




Security and Performance in the Delegated User-level Virtualization

Jiahao Chen^{1,2*}, Dingji Li^{1,2,3*}, Zeyu Mi^{1,2}, Yuxuan Liu^{1,2},
Binyu Zang^{1,2}, Haibing Guan⁴, and Haibo Chen^{1,2}

¹*Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University*

²*Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*

³*MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University*

⁴*Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University*

Abstract

Today’s mainstream virtualization systems are plagued by severe security threats due to the large attack surface exposed by in-kernel hypervisor components such as KVM. To address this issue, this paper proposes a novel design called delegated virtualization, which decouples the commodity hypervisor into two planes: the *hypervisor plane* for hypervisor control (which is typically small and has fixed logic) and the *VM plane* for handling virtual machine (VM) requests and exceptions at runtime. As our investigation reveals that all known hypervisor vulnerabilities that threaten the host kernel lie in the VM plane, delegated virtualization completely offloads the in-kernel VM plane to a user-space hypervisor called DuVisor that directly interacts with its VM without exiting to the host kernel, based on a small hardware extension (481 lines of Chisel). We have implemented the hardware extension on an open-source RISC-V CPU on FireSim and built a Rust-based DuVisor atop it. The evaluation results demonstrate that DuVisor significantly reduces the attack surface with negligible performance overhead (< 5%). DuVisor’s source code is publicly available at <https://github.com/IPADS-DuVisor>.

1 Introduction

The technique of system virtualization, also known as virtualization, is essential for efficiently running concurrent virtual machines (VMs). Since its conception, virtualization has undergone three rough stages of evolution, gradually moving hypervisor functions outside of kernel mode. In the first stage (Figure 1-a), all hypervisor functions were implemented in kernel mode to multiplex scarce resources of large mainframe machines, such as the IBM VM/370 [44, 48, 54, 83]. In the second stage (Figure 1-b), mainstream hypervisors began to offload hypervisor functions to user mode, which issued system calls to take advantage of a host operating system (OS) [42, 43] or a management VM [40]. However, some functions still remained in kernel mode, such as instruction


emulation and memory virtualization. The third stage (Figure 1-c) began with the release of hardware extensions (e.g., Intel VMX [19] and AMD SVM [2]), which further reduced the kernel involvement in virtualization by shifting some virtualization functions to hardware.

Nowadays, third-stage hypervisors that are based on hardware extensions typically utilize a split model consisting of two cooperative components: a kernel-mode module and a user-mode helper. For instance, the most popular Linux/KVM-based virtualization system¹ includes a global KVM kernel module [49, 61] and a per-VM user-mode helper, such as QEMU [26]. The KVM module interacts with hardware extensions and the host kernel, while the user-mode helper is responsible for VM management and I/O virtualization.

Unfortunately, vulnerabilities in the kernel-mode component of virtualization systems are discovered from time to time, making them a major threat to host security. For example, there have been more than a hundred CVEs reported in KVM during the course of its evolution [22]. These vulnerabilities can be exploited by a malicious VM to compromise the KVM component that interacts with the VM directly. The majority of these CVEs can be utilized to launch denial of service (DoS) attacks, causing host crashes and undermining the reliability of the host kernel as well as all co-located VMs [10, 16]. Even more concerning, once the KVM kernel module is hacked, the attacker may gain control of the entire system and carry out more severe attacks [3, 13]. In contrast, a compromised QEMU has a considerably smaller impact, usually limited to the current user-mode process and not affecting the host kernel or other VMs, thanks to the isolation between applications and kernel [24, 37].

The key idea of this paper is to **move all hypervisor components that directly interact with VMs at runtime to user space**, with the aim of minimizing the impact of any security bugs and reliability issues found in the user-mode hypervisor. However, there are three significant challenges

* Co-first authors.

 Corresponding author: Zeyu Mi (yzmizeyu@sjtu.edu.cn).

¹Xen [40] and VMware products [36, 88] also exhibit a similar architecture.

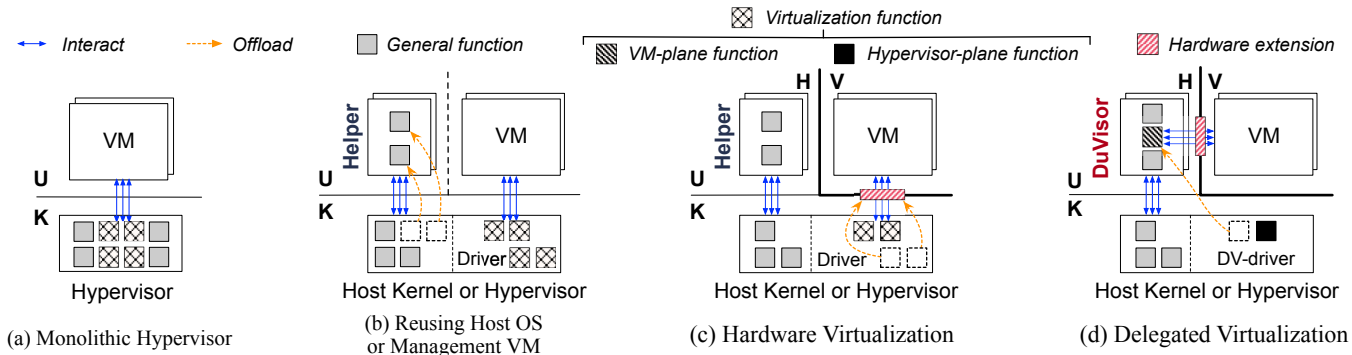


Figure 1: The architectural evolution of mainstream hypervisors (a → b → c) that gradually demote hypervisor functionalities out of kernel mode and the delegated virtualization proposed by this paper (d). *U* and *K* represent user mode and kernel mode; *H* and *V* are hypervisor mode and virtualization mode introduced by hardware virtualization. The *Virtualization functions* includes ① *VM-plane functions* serving VMs directly at runtime, and ② *hypervisor-plane functions* performing resource control and error handling for the VM plane. The *General functions* comprises other VM-required functions such as virtualizing I/O devices. (a) Stage-1: The monolithic hypervisor puts all functions in kernel mode; (b) Stage-2: Offloading some functions to a user-mode helper process (e.g., I/O backend drivers), which can reuse host OS or management VM that manage hardware resources; (c) Stage-3: Offloading some virtualization functions into hardware (e.g., shadow paging → nested paging); (d) We propose the DuVisor approach that delegates all VM-plane functions (e.g., VM exit handling) to user space and minimizes direct interactions between the host kernel and VMs at runtime.

that make the design more complex than it initially appears: 1) *Privilege Restriction*: modern hardware virtualization extensions are only configurable in kernel mode, such as setting a VM’s stage-2 page table. This necessitates the presence of a hypervisor component in the host kernel to use these extensions. 2) *Security Risk*: simply moving all the management of VMs’ hardware resources to user mode violates the least privilege principle and enlarges the attack surface. For instance, if QEMU is allowed to modify a VM’s stage-2 page table, it can then access any physical memory pages, posing a significant security risk. 3) *Performance Overhead*: most VM exits are now forwarded by the kernel to the user-mode functions to handle. Then, the control flow must return to the kernel again to resume VMs’ execution, resulting in excessive runtime ring crossings and unacceptable performance costs [56, 91].

We identify that the tight coupling between hardware virtualization extensions and kernel mode is the root cause of the challenges mentioned above. Fortunately, we observe that recent hardware advancements have made it possible to expose many hardware resources that were previously only accessible by the kernel to user mode. One prominent example is that Intel has released user-level interrupts, which allow a user-level process to handle physical interrupts [20]. Another example is physical memory checking, such as RISC-V Physical Memory Protection (PMP) [32, 63], which limits the physical memory range a program can access. With these recent hardware trends, we believe it is time to retrofit existing hardware virtualization extensions and expose virtualization interfaces to user mode securely and efficiently, addressing the challenges mentioned above.

This paper proposes a hypervisor design principle of **decoupling the VM plane from the hypervisor plane**. The *VM plane* frequently interacts with VMs at runtime by han-

dling their VM exits, and enables various virtual resources through instruction emulation, nested paging, device virtualization, etc. In contrast, the *hypervisor plane* serves the VM plane with physical resource control (e.g., invoking kernel interfaces to manage resources) and fatal error handling. Following this principle, we propose a delegated virtualization architecture, which eliminates the notion that the kernel mode should provide VM abstractions. Instead, delegated virtualization offloads the VM plane that interacts directly with VMs into per-VM hypervisors running in user mode, called *DuVisor*², while only leaving a tiny *DV-driver* in kernel mode responsible for the hypervisor plane.

As shown in Figure 1-d, we introduce a novel hardware extension called *Delegated Virtualization Extension (DV-Ext)* by slightly extending the existing hardware virtualization mechanism to securely expose hardware virtualization interfaces to user mode. Based on DV-Ext, all VM-plane functions in the existing hypervisor are offloaded to the user-mode *DuVisor* process. Specifically, *DuVisor* can directly utilize DV-Ext’s registers and instructions to serve runtime VM exits without trapping into the host kernel. On the other hand, the tiny *DV-driver* remaining in the kernel only wakes up occasionally to allocate physical resources or handle fatal errors for *DuVisor* processes.

DuVisor efficiently provides different virtualization functions in user mode with strong security guarantees. For CPU virtualization (§5.1), the *DuVisor* process creates a dedicated thread (*vthread*) for each virtual CPU (vCPU), and the *vthread* utilizes DV-Ext to handle this vCPU’s VM exits in user mode. For memory virtualization (§5.2), *DuVisor* configures a stage-2 page table for its VM and processes stage-2 page faults in user mode with a pre-allocated range of phys-

²Short for Delegated user-level HyperVisor

ical memory. The range used by a DuVisor and its VM is restricted by the DV-driver and DV-Ext via hardware physical memory checking. For I/O virtualization (§5.3), paravirtualized (PV) backend drivers in DuVisor directly communicate with their frontends in VMs. DuVisor further uses user-level posted interrupt to completely bypass the host kernel when sending notifications to its VM.

We have implemented DV-Ext based on a RISC-V Rocket CPU using FPGA. DV-Ext can be easily implemented by reusing existing hardware features, including hypervisor extension (H-Ext [28]) and user-level interrupts extension (N-Ext [33]). It only adds 481 lines of Chisel code. Based on DV-Ext, we use Rust to build DuVisor, and the code size is about 7K LoC. We also extend the Linux kernel v5.10.26 with a tiny DV-driver to cooperate with DuVisor by adding 337 LoC. Through software-hardware co-design, the kernel attack surface exposed to guest VMs is minimized, and any vulnerabilities that threaten existing hypervisors are now confined in the DuVisor process. Performance evaluation on cycle-accurate FireSim [59] shows that DuVisor incurs only negligible performance overhead for architectural operations and real-world applications.

In summary, the contributions of the paper are:

- We propose a delegated virtualization architecture that offloads the entire VM plane to the user-level DuVisor and leaves only a tiny DV-driver in the host kernel, minimizing the attack surface exposed to guest VMs and protecting the entire system from being impacted by the security and reliability issues in traditional hypervisors.
- We design a lightweight hardware DV-Ext that enables DuVisor to serve VM entirely in user mode.
- We implement the hardware extension on RISC-V with minimal modification and build a Rust-based DuVisor prototype.
- We evaluate the performance of DuVisor on AWS F1 FPGAs using cycle-accurate FireSim with a suite of real-world applications.

