A Hardware-Software Co-Design for Efficient Secure Containers

Jiacheng Shi, Yang Yu, Jinyu Gu✉, Yubin Xia
Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China

Abstract

Existing secure containers (VM-level containers) rely on virtualization hardware designed for general-purpose virtual machines, which leads to performance disadvantages compared with OS-level containers. The performance gap becomes much larger in nested clouds, where secure containers are deployed inside a VM.

This paper proposes CKI (Container Kernel Isolation), a hardware-software co-design for efficient secure containers, based on two insights. First, memory protection keys for kernel space (PKS) facilitates constructing a new privilege level for securely collocating multiple container kernels within the host kernel, without the involvement of virtualization hardware. Second, VM virtualization used by secure containers actually exceeds the demand of container virtualization. Thus, CKI avoids using the virtualization hardware for running container kernels and removes the unnecessary virtualization mechanism like two-stage address translation, reducing the overhead of secure containers.

Our experiments on real-world applications demonstrate the efficiency of CKI, reducing the latency of memory-intensive applications by up to 72% and 47% compared with state-of-the-art hardware-assisted virtualization (HVM) and software-based virtualization (PVM), respectively.

CCS Concepts: • **Software and its engineering → Operating systems**; • **Security and privacy → Virtualization and security**.

Keywords: Containers, Secure Containers, Memory Protection Keys, Operating Systems, Virtualization

ACM Reference Format:
Jiacheng Shi, Yang Yu, Jinyu Gu, Yubin Xia. 2025. A Hardware-Software Co-Design for Efficient Secure Containers. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3689031.3717473

EuroSys '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/25/03…$15.00
https://doi.org/10.1145/3689031.3717473

1 Introduction

Containers are widely used in the cloud for building and deploying applications, owing to their advantages like portability and scalability. There are two typical container architectures. OS-level containers have been criticized for poor security due to vulnerabilities in the OS kernel shared among mutually-untrusted containers. A malicious OS-level container may escape isolation by exploiting the large attack surface exposed by the syscall interface. In contrast, VM-level containers run each container on its own guest OS kernel for stronger security isolation, and gain more popularity in the cloud [15, 22, 29, 50, 56]. Compromising the guest kernel is harmless to the host kernel or other containers.

Despite many optimizations, VM-level containers still show performance disadvantages compared to OS-level containers due to the involvement of virtualization hardware designed for general-purpose virtual machines. For example, two-dimensional page table walk can increase the latency of memory-intensive applications by 46% on average [39, 69].

Moreover, the performance gap increases with nested virtualization [34, 45, 51, 52, 58, 65]. According to well-known cloud providers like Google and Alibaba, there is a growing demand for building cloud services based on public infrastructure-as-a-service (IaaS) clouds [9, 45]. In such a nested cloud, VM-level containers must run inside a VM, leading to high runtime overhead due to excessive context switches between the L2 VM (container), the guest hypervisor (L1 kernel), and the host hypervisor (L0 kernel). This overhead degrades the performance of memory-intensive and I/O-intensive applications by 28%∼226% and 1.8×∼4.3×, respectively, according to our evaluation.

We argue that the performance overhead stems from the mismatch between privilege levels needed by a secure container architecture and those provisioned by the CPU hardware. Specifically, the host kernel needs to isolate multiple container guest kernels, and each of them needs to isolate multiple container application processes. Thus, three privilege levels are needed. However, the CPU hardware usually offers two privilege levels for running the OS and applications, e.g., x86 ring-0/ring-3 and Arm EL1/EL0. Therefore, existing secure containers such as Kata Containers [15] and Firecracker [29] leverage hardware virtualization extensions

to obtain the extra privilege level, leading to performance overhead.

Some secure container architectures, such as PVM [45] and gVisor [3], eliminate the need for virtualization hardware. They deprivilege the container guest kernel to user mode and isolate the guest kernel and the container applications in separate address spaces. However, these architectures incur more context switch overhead because syscalls and exceptions inside the container must be redirected by the host kernel. For example, an empty syscall takes 90ns in an OS-level container and 336ns in a PVM container.

In this paper, we propose a new secure container design called CKI based on two insights. First, the recent CPU feature (Protection Keys Supervisor [14], PKS or MPK) can be retrofitted to create another privilege level within the kernel mode for accommodating guest (container) kernels. This new privilege level allows the guest kernel to efficiently serve its container applications [61], and thus brings performance benefits, e.g., minimizing the context switches in both bare-metal and nested clouds. Second, separating the kernel for each container is for security isolation instead of general-purpose virtualization. Thus, the unnecessary virtualization mechanism for supporting arbitrary VMs, i.e., two-stage address translation, could be removed to reduce overhead.

CKI faces three challenges in creating the new privilege level. First, PKS is designed solely for memory isolation while a malicious guest kernel may execute arbitrary privileged instructions as it executes in kernel mode. To address this, we propose lightweight hardware extensions to PKS for instruction isolation. Second, the security target of the PKS switch gates in CKI, i.e., preventing container escape or DoS, requires additional designs beyond basic MPK gates [64]. For example, the gates require new mechanisms for locating the per-vCPU area and preventing interrupt monopolizing or forgery. Third, PKS supports only 16 memory domains within one kernel address space, whereas a machine may host dozens to hundreds of secure containers. To support an arbitrary number of containers, CKI combines PKS and address space isolation for isolating different guest kernels. Specifically, it creates a separate address space for each guest kernel. It maps a kernel security monitor (KSM) in each address space and uses PKS to isolate the KSM from the guest kernel. The PKS isolation deprivileges the guest kernel so that it can perform privileged operations only through the interfaces provided by the KSM or the host kernel. The KSM implements the privileged operations that only access the private data of one secure container (e.g., page table updates), which can be invoked through a fast PKS switch gate. We only map these private data in the KSM, so that side-channel protections (e.g., PTI [21], IBRS [13]) can be eliminated from the PKS switch gate, which saves hundreds of CPU cycles [33].

