigation may allow attackers to steal sensitive data or disrupt systems. Isolation can limit attacker flexibility and contain exploitation of unresolved bugs within a VMM. Although Redwood is built on Cloud Hypervisor, its hardening techniques are applicable to other VMMs, including QEMU, and can mitigate classes of CVEs. In this section, we examine several such CVEs.

The VENOM [91] vulnerability (CVE-2015-3456), was discovered in 2015 but introduced in 2004. It allows a VM user to repeatedly fill a buffer in the emulated FDC in QEMU for up to 20ms, resulting in a buffer overflow. It permits adversaries to bypass, crash, or gain code execution on the host OS with QEMU process's privileges. Since most VMMs have root access to the host machine, the potential damage is severe, impacting other VMs' execution and enabling data theft. Redwood's isolation will deny (segmentation fault for Redwood) any attempt to overflow into other compartments, as access to memory in those compartments is disabled. This way, even without an available fix, the severity of the attack can be mitigated or significantly reduced.

FDC is initialized for every guest regardless of the configuration and cannot be removed or disabled. Whether a guest VM has a floppy disk or not, user-level access with sufficient permissions to communicate with the FDC I/O ports is all that is required to exploit this flaw. Redwood's specialization discards features like the floppy disk if they are not needed. It prevent adversaries from exploiting bugs, such as VENOM, present in default devices that are irrelevant to the VM.

We identified 128 QEMU CVEs related to various devices, including 43 related to virtio devices. For example, CVE-2021-3546 causes an out-of-bounds error on the vhost-user GPU device, potentially allowing a privileged guest user to crash the QEMU process on the host, leading to DoS or code execution. Enforced isolation can prevent attackers from exploiting such vulnerabilities to execute arbitrary code in another compartment's memory. We found 79 out-of-bounds errors in QEMU. Although QEMU is written in C, software written in memory-safe languages can also be susceptible to similar issues. For example, we found 370 CVEs related to out-of-bounds errors in software written in Rust.

CVE-2019-18960 [24] enables an attacker to steal, manipulate, and overflow memory of a rust-based Firecracker VMM; the overflow can enable arbitrary code execution with Firecracker process privileges. This demonstrates that even a VMM written in memory safe language is susceptible to memory-related attacks. Redwood defends against such attacks by isolating necessary I/O devices in separate compartments and discarding unnecessary implementations. Therefore, even if the security of one device (e.g., vsock) is compromised, an attacker cannot overflow into other device compartments. It's worth mentioning that these issues stem from the improper handling of raw pointers in Rust's unsafe blocks, which do not guarantee memory safety. Cloud Hypervisor and its dependencies contain ~5K lines of unsafe Rust code. However, it is often not possible to avoid using unsafe code in Rust.

DPDK and OVS have several critical CVEs. CVE-2020-14374 allows an attacker within a VM to exploit a buffer overflow vulnerability in the vhost-crypto DPDK application on the host, enabling arbitrary data writes in the process's address space and code execution. This vulnerability, stemming from a flawed bound check, allows an attacker to copy up to 4GB of application data with full control over the copied content. CVE-2020-14375, CVE-2020-14376, and CVE-2020-14377 pose similar threats through time-of-check time-of-use, buffer overflow, and buffer over-read vulnerabilities, respectively. OVS-DPDK maintains guest memory mappings for all VMs connected through vhost-user, so an adversary within any VM can potentially steal data from other VMs using these vulnerabilities. Our design only allows guest memory access when needed, so an attacker attempting to read from or write to another guest's memory would receive a segmentation fault. Consequently, these vulnerabilities will likely only cause the DPDK application to crash, rather than allow data theft. While our design cannot completely prevent future vulnerabilities, it significantly reduces their severity.

Defense against zero-day attacks. Several virtual disk formats facilitate the creation of a base image, upon which multiple disposable derived disk images can be instantiated [4]. These derived images use the base image as a backing file. The use of backing files and derived images is instrumental in the rapid instantiation of thin-provisioned VMs. When a VM instance is initialized, the VMM interprets the configuration files or command line parameters to determine the appropriate disk images. This process entails identifying both the backing files and derived images. For instance, the QCOW2 format embeds metadata within the derived image, linking it to its backing file. The VMM parses this metadata to establish the hierarchical relationship between the images. Typically, the VMM opens the derived image in read-write mode, allowing it to record modifications made by the VM, while the backing file is opened in read-only mode to maintain its integrity.