2 Background and Motivation

2.1 Hardware-assisted Virtualization

Mainstream hardware virtualization extensions [2, 4, 19, 28] offer comparable functionalities. We use RISC-V’s H-Ext as an exemplar due to its open-sourced implementations. As illustrated in Figure 1-c, H-Ext introduces two distinct modes, namely H mode and V mode, which are orthogonal to the existing privilege levels (U and K for user and kernel, respectively³). H mode is exclusively reserved for the hypervisor, while VMs operate in V mode. Only the hypervisor kernel mode (HK mode) is authorized to employ the virtualization interface for initiating, configuring, and resuming a VM, receiving traps from a VM to the hypervisor (also

³Although RISC-V kernel mode is referred to as “supervisor” (S) mode, we shall use the term kernel mode in this paper.

Table 1: CVE analyses of KVM [22] and Xen [38]. *Host* stands for the vulnerabilities that could lead to an attack on the host kernel, including *PE* (privilege escalation), *DoS* (denial of service), and *DL* (data leakage). *Other* refers to CVEs that only attack guest VMs or cannot be exploited. *LoC* shows the code size in LoCs of each hypervisor. *Time Scale* indicates the year strides for each of the two hypervisor CVE analyses.

Name	Total	Host			Other	LoC	Time Scale
		PE	DL	DoS			
KVM	111	21	7	52	31	142K	2008-2022
Xen	370	101	19	179	71	345K	2007-2022

known as VM exits), injecting virtual interrupts into a VM, and installing a stage-2 page table (S2PT). To control VMs, the helper process in hypervisor user mode (HU mode) must issue system calls to invoke functions provided by the kernel driver (e.g., KVM) in HK mode.

In this paper, we define the VM plane as all VM-serving functions that handle VM exits and virtualize resources at runtime, while referring to the hypervisor plane as the set of all hypervisor-serving functions that initialize the VM plane, manage physical resources, and handle emergency events. The VM plane in existing hypervisors spans the HK mode (e.g., CPU and memory virtualization) and the HU mode (e.g., device virtualization), whereas the hypervisor plane is located solely in the HK mode. Whenever a VM exit occurs, the hardware first switches the CPU control flow to the in-kernel VM plane. Most VM exits can be handled directly in the HK mode without switching to the user-mode helper. For instance, in the case of a stage-2 page fault (#S2PF), the hypervisor plane obtains a physical page, and the in-kernel VM plane inserts a new address mapping to the VM’s stage-2 page table before resuming the VM directly. However, other VM exits, such as some Memory-Mapped I/O (MMIO) trapings, cannot be entirely resolved by the in-kernel VM plane and must be forwarded to the user-level helper for emulation.

2.2 Vulnerabilities of Hypervisors

In contrast to the user-mode helper, which features a clear isolation boundary with the kernel, vulnerabilities arising from the in-kernel components of hypervisors pose significantly more severe threats to the host kernel. This is due to the fact that these components possess the highest level of privilege and interact directly and most frequently with VMs during runtime, thereby exposing a greater number of attack surfaces to VMs. The in-kernel components of hypervisors have accumulated a considerable number of publicly revealed vulnerabilities, underscoring their weak security and fault isolation. While there are also many vulnerabilities in the non-hypervisor components of the host kernel, this paper primarily focuses on hypervisor vulnerabilities, with non-hypervisor vulnerabilities being discussed in §7 and §9. Table 1 presents the statistics of disclosed vulnerabilities in KVM [22] and Xen [38], highlighting three key characteris-

Table 2: CVE classification based on subsystems of KVM. The data excludes 31 CVEs in the *Other* column of Table 1 that do not harm the host kernel. *Original* represents the original number of host-attacking CVEs in the KVM. *After Offload* means the number of host-attacking CVEs that remain in HK mode after offloading most components to HU mode by existing works [87, 91].

	Subsystem	Number of CVEs	
		Original	After Offload
VM Plane	Memory Virtualization	10	2
	Interrupt Virtualization	18	18
	ISA Emulation	19	10
	Para-Virtualization	4	0
	VM Exit Handling	17	17
	Device Virtualization	12	0
Hypervisor Plane	Hypervisor Initialization	0	-
	Resource Control	0	-
	Emergency Handling	0	-

tics of the vulnerabilities in the in-kernel components.

- **Large Vulnerability Quantity.** There are 111 and 370 disclosed CVEs in KVM and Xen, respectively. 72.07% and 80.81% of them cause kernel-level exploits, including information leakage, host DoS [10, 16, 84], and privilege escalation of a guest VM [3, 12, 13, 27, 82, 86].
- **Severe Security Threats.** Due to the high privilege of in-kernel components, their vulnerabilities can easily crash the entire system, disrupting the execution of other co-located VMs. As shown in Table 1, 65.77% and 75.68% of CVEs (i.e., PE and DoS) in KVM and Xen, respectively, can be exploited to launch DoS attacks via vulnerabilities such as NULL pointer dereferences [17] or out-of-bounds reads/writes [15]. Even worse, carefully crafted exploits may enable attackers to escalate a VM’s privilege and compromise the entire system, including all other VMs.
- **Low Exploit Costs.** CVE exploits can be researched and crafted at relatively low costs by well-financed experts who are motivated by the significant profit potential of successful attacks. For instance, it took just two months for a Google expert to develop an exploit that enabled VM escape via a vulnerability in KVM code [3].

2.3 Limitations of Deprivileged Execution

A long line of work has attempted to deprivilege the in-kernel functionalities of hypervisors to mitigate the threats of vulnerabilities in existing hypervisors [46, 80, 87, 91]. For example, NOVA [87] builds a microhypervisor based on the microkernel architecture. DeHype [91] endeavors to demote most parts of KVM into user mode while leaving a HypLet in kernel mode because the sensitive hardware virtualization instructions can only be executed in this mode. However, such deprivileged execution methods have two limitations:

Non-eliminable In-kernel Vulnerabilities. We investigated the 80 host-attacking CVEs of KVM from Table 1 and identified the subsystems in which they are present. As

Table 3: Breakdown of the latency of handling an MMIO read in QEMU/KVM on ARM, RISC-V and x86-64. *Kernel* represents the cycles spent on the in-kernel transfer operations. *User* stands for the cycles consumed by the I/O emulation and VM entry/exit.

Platform	Kernel	User	Total
ARM	4,323	1,596	5,919
RISC-V	3,135	4,067	7,202
x86-64	2,415	1,704	4,119

shown in the *Original* column of Table 2, these CVEs are distributed throughout all VM-plane subsystems, whereas none of them exist in the hypervisor plane. We further examined whether these CVEs could be addressed by prior works [87, 91] that attempted to offload as many VM-plane components as possible to user space. The CVE number that remains in the host kernel after offloading is displayed in the right-most column of Table 2. Unfortunately, the majority (58.75%) of CVEs cannot be eliminated because the VM-plane subsystems in which they reside must operate in the HK mode due to hardware privilege restrictions. For example, the interrupt virtualization subsystem accesses privileged registers, so it must remain in the HK mode. Similarly, some memory virtualization functions, particularly those configuring sensitive stage-2 page tables for VMs, must also remain in the HK mode. As a result, the in-kernel VM plane poses entrenched security and reliability risks to the host kernel.

Redundant and Costly Mode Switching. Because the hypervisor must use the kernel component to drive the hardware virtualization extension, moving most of the kernel component to user space will result in more frequent and expensive interactions between the VM and the user-level hypervisor due to the kernel’s involvement, leading to higher performance overhead. To understand the cost associated with kernel involvement, we break down the handling procedure of an MMIO read operation in QEMU/KVM to illustrate the VM-VMM communication cost and find that 73.04%, 43.53%, and 58.63% of CPU cycles are consumed by in-kernel transfer operations on ARM, RISC-V, and x86-64, respectively (Table 3). As a result, minimizing the host kernel part by delegating more kernel functions to user space [87, 91] will lead to significant performance overhead due to the expensive VM-VMM communication costs on each VM exit handling.

3 System Design Overview

In this paper, we introduce **delegated virtualization** to safeguard the overall security and reliability of virtualization systems by preventing compromised hypervisor components from directly breaching the host kernel. To circumvent the privilege restrictions of existing hardware virtualization, delegated virtualization simply exposes the existing virtualization interfaces to user mode without requiring intrusive hardware modifications. We explicitly decouple the VM plane from the hypervisor plane and offload all VM-plane func-

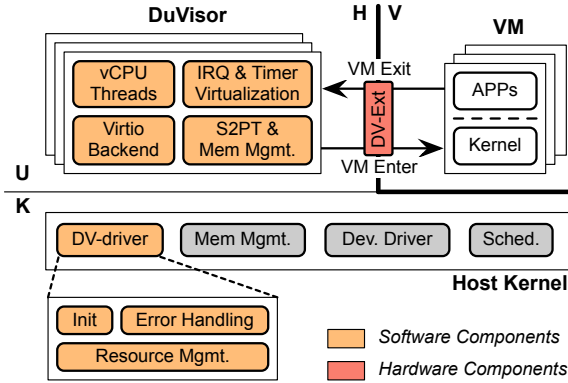


Figure 2: The architecture overview of DuVisor. *DV-Ext* is Delegated Virtualization Extension.

tions to the user-mode DuVisor. A tiny DV-driver in the host kernel serves as the hypervisor plane, which is removed from all runtime interactions (VM plane) between DuVisor and its VM.

The design of delegated virtualization offers two significant advantages. First, it minimizes the attack surface of the host kernel’s hypervisor components accessible to VMs and confines hypervisor vulnerabilities to user space, largely improving security and fault isolation between the VMs and the host kernel. Furthermore, the security and reliability benefits are obtained without any performance penalty due to the direct runtime interactions between a guest and its hypervisor.

The architecture of delegated virtualization, as depicted in Figure 2, comprises three primary components: the Delegated Virtualization Extension (DV-Ext), per-VM DuVisor hypervisor processes, and a global DV-driver in the kernel.

The Delegated Virtualization Extension (DV-Ext) must be installed on the hardware (§4). It empowers the host kernel to determine whether or not to delegate hardware virtualization functions to HU mode. If the delegated mode is enabled, the hardware virtualization interface can be accessed by unprivileged software without trapping into the host kernel. If it is not enabled, DV-Ext functions similarly to traditional hardware virtualization for compatibility.

An HU-mode DuVisor process leverages the hardware interface exposed by DV-Ext to control an unmodified VM (§5). To support the normal execution of a VM, DuVisor dynamically virtualizes physical resources to handle runtime VM exits. In this paper, this workflow is defined as the VM plane and is handled by the DuVisor process in HU mode. Moreover, to support memory virtualization in HU mode, the DuVisor process builds a stage-2 page table for its VM. If stage-2 page faults occur due to missing or illegal page mappings, DuVisor dynamically adds or updates mapping entries in the stage-2 page table. Like conventional hypervisors, DuVisor spawns distinct user-level threads for each vCPU, referred to as *vthreads*. It also supports PV I/O devices and virtual interrupts (including timers) for this VM. DuVisor can depend not only on the host kernel to manage external de-

Table 4: The registers and instructions added (or modified) by DV-Ext. The lowercase and uppercase names stand for registers and instructions. The registers starting with “hu” are accessible in HU mode while those starting with “h” can only be accessed in HK mode. The two instructions can be invoked in HU mode.