We implement a prototype of CKI and evaluate it with real-world container applications. We evaluate CKI in both bare-metal and nested clouds, comparing it with hardware-assisted virtualization (HVM) [15] and software-based virtualization (PVM) [45]. In a bare-metal cloud, CKI reduces the latency of memory-intensive applications by up to 18%/47% compared with HVM/PVM. In a nested cloud, CKI reduces the latency of memory-intensive applications by up to 72%/47% compared with HVM/PVM, and obtains up to 6.8×/1.2× throughput for I/O-intensive applications compared with HVM/PVM.

In summary, this paper makes the following contributions:

- A comprehensive design space exploration for secure containers, revealing a mismatch between the CPU privilege levels and the need of container kernel separation;
- A new secure container design called CKI that achieves efficient kernel separation by introducing a hardware-software co-designed privilege level;
- A system prototype with experimental results on real-world applications demonstrating its effectiveness.

2 Background and Motivation

2.1 Secure Container Models

OS-level containers [26, 63] achieve lightweight isolation by sharing a single OS kernel among multiple containers. However, the OS-level isolation mechanisms [1, 2, 20, 27, 46] are weak, given that many vulnerabilities have been discovered in commodity OS kernels like Linux. The complex syscall interfaces expose a large attack surface to the userspace, which can be exploited by containers for privilege escalation, information leakage, or denial-of-service (DoS) [4, 6, 7, 16, 53]. Secure containers enhance container isolation by limiting the effects of a compromised OS kernel. This can be achieved by two container architectures: VM-level containers and enclave-based containers, as depicted in Figure 1.

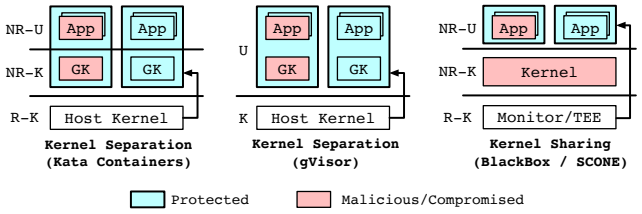

Figure 1. Secure containers: VM-level (kernel separation) and enclave-based (kernel sharing). GK: guest kernel, NR: non-root, R: root, U: ring-3, K: ring-0.

VM-level containers. VM-level containers run each container on a separate guest kernel. They require three levels of privilege for user applications, the guest kernel, and the host kernel. A malicious application within the container can compromise its guest kernel by exploiting kernel vulnerabilities, which, however, is harmless to the host kernel or other containers. The interface between the guest kernel

and the host kernel can be much simpler than the syscall interface [30, 66], making it difficult for a malicious guest kernel to compromise the host kernel.

Enclave-based containers. Enclave-based containers run all the containers on a shared kernel, just like OS-level containers. Yet, a security monitor [43] or a hardware-based trusted execution environment (TEE) [31, 44] deprives the shared kernel of the privilege to arbitrarily access the memory data or execution context of the protected containers.

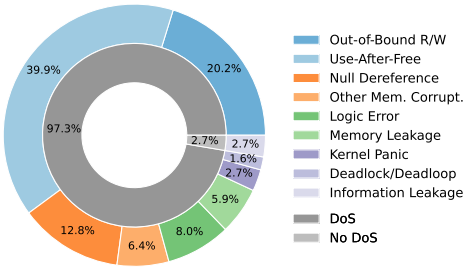

Figure 2. Linux kernel CVEs exploitable by containers.

VM-level containers are preferred when considering real-world CVEs. We collect Linux kernel CVEs that are exploitable in containers in the recent two years (2022-2023) and classify them (209 in total) by security effects, as shown in Figure 2. Among these CVEs, 97.3% can lead to denial-of-service (DoS) attacks, including breaking system states (e.g., out-of-bound write, use-after-free), causing irrecoverable errors (e.g., null pointer deference, kernel panic), or monopolizing hardware resources (e.g., memory leak, deadlock).

Although enclave-based containers can protect the confidentiality and integrity of container data, they cannot defend against DoS attacks due to the kernel-sharing design. In contrast, VM-level containers can prevent DoS because of the kernel-separation design.

2.2 Secure Containers in Nested Clouds

According to prior work like [45], there is a growing demand to build container platforms within VMs leased from an infrastructure-as-a-service (IaaS) cloud. In these nested clouds, VM-level containers are deployed inside another VM. For instance, Alibaba Cloud is shifting secure containers from bare-metal instances to general-purpose (virtualized) instances to achieve greater isolation, cost savings, and more flexible and elastic Kubernetes cluster management [45]. Additionally, Google's gVisor considers in-VM deployment and designs corresponding optimizations [9].

Running secure containers within an IaaS VM requires nested virtualization, in which an L0 kernel (host hypervisor) runs a VM with an L1 kernel (guest hypervisor), which then isolates multiple secure containers (L2 VM), each with its own L2 kernel. Both memory-intensive and I/O-intensive container applications may be deployed under nested virtualization [45, 52, 69]. We aim to mitigate the overhead of nested virtualization for these applications.

2.3 Memory Protection Keys

Memory Protection Keys (MPK) is a recent hardware feature for memory isolation on x86 CPUs [14]. MPK divides the pages in a virtual address space into at most 16 domains and leverages four previously unused bits in the page table entry (PTE) to represent the domain ID of each page. It also introduces a 32-bit per-core protection key register to configure the access permissions for each domain. The permission can be set to read-only, read-write, or non-accessible.