Several VMMs, including Cloud Hypervisor, support this feature [16]. However, a bug was identified in the implementation of backing files for the Cloud Hypervisor [14]. Specifically, their implementations allowed a disk image to reference a backing file that could, in turn, reference another backing file, or even the same initial file, creating the potential for unbounded recursion. This recursive referencing could result in a stack overflow when a disk image file references a backing file pointing to itself.

Moreover, many virtual disk images support hot-plugging, allowing disks to be added to VMs while they are actively in use. In the event of hot-plugging, if such a stack overflow occurs, a malicious user could potentially execute arbitrary code with the privileges of the VMM process. This vulnerability could lead to unauthorized access to the host system, compromising other VMs running on the same host, and enabling data hijacking or manipulation.

The bug was introduced in July 2023 [16], discovered by Cloud Hypervisor's continuous fuzzing tests in March 2024 [14], and remedied by limiting the maximum nesting depth for backing files in June 2024 [15]. Consequently, Cloud Hypervisor remained susceptible to this vulnerability for nearly a year. However, with the proposed compartmentalization and specialization, Redwood possesses a robust defense mechanism capable of mitigating attacks that exploit such implementation bugs, even in scenarios where the bug remains unidentified or the fix has not yet been applied.

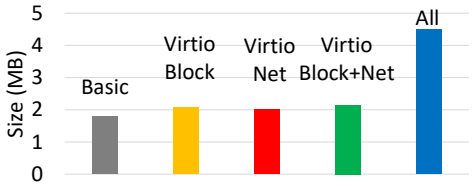


Figure 6: Image size, Redwood vs. Cloud Hypervisor (all).

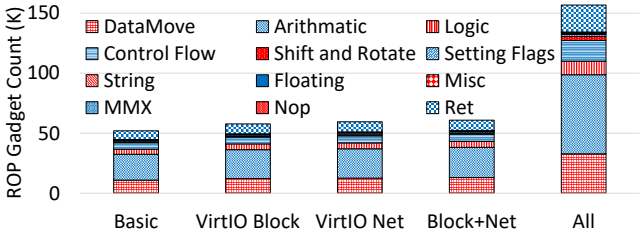


Figure 7: ROP gadgets, Redwood vs. Cloud Hypervisor (all).

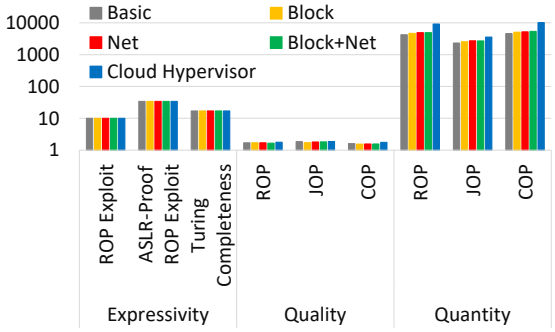
Image Size Reduction. We compare VMM image sizes across the default Cloud Hypervisor and various Redwood configurations. A basic Redwood configuration includes core VMM components, one boot-loading mechanism, and no virtio devices. Adding virtio devices, such as virtio-block, virtio-net, or both, slightly increases the image size. As shown in Figure 6, the basic Redwood configuration has a 50% smaller VMM image than Cloud Hypervisor, while the other three Redwood configurations are also significantly more compact. Redwood's specialization approach could similarly reduce image size in existing VMMs such as QEMU and Firecracker.

ROP Gadget Reduction. Reducing the number of ROP gadgets does not always improve security, but it can make it more difficult for an attacker to construct ROP chains to exploit a known vulnerability. Using Follner *et al.*'s methodology [28] and the Ropper tool [65], we measured ROP gadgets. Figure 7 shows the breakdown of ROP gadgets by category. The basic Redwood has 3x fewer gadgets than Cloud Hypervisor. Redwood with common devices (virtio-net and virtio-block) also has notably fewer gadgets. This shows VMM specialization's great potential for improving security.