Type	Mode	Name	Description
Registers	HU	hu_er	VM exit reason
		hu_einfo	Additional information about a VM exit
		hu_vpc	IP address of a faulted vCPU
		hu_ehb	Base address of the VM exit handler
		hu_vcpuid	The vCPU ID running on a physical core
	hu_vitr	Virtual interrupt number to be inserted	
Instructions	HU	h_enable	Turn on DV-Ext
		h_deleg	Delegate VM exits to HU mode
		h_vmid	The VM ID running on a physical core
Instructions	HU	HURET	Resume the vCPU execution
		HUFLUSHGPA	Flush TLB entries associated with a GPA

vices like storage media and network cards but also control devices in HU mode by DPDK [18] to boost I/O virtualization.

A tiny DV-driver is inserted into the host kernel (§6) as the hypervisor plane, which occasionally participates in the management of physical resources for each DuVisor without interfering with runtime VM exits processing. Specifically, the DV-driver uses DV-Ext to enable/disable delegated mode and allocates resources (such as physical memory) for DuVisor processes. To mitigate security risks, it also restricts DuVisor’s physical memory view and handles emergencies, such as exceptions triggered by illegal physical memory accesses by untrusted VMs. DuVisor still relies on the host kernel to schedule all its threads and VMs.

Assumptions and Threat Model. We assume that the hardware (including DV-Ext) is correctly implemented and trusted. The goal of DuVisor is to defend the host kernel against malicious VMs, so that the host kernel and the DV-driver are trusted as well. However, in a multi-tenant cloud environment, a guest VM controlled by a hostile tenant may exploit vulnerabilities in DuVisor to compromise the hypervisor. Therefore, the user-level DuVisor process is considered untrusted by the host kernel. Side-channel attacks [73, 74] and corresponding defense methods [37, 45, 76] are orthogonal to the design of DuVisor and are not considered in this paper.

4 Delegated Virtualization Extension

In this section, we describe the design of DV-Ext, which lifts the restriction that hardware virtualization extensions are inaccessible to user mode. DV-Ext provides hardware interfaces to user-level DuVisor for obtaining VM information and controlling VM behaviors. These hardware interfaces can take different forms on varied hardware architectures. This section elaborates on register-based hardware interfaces as an example to present a detailed design of DV-Ext, which is suitable for the RISC-V and ARM architectures mentioned earlier. Additionally, we discuss the design of hardware interfaces based on memory and specialized instructions in §9 to

demonstrate DV-Ext’s universality. Table 4 shows the registers and instructions of DV-Ext.

HU-mode Registers and Instructions. We observe that certain privileged registers are only configured during hypervisor initialization and are rarely accessed at runtime. Therefore, these registers can be considered hypervisor-plane registers and are not exposed to HU mode. The remaining registers, however, are frequently used during VM runtime and are defined as VM-plane registers. Accordingly, DV-Ext allows HU mode to access these registers without restriction.

VM-plane registers, as shown in Table 4, are denoted by names beginning with “*hu*”. They are accessible in HU mode when HK mode activates DV-Ext by setting up the *h_enable* register. The VM-plane registers are classified into two categories. The first category records VM information for VM exits, such as *hu_er* and *hu_einfo*, which the hypervisor reads for handling VM exits. The second category controls the runtime behaviors of the hypervisor or VMs. For example, a hypervisor can configure *hu_vitr* to inject a virtual interrupt to a vCPU.

Delegatable VM Exits. DV-Ext provides delegatable VM exits (DVE), which enables a VM to immediately trap to its DuVisor process in HU mode. The kernel mode can configure DVEs by modifying the *h_deleg* register, whose individual bits regulate the delegation of specific types of VM exits. For instance, the DV-driver in HK mode can delegate stage-2 page faults and sensitive instruction faults such as WFI (i.e., wait-for-interrupt instruction for entering low-power standby CPU state, similar to HLT on x86) to HU mode by setting the corresponding bits in *h_deleg*. When a DVE occurs, the hardware searches for a hypervisor handler using the address specified in *hu_ehb*. DV-Ext additionally provides the *HURET* instruction for HU mode to resume VM execution after handling a DVE, and the entry point of the VM is stored in the *hu_vpc* register.

HU-mode Memory Virtualization. Since the register storing the base address of a stage-2 page table is rarely modified after a VM is booted, DV-Ext does not expose it to HU mode. However, in HU mode, the user-level hypervisor can still freely update stage-2 translation by exposing the in-memory stage-2 page table to HU mode. Therefore, the page table format is consistent with the original stage-2 page table used in HK mode. As HU mode needs to flush TLB entries after updating stage-2 translation, DV-Ext exposes a TLB maintenance instruction to HU mode as the *HUFLUSHGPA* instruction, which can flush TLB entries associated with a specific GPA and VMID.

Exitless Interrupt Virtualization. Posted interrupt allows the hypervisor to deliver virtual interrupts to a running vCPU without VM exit. DV-Ext enables user-level posted interrupt that DuVisor can directly utilize for injecting virtual external interrupts in HU mode. DuVisor should specify the receiving vCPU’s VCPUID and the interrupt vector in *hu_vitr*. Similarly, DV-Ext supports V-mode posted interrupt that a

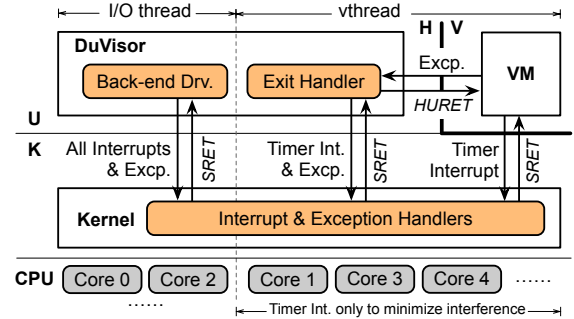


Figure 3: Exception and physical interrupt handling when running a DuVisor VM. I/O threads and vthreads can be dynamically scheduled on different CPU cores. To minimize the involvement of the host kernel as well as the interference of physical interrupts, the DV-driver can configure those cores that run vthreads (e.g., core 1/3/4) to trigger solely timer interrupts. For other physical interrupts such as external interrupts generated by I/O devices, it is natural to route them to cores that run I/O threads of the backend drivers (e.g., core 0/2).

vCPU can issue a virtual inter-processor interrupt (IPI) without VM exit by configuring *hu_vitr*. To prevent misdelivery, the hardware checks the VMID in *h_vmld* and the VCPUID information pre-configured by the DV-driver before triggering a virtual interrupt. An illegal operand triggers a fault into HK mode to wake up the DV-driver. DV-Ext also adds virtual timer interrupts that can be triggered without VM exit.

5 DuVisor Design

5.1 Handling VM Exits

VM exits are caused by either exceptions or physical interrupts. In existing virtualization systems, all VM exits are trapped to the in-kernel hypervisor component for handling, but this is not the case in DuVisor. In contrast, all exceptions that result in VM exits are sent to the user-level DuVisor, while physical interrupts continue to be trapped and handled by the host kernel. It is inappropriate to direct physical interrupts to HU mode (DuVisor) due to their vital importance to the host kernel for tasks such as scheduling and device management, as HU mode is not trusted.

Figure 3 illustrates how exceptions and physical interrupts are handled when running a DuVisor VM. During the preparation of the VM execution environment, I/O threads of the backend driver and vthreads are scheduled to different CPU cores. After preparation, vthreads in DuVisor execute an *HURET* instruction to enter V mode and start running guest code until a VM exit occurs. For VM exits caused by synchronous exceptions, the CPU control flow directly traps from the VM to DuVisor’s VM exit handler. The handler then determines the exception type by accessing corresponding HU mode registers provided by DV-Ext. After handling the VM exit, the vthread resumes the VM by executing *HURET* again.

On the other hand, VM exits caused by physical interrupts

are directed to the interrupt handler in HK mode. To ensure that the register states of DuVisor VMs are not corrupted due to scheduling, the DV-driver saves all DV-Ext-related registers before handling the physical interrupt and restores them before directly returning to V mode with an *SRET* instruction. Due to the small number of DV-Ext-related registers, the performance impact from this additional save/restore logic is minimal (§8.4).

To minimize interference, we configure the DV-driver so that only physical timer interrupts periodically occur on cores where vthreads are running. For physical external interrupts generated by physical devices, a PV I/O device in DuVisor relies on the in-kernel device driver to handle them. Hence, it is natural to direct these external interrupts to the same cores as the I/O threads of the backend driver. Moreover, if IOMMU [29] is available, a physical device can be passthrough into the VM for better performance, which sends posted interrupts that directly inject physical external interrupts to the VM without VM exit. Additionally, there are few physical IPIs between vthreads and I/O threads, thanks to the exitless interrupt virtualization (§5.3). Therefore, only physical timer interrupts may periodically trigger on cores running vthreads and bring the running VM into HK mode.

5.2 Restricted Memory Virtualization

In contrast to traditional hypervisors’ in-kernel stage-2 page table management, a DuVisor process handles stage-2 page faults and provides memory virtualization in HU mode without involving the kernel. To establish mappings of guest physical addresses (GPAs) for the VM, DuVisor populates stage-2 page table entries with host physical addresses (HPAs) in HU mode. Therefore, it requests the DV-driver to allocate contiguous memory regions with HPA information. Each stage-2 page fault traps to DuVisor, which then adds a free physical page from the pre-allocated memory region into the VM’s GPA space by updating its stage-2 page table.

One natural challenge is how to prevent the untrusted DuVisor from maliciously configuring the stage-2 page table to access arbitrary HPA. A malicious DuVisor, for instance, may allow its VM to access (or alter) sensitive data in another VM’s memory by directly mapping the attacker VM’s GPA to the victim’s HPA. Worse, the rogue VM can use this method to read and modify the host kernel memory. This issue can be addressed with a straightforward technique that requires DuVisor to manage a fake stage-2 page table instead of the real one. DuVisor only manages the fake stage-2 page table and must invoke system calls to ask the DV-driver to check this table and synchronize it to the real one. Although this method sounds reasonable, it frequently involves the kernel at runtime, leading to significant costs for memory-intensive workloads. Moreover, it complicates the memory management of the DV-driver.

We adopt an alternative approach, allowing DuVisor to manage the real stage-2 page table freely in HU mode

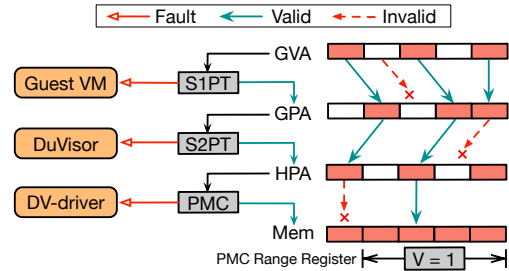


Figure 4: The translation and checking procedure of a memory access request issued from the V mode.

without entering kernel mode. Emerging hardware mechanisms, such as Intel TDX’s Physical Address Metadata Table (PAMT) [47], AMD SEV-SNP’s Reverse Mapping Table (RMP) [1], and ARM CCA’s Granule Protection Table (GPT) [5], can dynamically restrict access to physical memory. Take RISC-V Physical Memory Protection (PMP) for example, it checks every memory access against up to 64 PMP entries configured on each core. Inspired by this, we propose to utilize such physical memory checking (PMC) mechanisms to limit the physical memory range that VMs can access.