MPK has two variants: Protection Keys User (PKU) controls the permissions for user pages, while Protection Keys Supervisor (PKS) controls the permissions for kernel pages. The protection key registers for PKU and PKS are referred to as PKRU and PKRS, respectively. PKRU can be configured with an efficient instruction named *wrpkru*, while PKRS can be configured with *wrmsr* instruction.

2.4 Issues of VM-Level Containers

VM-level containers require three privilege levels, while the CPU hardware only offers two privilege levels for the kernel and applications. One problem with VM-level containers is *the lack of a third CPU privilege level to efficiently accommodate the guest (container) kernel*. Specifically, Figure 3 illustrates existing VM-level container designs, and Table 1 shows a comparison of them. The problems with the existing designs are summarized as follows.

(1) Hardware-assisted virtualization (HVM) isolates the guest kernel with dedicated VM control structures (VMCS) and EPT, resulting in suboptimal memory performance due to EPT translation and management, as well as poor I/O performance in nested clouds caused by slow VM exits.

(2) Software-based virtualization (PVM) deprivileges the guest kernel to user mode and isolates container memory with shadow paging, resulting in additional context switches during syscall handling and inefficient page table updates.

(3) LibOS-based containers break the isolation between applications and the guest kernel, reducing security guarantees and causing compatibility issues.

2.4.1 HVM.

The container design with HVM faces both performance and compatibility issues.

Overhead of EPT. Handling one EPT fault takes 3µs for HVM in a bare-metal cloud, which could increase the latency of memory-intensive applications by 2%~21% compared with OS-level containers (HVM-BM / RunC-BM in Figure 4). HVM also results in expensive TLB miss handling due to the two-dimensional page table walk, which incurs an average overhead of 46% for memory-intensive applications [39, 69].

In nested clouds, since there is no hardware support for three-stage address translation, the L1 kernel relies on the

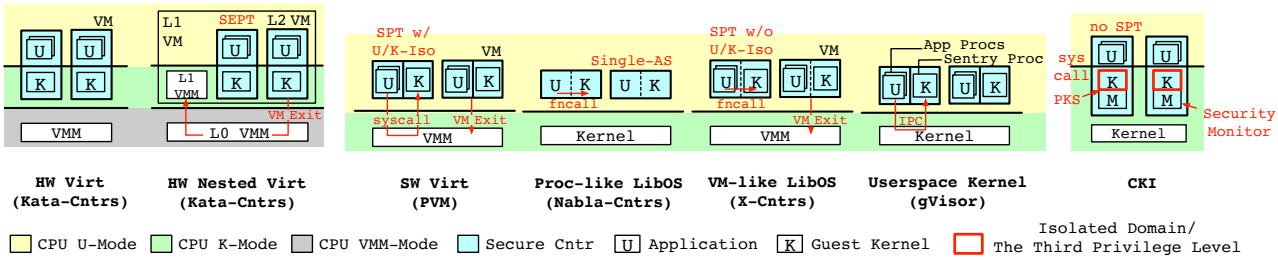

Figure 3. Comparison of different VM-level containers. Cntr: container, S(E)PT: shadow (extended) page table, AS: address space, fncall: function call, U/K-Iso: user/kernel isolation, VMM: hypervisor.

Table 1. Comparison of different VM-level containers. BM: bare-metal, NST: nested.

Aspect		HW-Assisted VM	SW-Based VM	Proc-Like LibOS	VM-Like LibOS	Userspace Kernel	CKI
Performance	Memory Intensive Applications	◐ (BM) ◔ (NST)	◐	●	◐	●	●
	I/O Intensive Applications	● (BM) ◔ (NST)	◐	●	●	◐	●
Security	Guest User-Kernel Isolation	●	●	○	○	●	●
Compatibility	Nested Cloud Deployment	◐	●	●	●	●	●
	Container Binary Compatibility	●	●	◐	●	◔	●

L0 kernel to maintain a shadow EPT (SPTE) for each L2 VM, resulting in high overhead for EPT management [45]. A page fault in an HVM container takes more than 32µs in a nested cloud. This increases the latency of memory-intensive applications by 28%~226% compared with OS-level containers (HVM-NST / RunC-BM in Figure 4).

Overhead of VM exit redirection. When running HVM containers in a nested cloud, the L1 kernel and the L2 VM execute with different VMCS, resulting in L0 intervention on VM exits. Specifically, when a VM exit occurs in an L2 VM, it triggers a trap to L0 kernel, and then L0 kernel resumes the L1 kernel to handle the L2 VM exit. After the L1 kernel processes the VM exit, it traps to L0 kernel again, and then L0 kernel resumes the L2 VM. An empty hypercall in an HVM container only takes 1.1us in a bare-metal cloud, while it takes 6.7us in a nested cloud. This VM exit overhead reduces the throughput of I/O-intensive applications by 1.8×~4.3× compared with PVM [45], which eliminates L0 intervention (HVM-NST / PVM-NST in Figure 5).

Compatibility issues. HVM also faces compatibility issues in nested clouds. First, some IaaS clouds disable hardware-assisted nested virtualization to reduce the attack surface of the L0 kernel [45]. Second, hardware-assisted nested virtualization is not supported for emerging confidential virtual machines because the L0 kernel is untrusted.

2.4.2 PVM.

PVM [45] implements VM-level containers with para-virtualization techniques. It runs container applications and the host kernel in user and kernel mode, respectively. It also runs the guest (container) kernel in user mode within an individual address space. PVM obtains better performance than hardware-assisted virtualization in nested clouds by avoiding VM exits to L0 kernel.