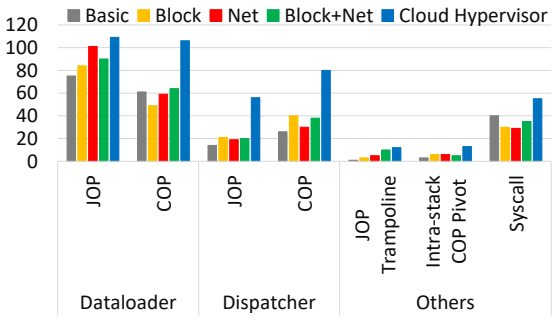
5.2 Gadget Set Analysis

Although Redwood variants contain significantly fewer ROP gadgets than Cloud Hypervisor, gadget count alone may not accurately reflect resilience to gadget-based attacks, since attackers may still exploit the remaining gadgets and some attack-surface reductions can introduce new ones. Brown *et al.* proposed gadget set analysis (GSA) [9] to assess the susceptibility of program variants to such attacks. Their GadgetSetAnalyzer performs static binary analysis and evaluates gadget sets based on their quantity, quality, and locality, providing a more meaningful measure of attack-surface reduction. We apply GSA to the default Cloud Hypervisor and multiple Redwood configurations, with results shown in Figure 8.

Figure 8a's results, plotted on a logarithmic scale, show that none of the Redwood VMMs increase the expressivity of the fundamental functional gadgets. However, the quality and quantity of ROP, JOP, and COP gadgets increase with the increase in components in a Redwood VMM, exhibiting the highest values for all default features enabled in Cloud Hypervisor. This suggests that fewer components help reduce gadget quality and quantity. Figure 8b shows that Redwood also reduces special-purpose gadgets that load data for and dispatch JOP and COP gadgets, as well as JOP trampoline, COP intra-stack pivot, and system call gadgets, compared to the default Cloud Hypervisor. In summary, Redwood's flexible configuration options can significantly reduce an attacker's freedom to exploit gadgets in the VMM.



(a) Gadget expressivity, quality, and quantity.



(b) Dataloader, dispatcher, and other gadgets.

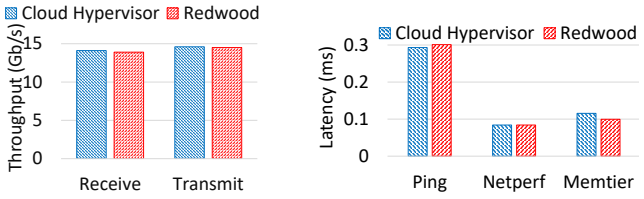
Figure 8: ROP gadget set analysis.

5.3 VirtIO Network Device Performance

In this section, we evaluate virtio-net in Redwood and Cloud Hypervisor using both microbenchmarks (iPerf3 [34], ping, Netperf [58], and memtier [53]) and macrobenchmarks (ApacheBench [85]). We measure network throughput and latency, as well as the performance of real-world applications, such as web and database servers, relevant to cloud workloads. Each server runs inside the VM, while the client runs on the host to generate traffic through the virtio-net device, without involving a physical NIC on the path.

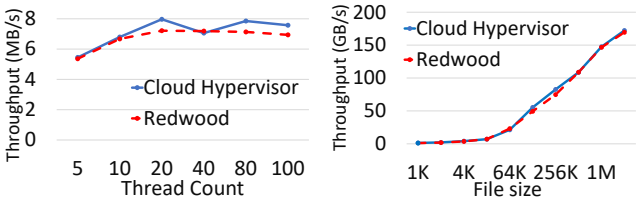
iPerf. We use the iPerf [34] micro-benchmark (v3.7) to measure network receive and transmit throughput over TCP. As shown in Figure 9a, the virtio-net performance in Cloud Hypervisor and Redwood indicates that there is no major overhead for MPK-based isolation for PV network devices.

Network Latency. Figure 9b compares the latency of isolated and regular virtio-net devices measured with various tools. Pinging the guest VM 100 times with a one-second interval yielded similar



(a) iPerf3 TCP throughput. (b) Latency.

Figure 9: Virtio-net performance.



(a) Varying concurrency. (b) Varying file size.

Figure 10: Apache server throughput.

latency for regular (0.29ms) and isolated (0.30ms) virtio-net devices. Netperf [58] reported 0.08402ms and 0.08399ms latency for regular and isolated devices, respectively, when sending 1,000 requests per second at even intervals to the guest VM. Memtier [53], when performing 100,000 SET and GET operations on Memcached [52] at a 1:10 ratio and 32B data size, incurred 0.11ms and 0.09ms latency for regular and isolated devices, respectively. These similar results for Cloud Hypervisor and Redwood indicate no effect on latency due to network device compartmentalization.