With PMC, we can allow DuVisor to configure its stage-2 page table optimistically. The MMU automatically checks whether the target HPAs of the VM’s memory accesses exceed the range limit of the pre-allocated physical memory regions. If so, the MMU triggers a fault to wake up the DV-driver. This design eliminates the stage-2 memory management module in the DV-driver. Furthermore, the overhead of dynamic checking is negligible since it is achieved by merely comparing offsets.

However, the current PMC technique is not specifically designed to restrict VM memory accesses. It examines every HPA access issued from the current physical core, which also restricts the host kernel and DuVisor from using physical addresses out of the configured entries. This is a significant constraint because the host OS can map the virtual addresses of the kernel and DuVisor to arbitrary physical memory addresses, which may exceed the ranges specified by the limited number of PMC regions. To overcome this constraint, DV-Ext extends the existing PMC mechanism slightly to make it work only for HPAs targeted by the V-mode memory accesses. DV-Ext adds a "Virtualization" (V) bit to each of the current PMC range registers. If the bit is set, the PMC only verifies the V-mode memory accesses according to the range registers.

Figure 4 illustrates how a guest virtual address (GVA) is translated into an HPA and finally reaches physical memory. The GVA is first translated into a GPA via MMU hardware according to the stage-1 page table (S1PT) built by the guest VM. Similarly, the GPA is translated into an HPA referring to the stage-2 page table controlled by DuVisor. A translation failure in the stage-2 page table triggers a stage-2 page fault into DuVisor for handling. Eventually, the PMC pre-

configured by the DV-driver checks the output HPA before accessing physical memory.

The DV-driver is responsible for allocating multiple physical memory regions for each VM, with each region being protected by a single PMP region. If the HPA exceeds the memory range specified by all PMP regions, the PMC will generate an exception and awaken the DV-driver to handle the situation. For example, the DV-driver may terminate the VM that triggered the exception and proceed to run other VMs. As the per-core PMP configurations are tied to a specific VM, the host kernel must save the PMP register values of the previous VM and install those of the next VM when switching between them.

5.3 I/O and Interrupt Virtualization

I/O virtualization in DuVisor works similarly to existing approaches. It supports PV (e.g., virtio) and emulated (e.g., tty) I/O devices for its VM. Although DuVisor is compatible with passthrough devices, its current implementation does not support them due to the lack of IOMMU on the RISC-V platform. However, we can easily support them when IOMMU becomes available (§ 9).

For each PV and emulated device, DuVisor spawns dedicated I/O thread(s) during VM initialization. These threads are responsible for responding to VM I/O requests and interacting with host I/O devices. For instance, a TX thread is launched for a virtio network device’s TX queue to handle network packets from the guest VM. To reduce the kernel attack surface, DuVisor can be combined with kernel-bypass virtio backends such as vhost-user. In particular, the RX thread keeps polling the NIC in HU mode to receive incoming network packets and notifies the guest VM through virtual external interrupts (vEXTs).

To enable efficient PV I/O notifications, DV-Ext supports directly injecting vEXTs into a running VM through user-level posted interrupt. Posted interrupt is the most efficient mechanism for interrupt virtualization, which allows a virtual interrupt to be injected into a running VM without triggering VM exits. However, on existing hardware, a hypervisor must enter kernel mode to send a posted interrupt, which means that communications between vCPUs and between the HU-mode helper and its VM must go through the host kernel, contrary to the design principle of DuVisor. By contrast, the user-level posted interrupt in DV-Ext does not require kernel participation. Specifically, the I/O thread injects vEXTs by writing the interrupt vector to the posted interrupt register in HU mode. If the target vCPU is running on a core, DV-Ext immediately triggers the vEXT on that core. Otherwise, DV-Ext records the vEXT information and does not deliver it until the target vCPU resumes execution.

The DV-driver assigns a unique VMID to each DuVisor during initialization and writes this VMID to the per-core *h_vmid* register every time a DuVisor is scheduled on a physical core. The VMID ensures that DuVisor’s I/O threads can

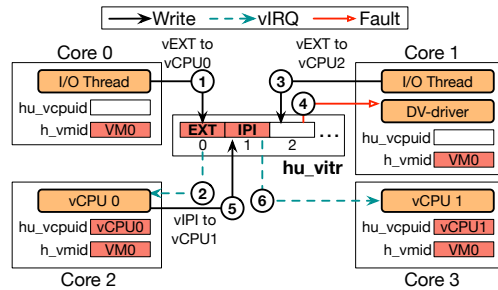


Figure 5: Exitless interrupt virtualization with user-level posted interrupt and V-mode posted interrupt. vEXT represents virtual external interrupt and vIPI represents virtual IPI.

only send posted interrupts to vCPUs with the same VMID as theirs. Since *h_vmid* is only accessible in HK mode, the DuVisor process and the guest VM cannot modify its value. Each vCPU has its VCPUID, which the guest kernel writes to *hu_vcpuid* during the boot process of each corresponding vCPU. The VCPUID is used by DV-Ext to identify the core on which the target vCPU is executing before delivering an interrupt.

Figure 5 depicts how a vEXT is delivered using user-level posted interrupts. For example, the I/O thread on core 0 attempts to insert a vEXT to vCPU 0 by writing the target vCPU’s associated location in *hu_vitr* ①. DV-Ext then finds that vCPU 0 is executing on core 2 and generates a vEXT on core 2 ②. If the sending thread attempts to send a wrong VCPUID or has a different VMID from the receiver, writing to *hu_vitr* will trigger a fault and wake up the DV-driver to handle this issue. For instance, the I/O thread on core 1 attempts to deliver a vEXT to a nonexistent vCPU 2 ③, and DV-Ext identifies this as an invalid operation and informs the DV-driver to handle the fault ④.

In addition to vEXTs, a multi-vCPU VM also requires efficient virtual IPIs (vIPIs) for inter-vCPU communications. To this end, DV-Ext supports V-mode posted interrupts that allow both sender and receiver vCPUs to incur no VM exit for a vIPI. Figure 5 also shows how a vIPI is generated using V-mode posted interrupts. The VMID and the VCPUID have the same effect as they do in user-level posted interrupt cases. Specifically, the vIPI issued from vCPU 0 on core 2 to vCPU 1 via writing *hu_vitr* ⑤ is triggered by DV-Ext on core 3, where vCPU 1 is running ⑥.

Furthermore, virtual timer interrupts (vTimers) are necessary for each DuVisor VM. DV-Ext supports directly firing an expired vTimer inside the VM. Currently, a DuVisor VM can receive vTimers without VM exits, but it must trap to the DuVisor to set up timer events. Although the hardware can further remove the VM exits of setting timer events, we do not consider it necessary because the infrequent vTimers have little impact on VM performance.

6 Implementation

6.1 DV-Ext Implementation

We chose the RISC-V platform to implement DV-Ext because it has rich open-sourced implementations of system-on-chip (SoC). We used a 5-stage in-order scalar processor, specifically the RISC-V Rocket Core [34], with a configuration of 16KB L1 ICache, 16KB L1 DCache, 512KB shared L2 cache, and 16GB DRAM.

DV-Ext does not require intrusive modifications to the CPU hardware to implement these VM-plane registers and instructions. Specifically, *hu_er*, *hu_einfo*, *hu_vpc*, and *hu_ehb* are aliases of *ucause*, *utval*, *uepc*, and *utvec* from RISC-V N-Ext (i.e., the user-level interrupts extension), and *HURET* is implemented based on N-Ext’s *URET*. The *HU-FLUSHGPA* instruction is implemented by exposing H-Ext’s *HFENCE.GVMA* to the HU mode. Similarly, *h_enable* and *h_deleg* are implemented by extending H-Ext’s *hstatus*, *hideleg*, and *hedeleg*. Therefore, most architectural implementations for these registers and instructions can be reused. Only *hu_vcpuid*, *hu_vitr*, and *h_vmid* are newly added registers for exitless interrupt virtualization.

Our DV-Ext implementation added 481 lines of Chisel to extend the existing H-Ext implementation [35] and support DVE. Additionally, we added 14 lines of Chisel to extend RISC-V PMP for VM memory restriction.

6.2 Software Implementation

Our prototype system of DuVisor consists of 7,128 LoC (5,052 lines of Rust, 166 lines of assembly, and 1,910 lines of C). The code of libraries we use is not included in the implementation effort. To implement the virtualization of CPU, memory and interrupt, 4,984 lines of Rust and 166 lines of assembly were written, in which the assembly is used to access architecture-dependent registers. For the virtualization of I/O devices, we ported the I/O backend of virtio block and virtio network devices from the *kvmtool* to DuVisor to reduce coding effort, accounting for 1,287 lines of C. We applied our design (e.g., the user-level posted interrupts) to these virtual devices as well as made some optimizations. Since there is no available DPDK support for RISC-V platforms currently, We also extended the virtio network backend with a user-space NIC driver using 623 lines of C to achieve a relatively fair performance comparison with KVM’s mature *vhost-net* backend. These I/O backend implementations comprise 1,910 lines of C in total.

We wrote a tiny Linux kernel module to work as the DV-driver and it has 337 LoC. DV-driver provides an *ioctl* system call for DuVisor to request several services. First, the DV-driver detects whether the hardware supports DV-Ext and enables it when a user process requests it. Second, it sets up the *h_deleg* register to configure DVEs. Third, the DV-driver allocates contiguous physical memory regions for DuVisor from Linux’s contiguous memory allocator (CMA) and pins

Table 5: CVEs in different KVM subsystems that can be successfully exploited in NOVA/DeHype and DuVisor’s architecture. We omit 31 CVEs that cannot attack the host kernel.

Subsystem	KVM			NOVA/DeHype			DuVisor		
	PE	DoS	DL	PE	DoS	DL	PE	DoS	DL
Memory Virtualization	3	6	1	1	1	0	0	0	0
Interrupt Virtualization	3	13	2	3	13	2	0	0	0
ISA Emulation	4	14	1	3	7	0	0	0	0
Para-Virtualization	0	4	0	0	0	0	0	0	0
VM Exit Handling	6	11	0	6	11	0	0	0	0
Device Virtualization	5	4	3	0	0	0	0	0	0
Sum		80			47		0		

the physical memory regions to ensure their availability at runtime. Before returning to HU mode, the DV-driver also dives into the M-mode OpenSBI and configures the PMP entries to restrict the VM’s physical memory access range. Each PMP entry is set up with a *pmpcfg* register specifying the V bit and memory accessibility as well as a *pmpaddr* register recording the physical address and length. The host kernel should also have a PMP fault handler that terminates the fault process gracefully. Currently, our prototype does not implement such a fault handler for simplicity, but it is not hard to extend the existing exception handler to implement one. Lastly, the DV-driver initializes a VMID for each DuVisor process that will be used by interrupt virtualization.