Table 2. Container performance on microbenchmarks (ns).

	Bare-metal (BM)			Nested (NST)	
	RunC	HVM	PVM	HVM	PVM
syscall	93	91	336	91	336
pgfault	1,000	4,347	6,727	34,050	7,346
hypercall	-	1,088	466	6,746	486

Overhead of syscall redirection. When an application invokes a syscall, it first traps to the host kernel. The host kernel then switches to the guest kernel page table, returns to user mode, and invokes the syscall handler in the guest kernel. The guest kernel returns to the user application with a reverse procedure after it handles the syscall. This syscall procedure adds two more CPU mode switches and two more page table switches compared with a native syscall. This increases the syscall latency from 90ns to 336ns, and incurs an average overhead of 6.6% on I/O intensive applications compared with HVM in a bare metal cloud. The application overhead is evaluated by emulating PVM syscall overhead in an HVM container, using the applications in Figure 5.

Overhead of shadow paging. PVM preserves the abstraction of two-stage address translation: guest virtual address (gVA) to guest physical address (gPA) to host physical address (hPA), by using the shadow paging mechanism. The host kernel maintains a shadow page table (translates gVA to hPA) for each container application. One container page fault involves at least 6 context switches between the host kernel and the guest. Two switches are needed to redirect the page fault to the guest kernel, two for updating the shadow page table,

and two more for returning to the user application. Moreover, the emulation logic during the page fault handling also incurs high overhead, such as page table walking, instruction emulation, shadow page table management, and exception injection. A page fault in a PVM container takes 7µs while a native page fault only takes 1µs. The overhead of shadowing paging increasing the latency of memory-intensive applications by 6%~73% compared with OS-level containers (PVM-BM / RunC-BM in Figure 5).

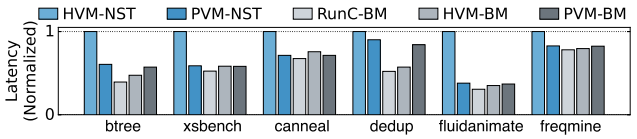

Figure 4. Comparing the performance of different containers on memory-intensive (page-fault-intensive) applications.

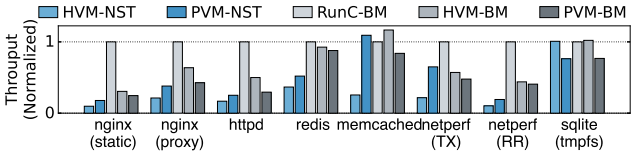

Figure 5. Comparing the performance of different containers on I/O-intensive applications.

2.4.3 LibOS and gVisor. LibOS-based secure containers [38, 48, 49, 62, 66] attach a library OS (libOS) to each container within either a process or a VM. They do not enforce user-kernel isolation within the secure container, running both the applications and the libOS in the same address space. This design avoids the page table switches during syscall handling, but it weakens the isolation guarantee of containers. Meanwhile, they usually have less compatibility, such as a lack of full support for multi-processing in a container.

gVisor [3] implements a new userspace kernel called Sentry, and runs each container on a private Sentry instance. gVisor lets the host kernel to handle the application page faults, avoiding the overhead of shadow paging. gVisor relies on Systrap [24] to redirect the application syscalls to the Sentry instance, based on binary rewriting. However, Systrap is much slower than native syscalls, because it involves interprocess communication. Meanwhile, as a re-implemented kernel, gVisor may lack the full compatibility and optimizations of Linux kernel.

3 Overview

3.1 Design Implications and Design Choices

Design implications. According to the analysis in §2.4, there are two design implications for efficiently building a new privilege level for container guest kernels.

(1) *Efficient switches between privilege levels.* As the guest kernel frequently communicates with applications (i.e., syscall/exception) and performs privileged operations (e.g., page table updates and I/O requests), we should minimize the context switch overhead during these procedures. Overheads like syscall redirection in PVM or VM exit redirection in nested HVM should be avoided.

(2) *Memory isolation without two-stage translation.* Two-stage address translation is designed for general-purpose virtualization, which exceeds the demand of container isolation, as containers do not rely on specific physical memory layout. Using single-stage translation can avoid the overhead of shadow paging or EPT translation/management.

Design choices. MPK can enforce efficient domain isolation within a single CPU privilege level [40, 41, 55], which can be leveraged for constructing the new privilege level. There are two possible designs: (1) Run the guest kernel in user mode and isolate it from the application using PKU (*Design-PKU*), (2) Run the guest kernel in kernel mode and isolate it from the host kernel using PKS (*Design-PKS*). Both designs support efficient syscalls without redirection. They also avoid VM exit redirection in nested clouds as the guest kernel and the host kernel execute with the same VMCS.

We choose *Design-PKS* instead of *Design-PKU* for the following reasons. First, because PKU has been widely adopted for various applications (intra-process isolation [35, 42, 64]), *Design-PKU* would inherently interfere with these existing use cases. This conflict undermines the original purpose of PKU in user-space applications. Second, *Design-PKU* necessitates replacing both *wrpkru* instructions and *syscall* instructions in container applications. However, as highlighted in Hodor [42], binary rewriting on arbitrary (container) images without source code can be undecidable, which may break container binary compatibility or hinder application features like just-in-time compiling. Third, *Design-PKU* introduces extra performance overhead in exception handling. For instance, injecting page faults from the host to the guest kernel requires additional cross-ring switches, adding approximately 750ns to page fault latency (which is originally around 1,000ns) on our tested platform.

3.2 Challenges of PKS-based Isolation

PKS is not originally designed for isolating container guest kernels, leading to the following challenges for our design.