Apache. To evaluate an HTTP server, we run Apache (v2.4.41) and the Apache benchmark (v2.3) on a guest VM and client machine, respectively. The server data consists of randomly generated files of 1KB to 1MB sizes. The benchmark sends 100,000 requests and measures the server-side throughput. Each experiment is repeated 3 times. Figures 10a and 10b show the throughput for various thread counts when using 4KB files and for various file sizes when using 40 concurrent threads, respectively. In this test, the virtio-net device in Redwood performs similarly to the one in Cloud Hypervisor, meaning that the HTTP server and clients benefit from the added security without losing network throughput.

5.4 Virtio Block Device Performance

For evaluating virtio-block device performance, we use a RAW image file on an SSD and attach it to the VM as disk. We use dd [26] as a micro-benchmark and SysBench (v1.0.20) [83], a well-known file I/O benchmarking tool, as a macro-benchmark. For each run, we flush the read buffer and use total I/O size (16GB) that is larger than the main memory (4GB) so that we use the disk aggressively.

dd. We use `/dev/zero` and `/dev/null` devices as the read source and write destination, respectively, to minimize overheads. We run each experiment 3 times, transferring 1GB of data from/to the device each time. Figure 11 shows similar performance for Cloud Hypervisor and Redwood.

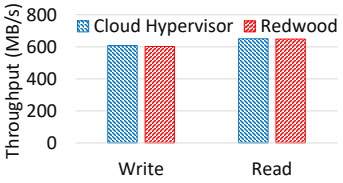
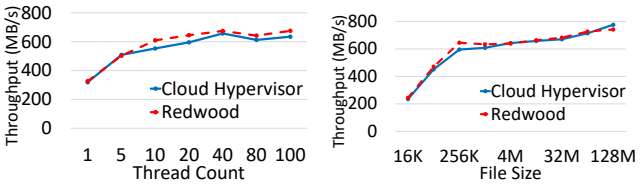


Figure 11: Storage throughput (dd).



(a) Varying number of threads. (b) SysBench file I/O throughput.

Figure 12: SysBench file I/O throughput.

SysBench File I/O. To measure file I/O performance, we use SysBench [83] to perform random reads and writes on a set of 100 files totaling 16GB. The ratio of read-to-write operations is 3:2, which reflects the prevalence of read operations in many real-world scenarios [32]. We run the same experiment with various numbers of threads (from 1 to 100) and block sizes (from 16KB to 128MB), each for 5 minutes.

Figures 12a and 12b show the throughput for different thread counts for 256KB block size and different block sizes with 20 threads, respectively. Cloud Hypervisor and Redwood have similar throughput, with average relative standard deviation (RSD) of 0.85% and 0.69% and latency of 25.63ms and 24.98ms, respectively.

5.5 MySQL database performance

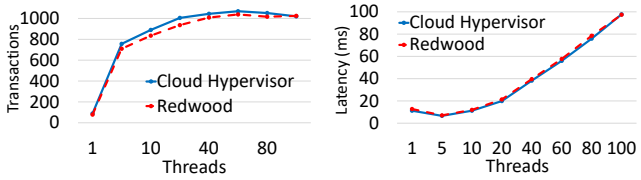
To evaluate both virtio-net and virtio-block performance on a real-world application, we choose MySQL [56] because, when deployed as a datacenter microservice, it exercises both network and storage operations. We set up a MySQL v8.0.46-0ubuntu0.24.04.2 database inside the VM with 16 tables, each containing 1,000,000 records. The dataset does not fit in memory; thus, the workload is not memory-bound and requires storage I/O.

We ran the sysbench [83] v1.0.20 Online Transaction Processing (OLTP) read-write benchmark from the host OS, sending queries over the network with different numbers of threads, ranging from 1 to 100. Figure 13a and Figure 13b show the transaction throughput and latency, respectively. As shown, Cloud Hypervisor and Redwood exhibit similar performance. The RSD is 0.027% and 0.052% for Cloud Hypervisor and Redwood, respectively.

5.6 Vhost User Network Device Performance

DPDK is known for providing better network performance for VMs running on the same host. Therefore, we measure network throughput using iPerf3 [34] and Redis [64] between two such VMs connected through OVS-DPDK vhost-user-net devices.

iPerf. We run the iPerf3 server on one VM and the client on another. Figure 14a shows the corresponding network throughput



(a) Throughput. (b) Latency.

Figure 13: MySQL transaction.



(a) TCP throughput. (b) Redis throughput.