Based on the Linux kernel which already switches the general purpose registers and V-mode CSRs, we further modified the context switch logic (74 LoC) to save and restore the DV-Ext registers if the process has enabled DV-Ext. If the next scheduled thread is not a vthread from the same VM, the PMP registers are also switched.

7 Security Analysis and Evaluation

In this section, we analyze the overall system security of DuVisor from the perspective of an attacker.

Attack from Guest to Host Kernel. A hostile VM can exploit vulnerabilities to compromise the hypervisor. If these vulnerabilities are exploited in kernel mode, the attacker can achieve VM escape, steal sensitive kernel data, and even crash the entire kernel. Table 5 shows such CVEs in different KVM subsystems and how many of them can be successfully exploited in NOVA [87]/DeHype [91] and DuVisor’s architecture. NOVA and DeHype, limited by hardware, cannot fully move these subsystems to user mode, thus still leaving 58.75% of the vulnerabilities undefended. In contrast, DuVisor has deprivileged all of these subsystems in HU mode, reducing the host kernel’s attack surface and preventing any of these CVEs from directly jeopardizing it.

Table 6 lists typical vulnerabilities that we evaluated on DuVisor. Their security and reliability threats are confined within the DuVisor processes. Therefore, they cannot harm the host kernel directly.

Attack from Guest to DuVisor. In theory, a malicious guest can exploit vulnerabilities to attack the user-level DuVisor. To enhance security, DuVisor is developed in Rust, a

Table 6: Case studies of KVM CVEs. This table lists 6 representative KVM CVEs that have the potential to disrupt the normal execution of the host kernel, resulting in DoS or even more severe attacks. To evaluate the impact of these CVEs on a system running DuVisor, we emulated the vulnerabilities in the corresponding subsystems of DuVisor. The results show that these CVEs only cause DuVisor itself to crash, while the host kernel can continue to execute other programs (including DuVisor VMs) correctly.

CVE-*	Attack Effect
2017-12188	Memory Virtualization: KVM’s misconfiguration of the stage-2 page table allows the guest to access host memory, which can cause the host OS to crash or even be controlled.
2018-16882	Interrupt Virtualization: A use-after-free issue in the posted interrupt handling may allow hostile VMs to execute arbitrary code in HK mode.
2016-8630	ISA Emulation: Improper implementation for instruction decoding of KVM may cause host kernel crashes and jeopardize the availability of the host OS.
2020-8834	VM Exit Handling: KVM’s imperfect isolation of the guest states allows malicious VMs to corrupt the stack and destroy the availability of the entire system.
2016-5412	Para-Virtualization: The incorrect implementation of a hypercall in KVM could lead to an infinite loop that crashes the host kernel.
2019-6974	Device Virtualization: A use-after-free vulnerability in device virtualization may lead to VM escape in the kernel.

high-performance language that guarantees memory safety and thread safety. This greatly reduces the security risks associated with memory vulnerabilities [6, 7, 9, 12, 14] and threading vulnerabilities [8, 11, 14], such as the use-after-free bugs [9, 12] in traditional hypervisors written in C/C++. In addition, a vulnerable DuVisor can be promptly patched in user space without rebooting the host OS.

Attack from DuVisor to Host Kernel. Although the DuVisor design prevents a malicious VM from directly compromising the host kernel through the in-kernel hypervisor component, the VM may still attempt to attack the host kernel after controlling the DuVisor. Various existing techniques can be leveraged to defend against such user-level attacks, which are orthogonal to the DuVisor design. The static resource allocation and vhost-user devices in DuVisor significantly reduce the system calls. For example, DuVisor only requires 17 system calls to serve a Linux VM at runtime. Therefore, the kernel can use seccomp [24] to effectively restrict the system calls and their parameters that a DuVisor can invoke at runtime. The host kernel can also be reconstructed as a microkernel to improve its isolation, although this is beyond the scope of this paper.

Furthermore, neither the DV-driver nor the DV-Ext interface gives DuVisor additional capabilities to compromise the host kernel. First, the DV-driver has a small enough code base that it could be formally verified. Second, although the DV-driver allows user-level processes to request physical memory, it can still effectively isolate them with the help of the dynamic PMC mechanism. Third, the registers and instructions introduced by DV-Ext cannot be exploited by user-level code to attack the kernel. The *hu_er* and *hu_einfo* registers provide information related to DVE and do not leak any data from the host kernel. The *hu_vitr*, *hu_vcpuid*, *hu_ypc*, and *hu_ehb* reg-

isters, as well as *HUFLUSHGPA*, only control VM behaviors and have no effect on the host kernel. The *HURET* instruction and DVE implement mode switches between the HU mode and a VM, but cannot directly enter the HK mode.

8 Performance Evaluation

We answer the following four questions in this section:

Q1: How does the DuVisor compare to the KVM/QEMU in terms of hypervisor primitive cost? (§8.2)

Q2: How does the performance of applications running on DuVisor compare to that of KVM/QEMU? (§8.3)

Q3: What is the performance impact of DV-Ext on the co-located KVM/QEMU that does not use this extension? (§8.4)

Q4: How much performance impact does the extended PMP mechanism have on DuVisor’s performance? (§8.5)

8.1 Experimental Setup

We ran experiments on the cycle-accurate FireSim platform [59], which consists of two FPGA boards. Each FPGA board has eight RISC-V processors (3.2GHz, rv64imafdch), 16GB RAM, and 115GB storage. We created a local area network (LAN) between the two boards using 1Gbps IceNICs for network-related benchmarks. Both FPGA boards were controlled by an EC2 instance running CentOS 7.6.1810 on a 16-core Intel E5-2686 v4 CPU (2.3 GHz) and 240 GB RAM. We used OpenSBI v0.8 [31] as the firmware for RISC-V, and used Linux kernel 5.10.26 as the host kernel, which was equipped with the DV-driver. The baseline is KVM [23] (with H-Ext [35] support) and *QEMU* v7.0.0-rc0 running on Linux 5.16.

For **Q1** and **Q2**, posted interrupts can greatly improve the performance in interrupt-intensive scenarios, but the current open-sourced RISC-V hardware does not support this feature. To make a fair performance comparison, we extended DV-Ext’s interrupt virtualization support to the kernel level and implemented an optimized KVM/QEMU (“**KVM-OPT**”) that enables kernel-level posted interrupts. To answer **Q3**, we compared KVM with a slightly modified KVM (“**KVM-DVext**”) that was patched with context save/restore code related to DV-Ext and virtualization registers. For **Q4**, we compared DuVisor with a PMP-less version (“**DuVisor-noPMP**”) that removes PMP checking from the hardware.

8.2 Microbenchmarks

In this section, we quantify the performance of five frequently used hypervisor primitives. We leveraged the *cycle* CSR to measure CPU cycles. Figure 6 shows the average cost of the five operations in KVM and DuVisor. We calculated the average cycle count after recording the total time spent performing each operation 10,000 times. For three synchronous exceptions, we only compared DuVisor with KVM because there is no difference between KVM and KVM-OPT when interrupt virtualization is not involved. For the other two asynchronous exceptions, results of KVM, KVM-OPT

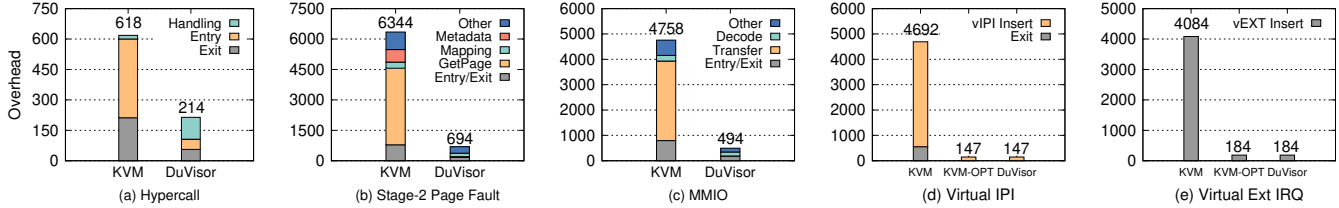


Figure 6: Breakdown of different hypervisor primitives (Unit: cycles). (a) shows a *null* hypercall. *Exit*: from invoking a hypercall in the guest VM to arriving at the hypercall handler in the hypervisor. *Entry*: the reverse procedure of *Exit*. *Handling*: processing in the hypercall handler. (b) shows a stage-2 page fault handling. *Entry/Exit*: from triggering a stage-2 page fault in the VM to arriving at the #S2PF handler, and the reverse procedure. *GetPage*: getting the available physical page for the fault GPA. *Mapping*: the PTE update in the stage-2 page table. *Metadata*: maintaining the metadata of the physical page to be mapped. *Other*: other logic such as lock protections and fault GPA checking. (c) shows an MMIO emulation. *Entry/Exit*: from invoking an MMIO operation in the VM to arriving at the user-space MMIO handler, and the reverse procedure. *Transfer*: transfers between the kernel in HK mode and the user-space VMM in HU mode, which DuVisor gets rid of. *Decode*: decoding the corresponding virtual MMIO device according to the fault address. *Other*: other logic, such as checking if the fault address belongs to the MMIO address range. (d) shows a virtual IPI sending. *vIPI Insert*: For KVM, from the hypervisor inserting the virtual IPI and kicking the receiver vCPU to the receiver vCPU’s arriving at the IPI handler. For KVM-OPT and DuVisor, from the sender vCPU’s writing *hu_vitr* register to the receiver vCPU’s arriving at the IPI handler. *Exit*: Only for KVM, from the sender vCPU’s invoking *SEND_IPI* hypercall to the hypervisor’s insertion, and from the receiver vCPU’s being kicked to it being inserted with the pending virtual IPI. (e) shows an I/O notification sending. *vEXT Insert*: For KVM-OPT and DuVisor, from the I/O thread’s writing *hu_vitr* register to the vCPU thread’s arriving at the IRQ handler. For KVM, from the I/O thread invoking the *SET_INTERRUPT* interface to the receiver vCPU’s arriving at the IPI handler.

and DuVisor are shown.

In the hypercall microbenchmark, both KVM and DuVisor ran a guest VM with a single vCPU pinned to a pCPU. The guest VM invoked a *null* hypercall, which trapped to the hypercall handler and then returned immediately without doing any functional operations. The number of cycles between the start of the hypercall and its return position was counted. As shown in Figure 6-a, DuVisor consumes 65.37% (404 cycles) less time during the hypercall procedure than KVM. This is because DuVisor in the user space does not need to perform operations that are only necessary in the kernel (e.g., enabling and disabling preemption and interrupts).