Challenge-1: Insufficient number of isolation domains. PKS only supports up to 16 domains in one address space, which is much lower than the potential number of secure containers. Therefore, it is infeasible to isolate each guest kernel in a dedicated PKS domain.

Challenge-2: Lack of privileged instruction isolation. PKS only offers memory isolation, while a malicious guest kernel running in kernel mode can break isolation using privileged instructions. Binary rewriting is a common technique for

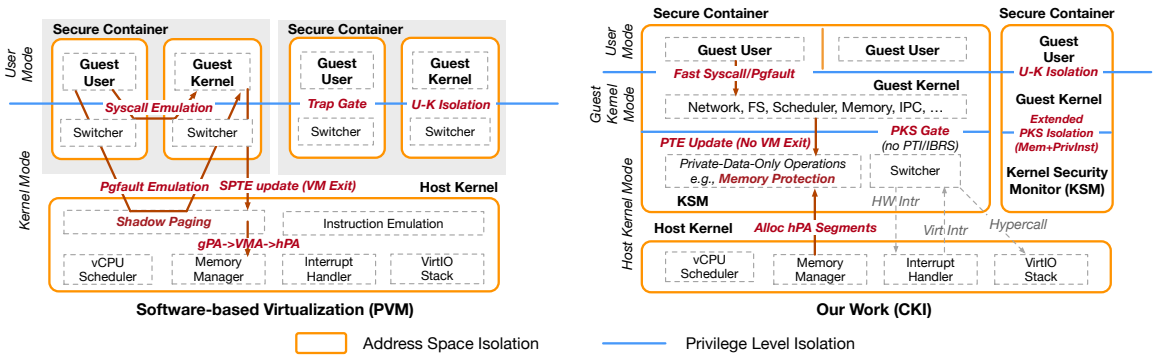

Figure 6. Comparing the architecture of software-based virtualization and CKI. PrivData-Only Operations: privilege operations that only access the private data of the current secure container. Switcher: a piece of context switching code. Note: these two designs work in both bare-metal and nested clouds (the host kernel is L1 kernel in nested clouds).

removing specific instructions from isolated software. However, it is infeasible to eliminate all of the privileged instructions at unaligned locations in an OS kernel [37, 40, 42]. For nested clouds, one potential solution is to intercept and inspect all privileged instructions in L1 VM using virtualization hardware [40], but it requires intrusive modifications to the L0 kernel, which may not be feasible.

Challenge-3: Incomplete switch gate capabilities. Under PKS-based isolation, the guest kernel communicates with the host kernel using PKS switch gates. However, such switch gates require capabilities that are not naturally supported by PKS. For example, the host kernel needs to intercepts hardware interrupts during guest execution using the switch gates, but a naive gate design might allow the guest kernel to inject fake interrupts into the host kernel.

3.3 Architecture Overview

CKI is a VM-level container architecture, where each container runs on a separate kernel. It implements three privilege levels without hardware virtualization extensions. PVM [45] is the state-of-the-art secure container design without using virtualization hardware. Figure 6 shows the architecture of CKI and how it differs from PVM. In short, CKI avoids the overhead of syscall redirection and shadow paging of software-based virtualization, and thus can achieve better performance no matter in bare-metal or nested cloud.

Abstraction. Each CKI secure container is like a VM with a guest kernel and multiple user processes, running on a host kernel. The guest kernel provides OS functionalities like memory management, scheduling, filesystem and network stack. The host kernel schedules the vCPUs of the guests, allocates memory to them, and emulates virtual devices (disks and network cards) for the guest kernel using the VirtIO protocol. All hardware interrupts are handled by the host kernel. When a guest kernel needs to invoke host functionalities, or when a hardware interrupt occurs, the guest vCPU exits to the host kernel via a piece of context-switching code called switcher (§4.2). For nested virtualization, CKI VM exit process does not involve L0 intervention. The host kernel may inject virtual interrupts when resuming the guest vCPU.

Differences. There are two key differences between CKI and software-based virtualization. First, CKI runs the guest kernel in a new privilege level within the kernel mode, allowing guest users to invoke syscalls without host kernel intervention. The memory of the guest kernel is mapped in the guest user address spaces and is isolated with the PTE U/K-bit, which eliminates page table switches during syscalls. Second, CKI does not implement two-stage address translation. The host kernel provides each guest VM with some contiguous segments of hPA that are directly managed by the memory manager in the guest kernel. Therefore, guest user page faults can be handled directly by the guest kernel, instead of triggering the shadow page fault in the host kernel. The PTE update operation is also simplified by eliminating VM exits, gPA to hPA translation, and shadow PTE generation.

Defining the new privilege level (guest kernel mode). CKI leverages PKS-based intra-kernel isolation to construct the new privilege level for de-privileging guest kernels. On the one hand, it combines PKS isolation with address space isolation to restrict the memory accessibility of guest kernels. On the other hand, it monitors the execution of privileged operations from guest kernels.

Specifically, different secure containers and the host kernel are isolated in different address spaces. Each guest kernel is deprivileged and operates with a privileged kernel security monitor (KSM), both of them running in the same address space but have different PKS permissions specified in PKRS (protection key rights register for kernel pages). The KSM (PKRS is zero) can access all the virtual memory, while the guest kernel (PKRS is *PKRS_GUEST*) cannot access the memory of its KSM. Within the address space of each secure

container, only two PKS domains are needed for the guest kernel and the KSM. Thereby, CKI can support an arbitrary number of secure containers without being affected by the PKS domain limitation (overcoming *Challenge-1*).