Figure 14: OVS-DPDK vhost-user-net performance.

for receive and transmit operations. As can be seen, the network throughput is not affected by isolation enforced on the VM memory from the DPDK vhost-user side.

Redis. We use a VM to run the Redis (v5.0.1) server, a popular key-value store. A microbenchmark in another VM runs millions of SET and GET operations, with a pipeline size of 1000, with varying levels of concurrency. Each operation performs a read/write of 128MB of data using 64-bit keys. Figure 14b shows the number of SET/GET operations per second. Overall, the regular and isolated DPDK vhost-user-net exhibit similar performance. The RSD for regular and isolated DPDK is 0.00076% and 0.00063%, respectively.

6 Related Work

Minimal and specialized VMMs. Several VMMs reduce attack surface by limiting the features exposed to guests. Firecracker [76] targets microVMs with a small device model and a jailer, crosvm [79] uses minijail [55] and seccomp-bpf [73] to sandbox device code, and Cloud Hypervisor [1] focuses on modern cloud workloads while avoiding many legacy devices. These systems show the value of reducing VMM functionality, but they largely represent fixed design points chosen for specific deployment models. Redwood instead provides highly flexible fine-grained per-VM specialization.

Unikernel systems similarly reduce attack surface through single-purpose guest images [36, 38, 42, 49, 51]. Unikernel Monitors [99] extend this minimalism to the monitor, but target unikernel-specific execution. Redwood is complementary: it preserves compatibility with conventional cloud OSs while adding multiboot support for unikernels, allowing both guest-side and VMM-side specialization.

Hypervisor disaggregation and VMM sandboxing. Prior systems decompose virtualization stacks to reduce the impact of compromised components. Xoar [18] splits Xen's Dom0 functionality into service VMs, Murray et al. [35] move domain building out of Dom0, and Nexen [78] decomposes Xen using page-based isolation. They establish the benefits of separating virtualization components, but they are designed around Xen's bare-metal architecture. Redwood targets KVM/Hyper-V-style deployments, where each VM is managed by a user-space VMM process. Instead of disaggregating hypervisors, Redwood compartmentalizes VMM components.

Deployed VMMs also use OS mechanisms to restrict VMM privileges. QEMU can be confined using non-root execution, SELinux, AppArmor, cgroups, namespaces, and seccomp filters [2, 13, 57, 63, 72, 75]while crosvm uses minijail [55, 79]. These mechanisms restrict what a VMM process can do to the host. Redwood addresses a complementary problem: limiting what a compromised component can do within the same VMM address space.

Fast intra-address-space isolation. Prior works provide efficient compartmentalization within a process. SFI [74, 95] instruments memory accesses, while page-table-based approaches such as Wedge [7], Shreds [17], Dune [6], and Light-Weight Contexts [45] isolate components with stronger but more expensive domain switches. MPK-based systems such as intra-unikernel isolation [82], ERIM [89], Hodor [31], libmpk [61], FlexOS [44], and CubicleOS [71] show that hardware protection keys can support low-overhead in-process isolation. Redwood applies in-process isolation to the VMM boundary, where virtual devices mediate sensitive VM and host resources. It compartmentalize each device instances and enables access only during its asynchronous event-handling path, confining compromised device logic without process-level disaggregation. This paper also extends in-process isolation to the use of I/O acceleration frameworks, such as OVS-DPDK [62, 86], where one backend process may serve multiple VMs.

Confidential computing. Hardware features such as SGX [23], TDX [84], SEV [77], TrustZone [47], ARM MTE [3], and CHERI [98] provide complementary isolation. Confidential-VM technologies protect guest memory from an untrusted host or VMM, whereas this paper reduces and compartmentalizes the VMM attack surface exposed to untrusted guests. Thus, our approach does not conflict with these mechanisms and can be used alongside them.

7 Conclusions

Cloud infrastructures are heavily dependent on virtualization technologies. In this paper, we focus on the security risks associated with virtualization and derive several design principles and techniques to mitigate them. Our flexible VMM model reduces the attack surface. Our Intra-VMM isolation mechanism provides resilience against vulnerability exploitation in VMMs. Redwood, our prototype VMM, has fine-grained features, including 18 devices, to choose from when building a VMM for any particular VM and separates 13 virtio devices into distinct memory compartments. Our experimental evaluation reveals that Redwood protects against many severe CVEs without losing performance or compatibility with cloud OSs and reduces the attack surface by up to 50%. Moreover, it enables support for running unikernels.