For stage-2 page fault handling, each hypervisor ran a guest VM with a single vCPU pinned to a pCPU. The guest VM read one byte from a page unmapped in the stage-2 page table, triggering a stage-2 page fault exception trapped to the hypervisor. The hypervisor allocated memory and established a valid mapping in the stage-2 page table before resuming the vCPU execution. We collected cycles before and after the guest VM read. As shown in Figure 6-b, DuVisor spends about 89.06% (5,650 cycles) less time compared with KVM. The main reason for the decreased cycles is that the KVM implementation is generic but more complex, whereas DuVisor can choose a dedicated but more concise implementation. According to our breakdown of the stage-2 page fault handling in KVM, most of the time is spent on getting the available physical page for the guest fault address, accounting for about 59.52% (3,776 cycles) of the total time as the *GetPage* part shows. Similarly, the *Other* and *Metadata* parts in KVM account for 23.31% (1,479 cycles) of the whole process, consisting of many generic Linux kernel logic, such as finding *virtual memory area (VMA)*, taking locks of *mmap* and maintaining metadata in Linux page structures. In com-

parison, DuVisor only spends 26 cycles in the *GetPage* part and 324 cycles in the *Other* and *Metadata* part.

For MMIO emulation, a single-vCPU guest VM pinned to a pCPU performed a load operation from an MMIO address of a virtual console device, which trapped to the user-level hypervisor and immediately returns. We counted the elapsed cycles of the MMIO read operation. The result shows that DuVisor takes 89.62% (4,264 cycles) less time than KVM primarily due to the shorter path of MMIO handling as shown in Figure 6-c. Traditional hypervisors such as KVM offload most MMIO emulations to user mode for security and reliability, which leads to a longer path than DuVisor. The breakdown shows that 65.89% (3,135 cycles) of the time during the MMIO emulation in KVM is spent on multiple world switches: VM(V) \leftrightarrow KVM(HK) \leftrightarrow QEMU(HU).

For vIPIs, both KVM and DuVisor ran a dual-vCPU guest VM and pin two vCPUs to separate cores. The sender vCPU sent an IPI and waited for the ACK from the receiver vCPU by polling on the shared memory, while the receiver vCPU wrote to the shared memory as soon as entering the IPI handler to inform the sender. We calculated the cycles on the sender vCPU from sending IPI to getting ACK from the shared memory. Figure 6-d shows the results. Since both KVM-OPT and DuVisor leverage hardware posted interrupt support, their virtual IPI processes are done without hypervisor involvement and thus equally cost 147 cycles. In contrast, the KVM spends 4,692 cycles on sending a vIPI. To send an IPI, the sender vCPU has to invoke a hypercall and trap to the KVM (*Exit* part), which occupies 11.83% of the total cost. While the rest 88.17% is cost by the cumbersome *vIPI Insert* part, in which the hypervisor sends the vIPI request, kicks the receiver vCPU, and inserts the vIPI before resuming the receiver vCPU.

For virtual external interrupts, a single-vCPU guest VM pinned to a pCPU spun and waited for external interrupts. We calculated the average consumed cycles from the hypervisor inserting a virtual external interrupt to the vCPU acknowledging the inserted interrupt in the interrupt handler. Using the same hardware posted interrupt support, both KVM-OPT and DuVisor averagely spend 184 cycles. While KVM needs 4,084 cycles in total due to the long emulation processes, such as kicking the vCPU.

8.3 Application Benchmarks

Table 7: Descriptions of application benchmarks.

Name	Description
Netperf	Netserver v2.6.0 on the local server (guest VM) and Netperf v2.6.0 on the remote client (native) to test the TCP stream throughput for 5 seconds.
iperf3	iperf v3.9 on both the local server (guest VM) and the remote client (native) to test the TCP throughput for 10 seconds.
Memcached	Memcached v1.6.10 running the memtier benchmark 1.3.0 on the remote client to test transactions per second. The thread number is set to the same as the number of server vCPUs. Each round of the test lasts 5 seconds.
Hackbench	Hackbench using Unix domain sockets and default 10 process groups running in 100 loops, measuring the time cost.
CPUPrime	CPU test in sysbench v0.4.12 that calculates prime numbers up to the max prime 10000. The thread number is set to the same as the number of server CPUs.

In this section, we evaluated the performance of five application benchmarks described in Table 7, compared KVM-OPT and DuVisor’s results with native, and analyzed the reasons for the performance differences. We ran the benchmark in three VMs with 1, 2 and 4 vCPUs, respectively. Every VM was equipped with 512MB memory, a virtio-based network device, and a virtio-based block device. In each case, we assigned the same number of CPUs and memory size to native as to VMs via *maxcpus* and *mem* using the kernel command line. To demonstrate the best performance of the in-kernel KVM, we used vhost-net as the network backend of the KVM-OPT VMs. For DuVisor VMs, we implemented a lightweight user-space network backend driver by porting *ixy* [51], a 1,000-LoC user-space ixgbe driver, to FireSim’s IceNet. Each vCPU and I/O thread of a guest VM was pinned to a separate physical CPU to avoid instability caused by the host kernel scheduler. All applications were evaluated after warmup to eliminate stage-2 page faults during benchmarks.

As shown in Figure 7-a and Figure 7-b, both KVM-OPT and DuVisor make full use of the NIC’s bandwidth with different vCPU numbers and have no significant overhead compared to native. Because KVM-OPT and DuVisor used different network backends, their performance when running network-intensive applications can vary due to backend implementations. Nevertheless, what we intend to demonstrate is that DuVisor can attain comparable performance to KVM-OPT’s mature vhost-net with a simply-implemented user-space network backend. For the network-intensive memcached application shown in Figure 7-c, DuVisor has a similar performance to KVM-OPT. However, they introduce up to 35% virtualization overhead when compared with the native. According to our analysis, the reasons for the perfor-

mance degradation mainly consist of the longer network data transfer path and the sub-optimal frontend/backend notifications. Massive small memcached requests travel longer than native before reaching the memcached in VMs due to the I/O virtualization. Besides, guest VMs’ interrupt frequency during the benchmark is much higher than that of the native due to the frequent notifications from the backend drivers, making memcached threads in VMs have less CPU time to process requests. We also compared QEMU/KVM with native on Intel and ARM platforms with similar configurations and find that they also introduce 15% to 40% overhead.

As shown in Figure 7-d, DuVisor is also comparable to KVM-OPT and native for hackbench. It is worth noting that KVM, which did not use the hardware interrupt virtualization, will incur about 12% more overhead in this experiment. The reason is that many IPIs are generated under this test, and the virtual IPI operation can be effectively accelerated by the posted interrupt, as shown in Figure 6-d. This also explains why DuVisor’s better microbenchmark performance results in no better application performance than KVM-OPT. Infrequent VM exits in DuVisor and KVM-OPT result in very low costs for hypervisor primitives (<5% CPU cycles), which are hardly observable in application benchmarks. Figure 7-e indicates that KVM-OPT and DuVisor attain the same performance as native execution in the CPUPrime benchmark.

As a result, the design of DuVisor does not introduce performance overhead compared with KVM-OPT, while achieving better host kernel security and reliability.

8.4 Impact on Co-located KVM VMs

When co-locating a traditional in-kernel hypervisor together with a user-level hypervisor, both hypervisors can independently configure registers related to virtualization, which can lead to VM state conflicts if not handled properly. Therefore, the host kernel needs to save and restore virtualization-related registers during context switches between them, introducing additional switching latency. To clarify how much impact such delay has on KVM, we evaluated and compared the application performance of KVM and KVM-DVext (with necessary context save/restore code). Figure 8 shows that there is no discernible performance difference between KVM and KVM-DVext, indicating that such additional switching latency due to DuVisor’s co-location has little impact on traditional VMs.

8.5 Memory Virtualization Overhead

Scaling Memory: To show DuVisor’s memory scalability compared with KVM-OPT, we ran memcached in a 4-vCPU guest VM with 512MB, 1024MB, 1536MB and 2048MB memory. As shown in Figure 9-a, DuVisor achieves almost the same performance as KVM-OPT in all cases. Compared with KVM-OPT, the memory virtualization of DuVisor differs only in that it places the stage-2 page table configuration in the user space and extends PMP for security checks. There-

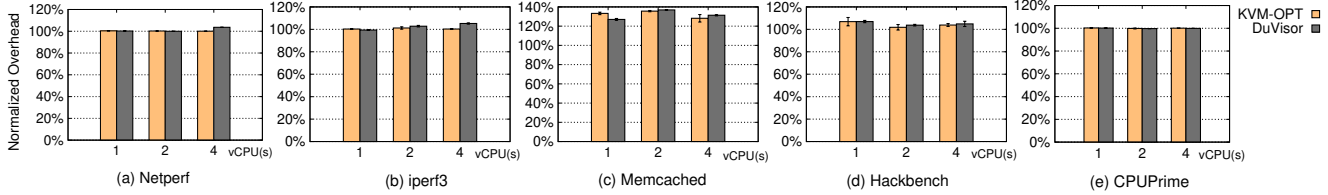


Figure 7: Normalized VM performance of real-world applications running in KVM-OPT and DuVisor. The Y-axis is the normalized overhead compared with the corresponding native environment. Error bars are added due to the performance fluctuation.

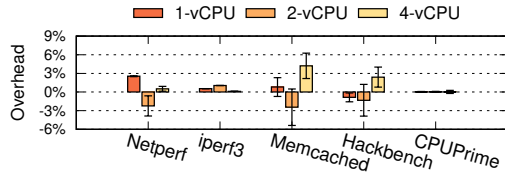


Figure 8: Normalized performance overhead of real-world applications of KVM-DVext compared with KVM.

fore, there is little impact on the memory access latency and thus scales well as the memory size grows.

Impact of PMP: DuVisor slightly extends the PMP hardware with a V bit to verify the validity of memory accesses from guest VMs. To evaluate whether this memory protection mechanism degrades the memory performance of the VM, we compared the memory test results of *lmbench* between DuVisor and DuVisor-noPMP in a dual-vCPU VM with 512MB memory. As shown in Figure 9-b, the memory bandwidth of the VM is almost the same with and without PMP checking. The result shows that the design of PMC does not introduce significant overhead to VMs.

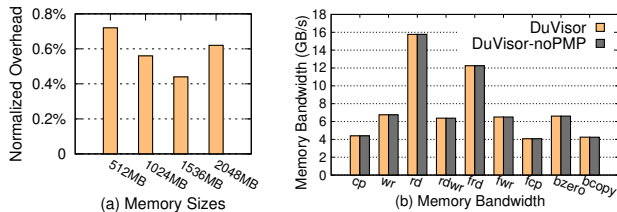


Figure 9: Performance of DuVisor's memory virtualization. (a) DuVisor's normalized overhead of different sizes of memory compared with KVM-OPT using memcached. (b) DuVisor's memory performance w/ and w/o PMC using *bw_mem* of *lmbench*.

9 Discussion and Limitations

Nested Virtualization. In traditional nested virtualization [41, 71, 77, 92], VM exits are intercepted by the L0 hypervisor (bare-metal one) before being handled by the L1 hypervisor (nested one), incurring tremendous runtime overhead. Prior work [72] optimizes nested VM exits via paravirtualization and passthrough. In the future, we plan to extend the idea of DVE to optimize nested virtualization by directly trapping VM exits to the L1 hypervisor.

Memory Utilization. DuVisor's PMP-based memory virtualization may lead to low memory utilization. To cover all VM memory with a limited number of PMP regions, we use the coarse-grained physical memory protection that can result in memory fragmentation and scalability issues. Addi-

tionally, it is difficult to support memory multiplexing (e.g., deduplication and overcommitment) among VMs based on PMP. Fortunately, such limitations can be mitigated by memory migration software mechanisms [65]. They can also be resolved through scalable fine-grained memory protection hardware mechanisms. For example, the RISC-V PMP table [25] has been proposed recently, which extends PMP to support physical memory restriction in page granularity.