Additionally, CKI adds a lightweight hardware extension to PKS that makes privileged instructions inexecutable in guest kernels (§4.1) (overcoming *Challenge-2*). A guest kernel can only perform privileged operations through the predefined interfaces offered by its KSM or the host kernel. The KSM implements the privileged operations that only access the private data of the secure container, such as page table updates (§4.3), and *iret* instruction. These operations can be invoked through an efficient PKS gate (switch between guest kernel and KSM) (§4.2). CKI eliminates the costly side-channel mitigation (e.g., PTI [21] and IBRS [13]) from this PKS gate, since only private data is mapped in the KSM [33].

Other privileged operations (e.g., VirtIO MMIO, timer setup, *hlt* instruction), which rely on global data (e.g., driver/scheduler metadata), are offered by the host kernel. The guest kernel can invoke such operations through a dedicated switcher (switch between the guest kernel and the host kernel). The switcher also contains interrupt gates that intercepts hardware interrupts during guest execution and redirect them to the host kernel. CKI relies on multiple techniques to prevent interrupt monopolizing and interrupt forgery by a compromised guest kernel (§4.4) (overcoming *Challenge-3*).

3.4 Threat Model

CKI inherits the threat model of VM-level containers. The host kernel and the KSMs isolate multiple containers, while the guest kernel in a container isolates multiple user processes. One container may be compromised and then attempt to break inter-container isolation, e.g., by executing destructive privileged instructions or corrupting critical in-memory structures (like page table, IDT). The KSMs and the host kernel are assumed to be trusted as their attack surfaces are small (hypervisor interface).

Transient execution attacks [47, 54] within a single secure container are out of scope. Inter-container transient execution attacks are mitigated by running each container in its own address space and enabling Spectre mitigation [12, 13, 25] in the host kernel.

4 Detailed Design

4.1 PKS-based Privileged Instruction Isolation

CKI chooses to construct the new privilege level inside the kernel mode due to performance considerations, i.e., the container user processes can interact with the (container) guest kernel in the original efficient manner. For example, a process can still invoke system calls directly through *syscall* instruction, without extra context switches.

As the guest kernel is untrusted, CKI needs to prevent it from executing privileged instructions that can breach security isolation. The current PKS hardware feature can only offer memory isolation within the kernel mode, which cannot restrict the execution of privileged instructions. Moreover, existing software-based instruction isolation techniques like binary rewriting do not apply to CKI (§3.2).

Hardware extension. Thus, CKI introduces a lightweight hardware extension to prevent the guest kernel from executing privileged instructions that could lead to destructive consequences. Since the PKRS is non-zero (limited memory view) during guest kernel execution and zero (unlimited memory view) during KSM execution, the extension can rely on the PKRS register value to determine which one is currently executing. The extension blocks all destructive privileged instructions when the PKRS is non-zero. Executing these instructions in the guest kernel triggers an exception that traps to the host kernel. Non-destructive privileged instructions remain executable in the guest kernel to minimize overhead. Table 3 lists the privileged instructions and whether they are blocked in the guest kernel.

Blocked instructions. Most privileged instructions are blocked, except for the harmless ones listed in Table 3. The blocked instructions can be virtualized using the similar techniques in software-based virtualization, i.e., replacing them with calls to the host kernel or the KSM.

We block any instructions that write system registers (such as GDTR and IDTR), control registers, or model-specific registers (MSR). The interrupt returning instruction (*iret*) is blocked as it can modify segment registers. We also block instructions that are unnecessary for the container guest kernel, such as instructions related to port I/O and system management mode.

OS kernels use *cli*/*sti* and *popf* instructions to enable or disable interrupt handling on the CPU. These instructions are blocked in the guest kernel to prevent DoS. CKI adopts interrupt handling mechanisms from para-virtualization [32, 45]. All hardware interrupts are handled by the host kernel, which injects virtual interrupts into the guest kernel. Instead of using privileged instructions, the guest kernel maintains the interrupt enabled/disabled state with an in-memory bit that is visible to the host kernel.

Not-blocked instructions. The PKRS modification instruction should be executable in the guest kernel; otherwise, the guest kernel cannot invoke the KSM. Existing x86 hardware implements PKRS as a model-specific register (MSR). However, *wrmsr* should be blocked in the guest kernel to prevent arbitrary manipulation of other MSRs. We introduce a new hardware instruction, *wrpkrs*, for PKRS modification, which has semantics similar to the existing *wrpkru* instruction (modifying PKRU, a userspace equivalent of PKRS).

Categories	Related Instructions or Registers	Blocked?	Usages in Container Guest Kernels or Brief Explanations
System Registers	IDTR, GDTR, TR ...	Yes	They are required at boot time only and replaced with KSM calls.
MSRs	RDMSR/WRMSR	Yes	They are used for updating timer and sending IPI, which are replaced with hypercalls.
Control Registers	- MOV CRn, reg	- No	- It is used for reading CR0 and CR4, which is harmless.
	- MOV reg, CR0/CR4	- Yes	- Replaced with KSM call: initializing CR0/CR4, toggling CR0 TS-bit for lazy FPU switching.
	- MOV reg, CR3	- Yes	- Replaced with KSM call: updating CR3 for address space switching.
	- CLAC/STAC	- No	- They are used to toggle the AC-bit (SMAP-enabled) in CR4, which is harmless.
TLB States	- INVLPG	- No	INVLPG is used to flush the TLB. Each secure container and the host are isolated in different
	- INVPCID	- Yes	PCID contexts to prevent one container from flushing the others' TLB entries with INVLPG.
Syscall/ Exception	- SWAPGS, SYSRET	- No	- They are handled with special methods for better syscall performance.
	- IRET	- Yes	- It is used for returning from exceptions and replaced with a KSM call.
Other Privilege Instructions	- HLT	- No	- It is replaced with a hypercall that pause the current vCPU.
	- STI/CLI, POPF	- Yes	- The interrupt enabling/disabling state in the guest is maintained in memory.
	- IN/OUT, SMSW ...	- Yes	- They are not used in a para-virtualized container guest kernel.
PKRS Register	WRPKRS	No	It is used in the PKS switch gates and protected with binary rewriting.