Redwood is available at https://github.com/ssrg-vt/cloud-hypervisor/tree/redwood.

Acknowledgments

This work is supported by ONR under grants N00014-18-1-2022, N00014-19-1-2493, N00014-16-1-2104, and N0001416-1-2711.

References

[1] Cloud Hypervisor a Series of LF Projects LLC. Cloud Hypervisor. https://www.cloudhypervisor.org/, 2022.

[2] AppArmor. https://apparmor.net/, 2024. Accessed 01/14/2024.

[3] Armv8.5-A Memory Tagging Extension. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf. Accessed: 08/09/2023.

[4] QCOW2 backing files & overlays. https://kashyapc.fedorapeople.org/virt/lc-2012/snapshots-handout.html, 2024. Accessed 12/04/2024.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, 2003.

[6] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, Hollywood, CA, October 2012. USENIX Association.

[7] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '08, pages 309–322, USA, 2008. USENIX Association.

[8] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. IncludeOS: A minimal, resource efficient unikernel for cloud services. In *Proceedings of the 7th IEEE International Conference on Cloud Computing Technology and Science*, CloudCom '15, pages 250–257, 2015.

[9] Michael D. Brown and Santosh Pande. Is less really more? towards better metrics for measuring security improvements realized through software debloating. In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*, Santa Clara, CA, August 2019. USENIX Association.

[10] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. Hardware and software support for virtualization. *Synthesis Lectures on Computer Architecture*, 12(1):1–206, 2017.

[11] The Cargo Book. https://doc.rust-lang.org/cargo/, 2023. Accessed 05/22/2023.

[12] Bruno Casella. A performance analysis of vm-based trusted execution environments for confidential federated learning. In *2025 33rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 204–208, 2025.

[13] Control Groups. https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html, 2024. Accessed 01/14/2024.

[14] Stack overflow from opening qcow files with backing file pointing to itself. https://github.com/cloud-hypervisor/cloud-hypervisor/issues/6472, 2024. Accessed 12/04/2024.

[15] block: qcow: limit max nesting depth for backing file. https://github.com/cloud-hypervisor/cloud-hypervisor/pull/6482, 2024. Accessed 12/04/2024.

[16] Stack overflow from opening qcow files with backing file pointing to itself. https://github.com/cloud-hypervisor/cloud-hypervisor/pull/5573, 2024. Accessed 12/04/2024.

[17] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-Grained Execution Units with Private Memory. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71, 2016.

[18] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 189–202, 2011.

[19] QEMU contributors. QEMU: A generic and open source machine emulator and virtualizer. https://www.qemu.org/, 2022.

[20] Luigi Coppolino, Salvatore D'Antonio, Giovanni Mazzeo, and Luigi Romano. An experimental evaluation of tee technology: Benchmarking transparent approaches based on sgx, sev, and tdx. *Comput. Secur.*, 154(C), July 2025.

[21] Jonathan Corbet. Memory protection keys., 2015. https://lwn.net/Articles/643797/.

[22] Microsoft Corporation. Introduction to Hyper-V on Windows 10. https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/, 2022.

[23] Victor Costan and Srinivas Devadas. Intel SGX Explained., 2016. https://eprint.iacr.org/2016/086.pdf.

[24] CVE-2019-18960, 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-18960.

[25] CVE Details. https://www.cvedetails.com/, 2023. Accessed 02/01/2023.

[26] dd – convert and copy a file, 2020. https://man7.org/linux/man-pages/man1/dd.1.html.

[27] Features. https://doc.rust-lang.org/cargo/reference/features.html, 2023. Accessed 05/22/2023.

[28] Andreas Follner, Alexandre Bartel, and Eric Bodden. Analyzing the Gadgets. In *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems - Volume 9639*, ESSoS 2016, page 155–172, Berlin, Heidelberg, 2016. Springer-Verlag.

[29] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, OSDI'16, page 653–669, USA, 2016. USENIX Association.

[30] Irfan Habib. Virtualization with KVM. *Linux Journal*, 2008(166):8, 2008.

[31] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 489–503, USA, 2019. USENIX Association.

[32] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The art of multiprocessor programming*. Newnes, 2020.

[33] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 393–405, New York, NY, USA, 2016. Association for Computing Machinery.

[34] iPerf - The ultimate speed test tool for TCP, UDP and SCTP, 2020. https://iperf.fr/.