IOMMU Support. Although IOMMU [29] is not yet supported on currently available RISC-V hardware, DuVisor is theoretically able to support it. Specifically, the stage-2 page table of IOMMU can be directly controlled by DuVisor, as the DV-driver can restrict access from guest devices by the IOPMP mechanism [30] with the V bit introduced by DV-Ext. The passthrough of the guest devices and the management of IOMMU's stage-1 page table within VMs is no different from traditional virtualization.

DV-Ext Universality. While the current prototype is implemented on RISC-V platforms, applying DV-Ext to other architectures would also be feasible, as they all share the same high-level virtualization functions. Consider Intel VMX hardware virtualization as an example.

CPU virtualization: Intel VMX directs all VM exits to the host kernel mode, and guest states in the in-memory VMCS can only be obtained and configured with the kernel-only VMREAD and VMWRITE instructions. To apply DV-Ext, Intel can just take VM exits and expose such VMX instructions to the DuVisor in the host user mode, while preventing access to host states (e.g., host CR3) in the VMCS.

Memory virtualization: The stage-2 page tables can be managed by the user-mode DuVisor without hardware modifications. However, PMP-like primitives are also necessary to enforce security. Fortunately, such hardware has emerged with confidential VM extensions [1, 21, 64, 69]. To apply DV-Ext, Intel can extend existing TDX's Physical Address Metadata Table (PAMT) [47] to provide fine-grained physical memory protection for DuVisor.

Interrupt virtualization: Intel can expose VMCS fields related to virtual interrupts to the DuVisor to deliver virtual interrupts. Specifically, virtual interrupts can be issued in user space by writing the VMCS with the user-mode VMX instructions mentioned above.

Host Kernel Vulnerability. Although DuVisor minimizes the attack surfaces of in-kernel hypervisors that are exposed to VMs, it does not completely exclude the host kernel from the VMs' runtime. Resource management (e.g., schedul-

ing and physical interrupt handling) involves non-hypervisor components in the host kernel, which may still contain many vulnerabilities, especially in mainstream monolithic kernels. Consequently, DuVisor is still vulnerable to the vulnerabilities of non-hypervisor components. Reconstructing it as microkernels may mitigate the problem, however, there are tradeoffs among compatibility, performance, and security.

10 Related Work

Moving Kernel Functions to Userspace. Deprivileging kernel features to userspace is a classic approach to enhance security, ease development, and improve performance. Microkernel is one typical design [52,62,70,78], where system services such as file systems and drivers run in user mode. For monolithic kernels, similar methods also exist, which implement the file systems [50,79], scheduler [57], network service [75] and sandbox [53] in user space. In terms of hypervisor functions, research focuses on reducing the hypervisor TCB by moving some of its functions to userspace, as demonstrated by designs such as DeHype [91] and NOVA [87]. Unlike DuVisor, they still require an in-kernel trusted module to perform VM-plane functions via hardware virtualization interfaces. DuVisor is the first system that entirely moves runtime VM-plane functions to user space, benefiting virtualization architectures on both monolithic kernels [40,61] and microkernels [55].

Securing VMs. Apart from the above solutions, many other studies have investigated how to achieve better isolation for VMs atop unreliable hypervisors. Numerous efforts have focused on reducing the hypervisor TCB [66,81,85,92]. CloudVisor [92] leverages nested virtualization to deprivilege the Xen [40] hypervisor. HypSec [66] separates a tiny CoreVisor as TCB from the KVM hypervisor. Other approaches have worked on hardening the hypervisor TCB, such as SeKVM [67,68], which use formal verification to ensure security guarantees of hypervisors. Unlike DuVisor, these solutions still rely on in-kernel hardware virtualization interfaces and incur modest performance overhead compared to unmodified traditional hypervisors.

Some solutions improve hypervisor reliability by providing per-VM hypervisor instances that are isolated from each other. Nexen [84] deconstructs the hypervisor into per-VM non-privileged service slices. HyperLock [90] decomposes the hypervisor into isolated shadow copies for each VM. In contrast to DuVisor, they still rely on traditional hardware virtualization interfaces and suffer from performance penalties of software isolation mechanisms. Others propose hardware extensions to remove the vulnerable hypervisor from the TCB [39,58,60,66,89]. NoHype [60] eliminates the hypervisor and its attack surfaces by static partitioning physical resources with hardware modifications. Nonetheless, it disallows resource oversubscription and is thus impractical for deployment in production scenarios.

Industrial confidential virtual machine (CVM) solutions,

such as AMD SEV-SNP [1] and ARM CCA [5,69], leverage specialized hardware security extensions to protect the data of VMs against a malicious hypervisor, which cannot access or taint the memory and registers of VMs. Both CVMs and DuVisor are vulnerable to non-hypervisor DoS attacks because they both rely on the host kernel. However, unlike CVMs, DuVisor can avoid DoS attacks due to in-kernel hypervisor vulnerabilities by deprivileging all VM-plane functions to user space to minimize the host kernel’s runtime attack surfaces exposed to VMs. On the other hand, CVMs require guest OS device driver modifications, while DuVisor supports unmodified VMs.

CVM and DuVisor are orthogonal techniques that can be combined for greater benefits. The design of DuVisor can be applied to defend against DoS attacks due to in-kernel hypervisor vulnerabilities for CVMs, which we plan to explore as future work. Existing CVMs are controlled by the in-kernel hypervisor through secure firmware interfaces (e.g., ARM CCA’s RMM and TF-A, Intel TDX module) that can only be invoked in the host kernel. DuVisor can be used alongside CVMs by exposing these interfaces to user space, thereby eliminating shared in-kernel hypervisor vulnerabilities.

11 Conclusion

We introduce the first delegated virtualization architecture that delegates all VM-plane virtualization functions to user space without trapping into the host kernel, minimizing attack surfaces exposed to VMs by in-kernel hypervisor components. To enable delegated virtualization, we present DV-Ext and DuVisor. DV-Ext is a hardware virtualization extension that securely exposes hardware virtualization interfaces to user space. DuVisor is a user-level hypervisor design that directly serves VM-hypervisor interactions in user space. We also present security techniques to prevent malicious use of DV-Ext. We have implemented a prototype for DV-Ext and DuVisor on the RISC-V platform. The security and performance evaluation results demonstrate that DuVisor protects the host kernel from hypervisor vulnerabilities without compromising performance compared to Linux KVM, establishing a new direction for secure virtualization research and development.

12 Acknowledgments

We express our sincere gratitude to our shepherd Jason Nieh for providing us with valuable suggestions that significantly helped in improving our paper. We thank the anonymous OSDI reviewers for their insightful suggestions. We are grateful to Yubin Xia and Rong Chen for their thorough and constructive comments that greatly improved this paper. We also thank Chenhui Ji and Yifan Tan for their contributions to the artifact evaluation. This work was supported in part by the National Natural Science Foundation of China (No. 62002218, 61925206, 62132014).

References

- [1] AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. <https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf>. Referenced May 2023.
- [2] AMD64 Architecture Programmer's Manual, Volume 2: System Programming. <https://www.amd.com/system/files/TechDocs/24593.pdf>. Referenced May 2023.
- [3] An EPYC Escape Case Study of KVM. <https://googleprojectzero.blogspot.com/2021/06/an-epyc-escape-case-study-of-kvm.html>. Referenced May 2023.
- [4] ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile. <https://developer.arm.com/documentation/102105/latest>. Referenced May 2023.
- [5] ARM CCA Hardware Architecture. <https://developer.arm.com/documentation/ddi0615/latest/>. Referenced May 2023.
- [6] CVE-2013-1796. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1796>. Referenced May 2023.
- [7] CVE-2014-0049. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0049>. Referenced May 2023.
- [8] CVE-2014-7842. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-7842>. Referenced May 2023.
- [9] CVE-2018-16882. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16882>. Referenced May 2023.
- [10] CVE-2019-19332. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-19332>. Referenced May 2023.
- [11] CVE-2019-6974. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-6974>. Referenced May 2023.
- [12] CVE-2019-7221. <https://bugs.chromium.org/p/project-zero/issues/detail?id=1760>. Referenced May 2023.
- [13] CVE-2021-22543. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-22543>. Referenced May 2023.
- [14] CVE-2021-29657. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-29657>. Referenced May 2023.
- [15] CVE-2021-4093. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-4093>. Referenced May 2023.
- [16] CVE-2021-43056. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-43056>. Referenced May 2023.
- [17] CVE-2021-8106. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-8106>. Referenced May 2023.
- [18] DPDK. <https://www.dpdk.org/>. Referenced May 2023.
- [19] Intel® 64 and IA-32 Architectures Software Developer's Manual. <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf>. Referenced May 2023.
- [20] Intel® Architecture Instruction Set Extensions Programming Reference. <https://software.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html>. Referenced May 2023.
- [21] Intel® Trust Domain Extensions (Intel® TDX). <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html>. Referenced May 2023.
- [22] KVM CVE. <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=KVM>. Referenced May 2023.
- [23] KVM RISC-V. <https://github.com/KVM-riscv>. Referenced May 2023.
- [24] Linux Seccomp. <https://en.wikipedia.org/wiki/Seccomp>. Referenced May 2023.
- [25] PMP Table Extension. <https://docs.google.com/document/d/158j99tm1gmZ5VH010scZhLKRU51JzJhj2zS2R1UWMeQ/edit#heading=h.rjobwmo1vft1>. Referenced May 2023.
- [26] QEMU: A Generic and Open Source Machine Emulator and Virtualizer. <https://www.qemu.org/>. Referenced May 2023.
- [27] QEMU-KVM Guest to Host Kernel Escape Vulnerability: vhost/vhost_net kernel buffer overflow. https://bugs.gentoo.org/show_bug.cgi?id=CVE-2019-14835. Referenced May 2023.
- [28] RISC-V Hypervisor Extension, Version 1.0.0-rc. <https://github.com/riscv/riscv-isa-manual/blob/master/src/hypervisor.tex>. Referenced May 2023.
- [29] RISC-V IOMMU Specification. <https://github.com/riscv-non-isa/riscv-iommu/blob/main/riscv-iommu.pdf>. Referenced May 2023.
- [30] RISC-V IOPMP Specification. <https://github.com/riscv-non-isa/iopmp-spec>. Referenced May 2023.
- [31] RISC-V OpenSBI, Version 0.8. <https://github.com/riscv-software-src/opensbi/releases/tag/v0.8>. Referenced May 2023.
- [32] RISC-V Privileged Architectures, Version 1.12. <https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf>. Referenced May 2023.
- [33] RISC-V "N" Standard Extension for User-Level Interrupts, Version 1.1. <https://five-embeddev.com/riscv-isa-manual/latest/n.html>. Referenced May 2023.