Table 3. Deprive the container guest kernel of the ability to execute destructive privileged instructions.

The *wrpkrs* instruction should only be present at predefined switch gates, so that we eliminate all *wrpkrs* instructions, including the unaligned ones, from the guest kernel code with the similar binary rewriting technique introduced in prior work [64]. To prevent the guest kernel from dynamically creating *wrpkrs* instructions, all kernel code is mapped as read-only during guest kernel initialization, and the KSM prohibits new kernel-executable mapping during container execution. CKI does not need to support dynamic patching or loading of guest kernel code, as this is unnecessary for containers. Note that CKI aims to provide a container environment rather than support arbitrary guest kernels.

The *sysret* and *swapgs* instructions are utilized during syscall handling. Calling the KSM for these instructions would increase the empty syscall latency from 90ns to 153ns. To enhance performance, we allow these instructions to be executable in the guest kernel. The *sysret* instruction could be exploited to modify the RFLAGS register and disable interrupts (DoS). Therefore, we add a lightweight extension to this instruction to ensure that the IF (interrupt enabled) flag remains turned on when the PKRS is non-zero.

The guest kernel can flush the TLB with *invlpg*. We isolate each secure container and the host in different PCID contexts to prevent performance attacks, as *invlpg* only flushes the TLB entries of the current PCID.

4.2 Switch Gates for Context-Switching

Figure 7 shows the context-switching flows in CKI. CKI offers fast paths for the most frequent switches, i.e., syscall, exception, and KSM invocation. It provides slow paths for other switches, i.e., host kernel invocation (hypercall) and hardware interrupts.

When designing the PKS switch gates for these context switches, a challenge is the locating of the per-vCPU area in the KSM. Since the guest kernel can arbitrarily modify *kernel_gs*, the KSM cannot rely on this register to identify the current vCPU.

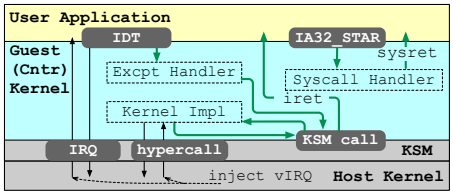

Figure 7. The context switches in CKI. Green lines: fast paths. (v)IRQ: (virtual) interrupt. Cntr: container.

Syscall and exception. When the user application in a container invokes a syscall, it traps to the guest kernel's entry point defined in the *IA32_STAR* register. Similarly, when the application triggers an exception such as a page fault, it traps to the guest kernel's entry point in the interrupt descriptor table (IDT). PKRS is set to *PKRS_GUEST* in user mode, allowing the entry point to call the untrusted handler function without a PKS switch. The entry and exit code for syscalls and exceptions use three privileged instructions: *swapgs*, *sysret*, and *iret*. The *swapgs* and *sysret* instructions are executable in the guest kernel, while the *iret* instruction must be executed by calling KSM (§4.1).

KSM call. The guest kernel uses the KSM call gate to invoke the privileged operations offered by KSM (Figure 8a). The gate sets PKRS to zero, switches to a secure stack that is inaccessible to the guest kernel, calls a handler function, and finally restores PKRS and the stack pointer.

An attacker may jump to the *wrpkrs* instruction at the end of the gate with ROP-like attacks, to arbitrarily modify PKRS and execute malicious code. To prevent this attack, the new PKRS value is checked after modification, as shown by the *switch_pks* macro in Figure 8a.

Per-vCPU area. Since the KSM can be invoked on multiple vCPUs simultaneously, each vCPU has its own secure stack, located in a per-vCPU area of the KSM memory. OS kernels typically use *kernel_gs* register to locate per-CPU variables,

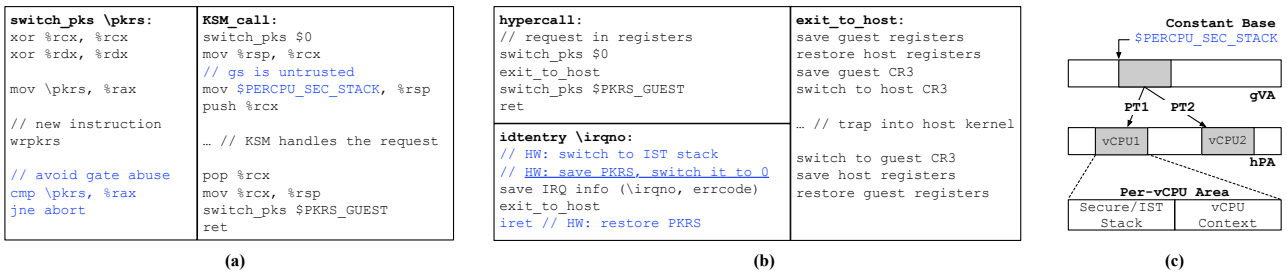

Figure 8. (a) Code snippets of the KSM call gate, (b) code snippets of the hypercall gate and the interrupt gate, (c) the switch gates can find the per-vCPU area at a constant virtual address, without a trusted gs register, PT: page table.

i.e., the *kernel_gs* register on each CPU stores a different base for local CPU variables. However, CKI allows the guest kernel to execute the *swapgs* instruction (§4.1), and thus a malicious guest kernel can arbitrarily modify *kernel_gs*. To tackle with this problem, CKI puts the per-vCPU area at a constant virtual address, which can be found without *kernel_gs*. As depicted in Figure 8c, CKI maintains multiple per-vCPU page tables for each page table in the guest kernel. When a guest thread executes on different vCPUs, different per-vCPU page tables are used. Each per-vCPU page table maps a different per-vCPU area (hPA) at the same constant virtual address (gVA).