[35] Abdullah Issa, Toby Murray, and Gidon Ernst. In Search of Perfect Users: Towards Understanding the Usability of Converged Multi-Level Secure User Interfaces. In *Proceedings of the 30th Australian Conference on Computer-Human Interaction*, OzCHI '18, pages 572–576, 2018.

[36] Antti Kantee and Justin Cormack. Rump Kernels No OS? No Problem! *USENIX; login: magazine*, 2014.

[37] Kconfig Language. https://www.kernel.org/doc/html/next/kbuild/kconfig-language.html, 2023. Accessed 05/22/2023.

[38] Avi Kivity, Dor Laor Glauber Costa, and Pekka Enberg. OSv - Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Annual Technical Conference*, ATC '14, page 61, 2014.

[39] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, SOSP'09, pages 207–220, 2009.

[40] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 437–452, New York, NY, USA, 2017. Association for Computing Machinery.

[41] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. A Linux in Unikernel Clothing. In *Proceedings of the 15h European Conference on Computer Systems*, EuroSys '20, 2020.

[42] Stefan Lankes, Simon Pickartz, and Jens Breitbart. HermitCore: a unikernel for extreme scale computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '16, 2016.

[43] Jonghyun Lee, Yongqin Wang, Rachit Rajat, and Murali Annavaram. Characterization of gpu tee overheads in distributed data parallel ml training. *arXiv preprint arXiv:2501.11771*, 2025.

[44] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. FlexOS: Towards Flexible OS Isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pages 467–482, New York, NY, USA, 2022. Association for Computing Machinery.

[45] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, OSDI '16, pages 49–64, USA, 2016. USENIX Association.

[46] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, CCS '15, pages 1607–1619, New York, NY, USA, 2015. Association for Computing Machinery.

[47] ARM Ltd. Building a Secure System using TrustZone Technology, 2009. https://documentation-service.arm.com/static/5f212796500e883ab8e74531?token=.

[48] A Madhavapeddy, R Mortier, C Rotsos, DJ Scott, B Singh, T Gazagnaire, S Smith, S Hand, and J Crowcroft. Unikernels: library operating systems for the cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, 2013.

[49] A Madhavapeddy, R Mortier, C Rotsos, DJ Scott, B Singh, T Gazagnaire, S Smith, S Hand, and J Crowcroft. Unikernels: library operating systems for the cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'13, pages 461–472, 2013.

[50] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 218–233, 2017.

[51] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI '14, pages 459–473, 2014.

[52] Memcached, 2020. http://memcached.org/.

[53] NoSQL Redis and Memcache traffic generation and benchmarking tool, 2020. https://github.com/RedisLabs/memtier_benchmark/.

[54] Microsoft Corporation. Hyper-V Architecture. https://learn.microsoft.com/en-us/windows-server/virtualization/hyper-v. Accessed: 2026-06-06.

[55] minijail sandboxing and containment tool used in ChromeOS and Android. https://google.github.io/minijail/, 2024. Accessed 01/14/2024.

[56] MySQL, 2020. https://www.mysql.com/.

[57] Namespaces in operation, part 1: namespaces overview. https://lwn.net/Articles/531114/, 2024. Accessed 01/14/2024.

[58] Netperf Manual, 2020. http://www.cs.kent.edu/~farrell/dist/ref/Netperf.html.

[59] Ron Oglesby and Scott Herold. *VMware ESX Server: Advanced Technical Design Guide (Advanced Technical Design Guide series)*. The Brian Madden Company, 2005.

[60] Open Virtual Machine Firmware. http://www.linux-kvm.org/downloads/lersek/ovmf-whitepaper-c770f8c.txt. Accessed: 08/09/2023.

[61] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. Libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 241–254, USA, 2019. USENIX Association.

[62] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, 2015.

[63] Security. https://qemu-project.gitlab.io/qemu/system/security.html, 2024. Accessed 01/14/2024.

[64] Redis, 2020. https://redis.io/.

[65] Ropper, 2020. https://github.com/sashs/Ropper.

[66] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[67] Rust CVEs. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=rust, 2023. Accessed 05/15/2023.

[68] Rust Hypervisor Firmware. https://github.com/cloud-hypervisor/rust-hypervisor-firmware. Accessed: 08/09/2023.

[69] Rust in QEMU. https://www.qemu.org/docs/master/devel/rust.html, 2025. Accessed 07/14/2025.