- [34] Rocket Chip. <https://github.com/chipsalliance/rocket-chip>. Referenced May 2023.
- [35] Rocket Chip H-Ext PR. <https://github.com/chipsalliance/rocket-chip/pull/2841>. Referenced May 2023.
- [36] The Architecture of VMware ESXi. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/ESXi_architecture.pdf. Referenced May 2023.
- [37] The Current State of Kernel Page-table Isolation. <https://lwn.net/Articles/741878/>. Referenced May 2023.
- [38] XEN CVE. <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=XEN>. Referenced May 2023.
- [39] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. HyperSentry: Enabling Stealthy in-Context Measurement of Hypervisor Integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS '10*, page 38–49, New York, NY, USA, 2010. Association for Computing Machinery.
- [40] 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*, page 164–177, New York, NY, USA, 2003. Association for Computing Machinery.
- [41] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10*, page 423–436, USA, 2010. USENIX Association.
- [42] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *SIGOPS Oper. Syst. Rev.*, 31(5):143–156, oct 1997.
- [43] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Trans. Comput. Syst.*, 30(4), nov 2012.
- [44] J. P. Buzen and U. O. Gagliardi. The Evolution of Virtual Machine Architecture. In *Proceedings of the June 4-8, 1973, National Computer Conference and Exposition, AFIPS '73*, page 291–299, New York, NY, USA, 1973. Association for Computing Machinery.
- [45] Sanchuan Chen, Fangfei Liu, Zeyu Mi, Yinqian Zhang, Ruby B. Lee, Haibo Chen, and Xiaofeng Wang. Leveraging Hardware Transactional Memory for Cache Side-Channel Defenses. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, ASIACCS '18*, page 601–608, New York, NY, USA, 2018. Association for Computing Machinery.
- [46] 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*, page 189–202, New York, NY, USA, 2011. Association for Computing Machinery.
- [47] Intel Corporation. Architecture Specification: Intel Trust Domain Extensions (Intel TDX) Module, Section 13. 2020.
- [48] R. J. Creasy. The Origin of the VM/370 Time-Sharing System. *IBM Journal of Research and Development*, 25(5):483–490, 1981.
- [49] Christoffer Dall and Jason Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '14*, page 333–348, New York, NY, USA, 2014. Association for Computing Machinery.
- [50] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and Protection in the ZoFS User-Space NVM File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19*, page 478–493, New York, NY, USA, 2019. Association for Computing Machinery.
- [51] Paul Emmerich, Maximilian Pudelko, Simon Bauer, Stefan Huber, Thomas Zwickl, and Georg Carle. User Space Network Drivers. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2019)*, September 2019.
- [52] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. The Design and Implementation of Microdrivers. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII*, page 168–178, New York, NY, USA, 2008. Association for Computing Machinery.
- [53] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2004, San Diego, California, USA*. The Internet Society, 2004.
- [54] P. H. Gum. System/370 Extended Architecture: Facilities for Virtual Machines. *IBM Journal of Research and Development*, 27(6):530–544, 1983.
- [55] Gernot Heiser and Ben Leslie. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *Proceedings of the 1st ACM Asia-Pacific Workshop on Workshop on Systems, APSys '10*, page 19–24, New York, NY, USA, 2010. Association for Computing Machinery.
- [56] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. A Case against (Most) Context Switches. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS '21*, page 17–25, New York, NY, USA, 2021. Association for Computing Machinery.
- [57] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. GhOST: Fast &

- Flexible User-Space Delegation of Linux Scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 588–604, New York, NY, USA, 2021. Association for Computing Machinery.
- [58] Seongwook Jin, Jeongseob Ahn, Sanghoon Cha, and Jaehyuk Huh. Architectural Support for Secure Virtualization under a Vulnerable Hypervisor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, page 272–283, New York, NY, USA, 2011. Association for Computing Machinery.
- [59] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. Firesim: FPGA-Accelerated Cycle-Exact Scale-out System Simulation in the Public Cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 29–42. IEEE Press, 2018.
- [60] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. NoHype: Virtualized Cloud Infrastructure without the Virtualization. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, page 350–361, New York, NY, USA, 2010. Association for Computing Machinery.
- [61] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux Virtual Machine Monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230. Dttawa, Dntorio, Canada, 2007.
- [62] 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 ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.
- [63] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the 15th European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.
- [64] Dingji Li, Zeyu Mi, Chenhui Ji, Yifan Tan, Binyu Zang, Haibing Guan, and Haibo Chen. Analysis and Optimization of Network I/O Tax in Confidential Virtual Machines. In *Proceedings of the 2023 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '23, Boston, MA, July 2023. USENIX Association.
- [65] Dingji Li, Zeyu Mi, Yubin Xia, Binyu Zang, Haibo Chen, and Haibing Guan. TwinVisor: Hardware-Isolated Confidential Virtual Machines for ARM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 638–654, New York, NY, USA, 2021. Association for Computing Machinery.
- [66] Shih-Wei Li, John S. Koh, and Jason Nieh. Protecting Cloud Virtual Machines from Hypervisor and Host Operating System Exploits. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*, pages 1357–1374, Santa Clara, CA, August 2019. USENIX Association.
- [67] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 21)*, pages 3953–3970. USENIX Association, August 2021.
- [68] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. A Secure and Formally Verified Linux KVM Hypervisor. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1782–1799, 2021.
- [69] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and Verification of the ARM Confidential Compute Architecture. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 465–484, Carlsbad, CA, July 2022. USENIX Association.
- [70] J. Liedtke. On Micro-Kernel Construction. *SIGOPS Oper. Syst. Rev.*, 29(5):237–250, dec 1995.
- [71] Jin Tack Lim, Christoffer Dall, Shih-Wei Li, Jason Nieh, and Marc Zyngier. NEVE: Nested Virtualization Extensions for ARM. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 201–217, New York, NY, USA, 2017. Association for Computing Machinery.
- [72] Jin Tack Lim and Jason Nieh. Optimizing nested virtualization performance using direct virtual hardware. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 557–574, New York, NY, USA, 2020. Association for Computing Machinery.
- [73] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD, August 2018. USENIX Association.
- [74] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, 2015.
- [75] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.
- [76] Zeyu Mi, Haibo Chen, Yinqian Zhang, Shuanghe Peng, Xiaofeng Wang, and Michael K. Reiter. CPU Elasticity to Mitigate Cross-VM Runtime Monitoring. *IEEE Transactions on Dependable and Secure Computing*, 17(5):1094–1108, 2020.

- [77] Zeyu Mi, Dingji Li, Haibo Chen, Binyu Zang, and Haibing Guan. (Mostly) Exitless VM Protection from Untrusted Hypervisor through Disaggregated Nested Virtualization. In Srdjan Capkun and Franziska Roesner, editors, *Proceedings of the 29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1695–1712. USENIX Association, 2020.
- [78] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the 14th EuroSys Conference 2019, EuroSys '19, New York, NY, USA, 2019*. Association for Computing Machinery.
- [79] Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, and Thomas E. Anderson. High Velocity Kernel File Systems with Bento. In Marcos K. Aguilera and Gala Yadgar, editors, *Proceedings of the 19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 65–79. USENIX Association, 2021.
- [80] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. Improving Xen Security through Disaggregation. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '08*, page 151–160, New York, NY, USA, 2008. Association for Computing Machinery.
- [81] Anh Nguyen, Himanshu Raj, Shravan Rayanchu, Stefan Saroiu, and Alec Wolman. Delusional Boot: Securing Hypervisors without Massive Re-Engineering. In *Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12*, page 141–154, New York, NY, USA, 2012. Association for Computing Machinery.
- [82] Gaoning Pan, Xingwei Lin, Xinlei Ying, Jiashui Wang, and Chunming Wu. Scavenger: Misuse Error Handling Leading To QEMU/KVM Escape. In *Black Hat Asia, 2021*.
- [83] L. H. Seawright and R. A. MacKinnon. VM/370—A study of multiplicity and usefulness. *IBM Systems Journal*, 18(1):4–17, 1979.
- [84] Le Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. Deconstructing Xen. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.
- [85] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko Kato. BitVisor: A Thin Hypervisor for Enforcing i/o Device Security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '09*, page 121–130, New York, NY, USA, 2009. Association for Computing Machinery.
- [86] Baibhav Singh and Rahul Kashyap. Back To The Future: A Radical Insecure Design of KVM on ARM. In *Black Hat USA, 2018*.
- [87] Udo Steinberg and Bernhard Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems, EuroSys '10*, page 209–222, New York, NY, USA, 2010. Association for Computing Machinery.
- [88] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation’s Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference (USENIX ATC 01)*, Boston, MA, June 2001. USENIX Association.
- [89] Jakub Szefer and Ruby B. Lee. Architectural Support for Hypervisor-Secure Virtualization. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII*, page 437–450, New York, NY, USA, 2012. Association for Computing Machinery.
- [90] Zhi Wang, Chiachih Wu, Michael Grace, and Xuxian Jiang. Isolating Commodity Hosted Hypervisors with HyperLock. In *Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12*, page 127–140, New York, NY, USA, 2012. Association for Computing Machinery.
- [91] Chiachih Wu, Zhi Wang, and Xuxian Jiang. Taming Hosted Hypervisors with (Mostly) Deprivileged Execution. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.
- [92] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-Tenant Cloud with Nested Virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles, SOSP '11*, page 203–216, New York, NY, USA, 2011. Association for Computing Machinery.

A Artifact Appendix

Abstract

The artifact evaluation of DuVisor contains two parts: the security evaluation and the performance evaluation. For security evaluation, we evaluate representative CVEs in the DuVisor on the QEMU-emulated RISC-V platform. For performance evaluation, we measure the performance of various microbenchmarks and application benchmarks on native, DuVisor, vanilla KVM and optimized KVM using the cycle-accurate FireSim platform.

Scope

Security Evaluation: DuVisor is able to prevent host kernel from crashing even if the user-level VM-plane is attacked. As mentioned in the Table 6 of our paper, this artifact emulates 6 representative KVM CVEs and evaluates their impact on the system. The results can show that these CVEs could crash DuVisor itself, but the host kernel can continue to execute other programs (including DuVisor VMs) normally.

Performance Evaluation: DuVisor achieves higher security while also maintains comparable performance to the optimized KVM (i.e., KVM-OPT in our paper). As shown in Figure 7-10 of our paper, this artifact compares various benchmarks between DuVisor and KVM, and also evaluates the impact of DV-Ext hardware extension KVM.

The results can show that DuVisor performs good and DV-Ext has little impact on existing KVM.

Contents

- **Run-time environment:** FireSim cycle-accurate FPGA platform based on AWS EC2 instances (two C5 and one F1)
- **Hardware:** QEMU (security AE) and RocketChip (performance AE)
- **Software:** OpenSBI, Linux, QEMU, DuVisor, related benchmarks
- **Metrics:** Benchmark results, usually latency and throughput
- **Estimated time:** about 20 hours with pre-built software images

- **Available on GitHub:** <https://github.com/IPADS-DuVisor/ae-guide/tree/main/>

Hosting

The artifacts are available on the GitHub, please refer to the main branch of this guide: <https://github.com/IPADS-DuVisor/ae-guide/tree/main/>

Requirements

Because the FireSim platform relies on special AWS FPGA (F1) instances, requiring multiple machines and complicated environment configurations. Besides, a newer version of FireSim platform may not be compatible with an older one. To simplify the AE procedure, we provided pre-configured AWS instances for reviewers.