[70] QEMU rust. https://github.com/qemu/qemu/tree/master/rust, 2025. Accessed 07/14/2025.

[71] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, ASPLOS '21, page 546–558, New York, NY, USA, 2021. Association for Computing Machinery.

[72] A seccomp overview. https://lwn.net/Articles/656307/, 2024. Accessed 01/14/2024.

[73] Seccomp BPF (SECure COMPuting with filters). https://www.kernel.org/doc/html/v6.4/userspace-api/seccomp_filter.html, 2024. Accessed 01/14/2024.

[74] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security '10, page 1, USA, 2010. USENIX Association.

[75] SELinux Project Wiki. https://selinuxproject.org/page/Main_Page, 2024. Accessed 01/14/2024.

[76] Amazon Web Services. Firecracker: Secure and fast microVMs for serverless computing. https://firecracker-microvm.github.io/, 2022.

[77] AMD Secure Encrypted Virtualization (SEV), 2022. https://developer.amd.com/sev/.

[78] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. Deconstructing Xen. In *NDSS*, 2017.

[79] Google Open Source. Book of crosvm. https://crosvm.dev/book/, 2024.

[80] Google Open Source. ChromiumOS. https://www.chromium.org/chromium-os/, 2024.

[81] SPDK Contributors. Storage Performance Development Kit (SPDK). http://spdk.io/.

[82] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-Unikernel Isolation with Intel Memory Protection Keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, pages 143–156, New York, NY, USA, 2020. Association for Computing Machinery.

[83] SysBench Manual, 2020. https://man7.org/linux/man-pages/man1/dd.1.html.

[84] Intel Trust Domain Extensions (Intel TDX)., 2020. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[85] The Apache Software Foundation. ab - Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.2/en/programs/ab.html.

[86] The Linux Foundation. Data Plane Development Kit (DPDK). http://dpdk.org/.

[87] Josh Triplett. Using the KVM API. *LWN*, 2015. https://lwn.net/Articles/658511/.

[88] Michael S Tsirkin, C Huck, and P Moll. Virtual I/O Device (VIRTIO) Version 1.1. *OASIS Committee: Burlington, MA, USA*, 2018.

[89] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient in-Process Isolation with Protection Keys (MPK). In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, pages 1221–1238, USA, 2019. USENIX Association.

[90] vDPA - virtio Data Path Acceleration . https://vdpa-dev.gitlab.io/. Accessed: 08/09/2023.

[91] The VENOM vulnerability, 2015. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3456.

[92] VFIO - "Virtual Function I/O" . https://docs.kernel.org/driver-api/vfio.html. Accessed: 08/09/2023.

[93] Virtual I/O Device (VIRTIO) Version 1.2. https://docs.oasis-open.org/virtio/virtio/v1.2/csd01/virtio-v1.2-csd01.html, 2022. Accessed 06/23/2024.

[94] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. You Shall Not (by)Pass! Practical, Secure, and Fast PKU-Based Sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, pages 266–282, New York, NY, USA, 2022. Association for Computing Machinery.

[95] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. *SIGOPS Oper. Syst. Rev.*, 27(5):203–216, dec 1993.

[96] Xiaoguang Wang, SengMing Yeoh, Pierre Olivier, and Binoy Ravindran. Secure and Efficient In-Process Monitor (and Library) Protection with Intel MPK. In *Proceedings of the 13th European Workshop on Systems Security*, EuroSec '20, page 7–12, New York, NY, USA, 2020. Association for Computing Machinery.

[97] Jon Watson. Virtualbox: bits and bytes masquerading as machines. *Linux Journal*, 2008(166):1, 2008.

[98] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, pages 20–37, 2015.

[99] Dan Williams and Ricardo Koller. Unikernel Monitors: Extending Minimalism Outside of the Box. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, HotCloud '16, pages 71–76, USA, 2016. USENIX Association.

[100] QEMU Virtio Vring divide by zero, 2017. https://vuldb.com/?id.110312.

[101] QEMU Plan 9 File System Time-Of-Check Time-Of-Use Privilege Escalation Vulnerability, 2021. https://www.zerodayinitiative.com/advisories/ZDI-21-159/.

[102] VENOM Zero-Day May Affect Thousands Of Cloud, Virtualization Products, 2015. https://www.darkreading.com/cloud/venom-zero-day-may-affect-thousands-of-cloud-virtualization-products.

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