Hypercall. The guest kernel uses the hypercall gate to invoke the privileged operations offered by the host kernel. The gate first switches PKRS to zero because it needs to execute privileged instructions and access the KSM memory (per-vCPU area). It then performs a full context switch to save the guest kernel context and restore the host kernel context, which includes a page table switch, general-purpose/system registers switches and side-channel mitigation like IBRS. The host and guest contexts are stored in the per-vCPU area. The host kernel then reads the request from the guest context and processes it. After the request is completed, the host kernel restores the guest context, and the guest kernel resumes from the hypercall gate.

Hardware interrupt. A hardware interrupt triggers a trap from the guest to the host kernel. The IDT entry for a hardware interrupt points to an interrupt gate.

The interrupt gate saves the interrupt information to the per-vCPU area and switches to the host kernel. The host kernel reads the information, constructs an interrupt context, and calls the interrupt handler. After handling the interrupt, the host kernel restores the interrupted guest context.

We add a hardware extension to save the PKRS register upon interrupt entry and switch PKRS to zero (detailed in §4.4). After handling the interrupt, the *iret* instruction should be executed with PKRS set to zero (§4.1), but it needs to restore PKRS to *PKRS_GUEST* when restoring the guest kernel context. Therefore, we extend the *iret* instruction, allowing it to modify the PKRS register.

4.3 Memory Protection

A malicious guest kernel may attempt to breach memory isolation by manipulating the page table. To isolate the memory view of the guest, the KSM intercepts and verifies all page table updates in the guest kernel.

Page table monitoring. To intercept page table updates, CKI employs a mechanism similar to nested kernel [36], which is based on the following invariants: (1) Only declared pages can be used as page table pages (PTP); (2) Declared PTPs are read-only in the guest kernel; (3) Only a declared top-level PTP can be loaded into the CR3 register. Unlike nested kernel, CKI uses PKS instead of the PTE writable bit to control PTP write permissions. CKI divides all the PTPs in a guest virtual address space into a specific PKS domain. It adds the PKS domain ID, *pkey_PTP*, to each guest PTE that maps a PTP. When the guest kernel is executing, this PKS domain is configured as read-only in the PKRS register.

The KSM maintains a descriptor for each physical page belonging to the guest. The guest kernel can invoke the KSM to declare a PTP or update a PTE. When declaring a PTP, the PTP level is specified and recorded in the descriptor. The KSM then walks the page table to find the PTE that maps this PTP and adds *pkey_PTP* to it. The KSM also checks a reference counter in the descriptor to ensure that the PTP is only mapped once. When updating a PTE, the KSM verifies that the new PTE points to a valid next-level PTP or a data page belonging to the guest, and it does not map a declared PTPs. Furthermore, to prevent malicious *wrpkrs* instructions (§4.1), KSM prohibits the update if the new mapping is kernel-executable.

Per-vCPU page table. As mentioned in §4.2, CKI maintains multiple per-vCPU page tables for each guest page table. Each per-vCPU page table maps a different per-vCPU area in the KSM memory. Specifically, KSM maintains multiple per-vCPU copies for each top-level PTP in the guest. When a top-level PTP is declared, KSM adds the mapping of its own code and data, including the per-vCPU area, to each of these copies. When the guest kernel invokes the KSM to update CR3, the KSM verifies that the new CR3 value

points to a declared top-level PTP, and then loads the corresponding PTP copy into CR3. Moreover, the KSM provides an interface for reading PTEs in the top-level PTP, where the accessed/dirty-bit is propagated from the copies to the original PTP.

Comparison with shadow paging. Compared with shadow paging, the performance benefits of CKI arise from more lightweight page fault and PTE updating procedures.

(1) *Lightweight page faults.* Without two-stage address translation, a user page fault in CKI can be directly handled by the guest kernel. In contrast, under shadow paging, a user page fault is intercepted by the host kernel, which performs a page table walk to determine the type of page fault (first-stage or second-stage) and then injects the page fault into the guest kernel.

(2) *Lightweight PTE updates.* First, under shadow paging, a PTE update in the guest triggers a VM exit to the host kernel. In contrast, the guest kernel in CKI can call KSM for PTE updating through a lightweight PKS gate. Second, shadow paging associates gPAs with the virtual memory areas (VMAs) of the QEMU process. When writing a gPA to a PTE, it must find the hPA associated with the gPA from the mapping of the VMA, which is time-consuming. In contrast, CKI delegates hPAs to guest kernels, allowing the guest kernel to directly fill the hPA, instead of a gPA, in the PTE.

A limitation of CKI is that it allocates contiguous physical memory segments to each secure container, which may result in low memory utilization due to memory fragmentation.

4.4 Interrupt Abuse Prevention

A compromised guest kernel has three potential methods for initiating DoS attacks by abusing interrupts. First, it may alter the code of the interrupt gate to monopolize all the interrupts. If so, the host and other containers can no longer receive interrupts. Second, it might manipulate the interrupt stack to trigger an unrecoverable fault. Specifically, when an interrupt occurs, the CPU pushes context data onto the interrupt stack. If the interrupt occurs in kernel mode, the CPU by default uses the stack at the occurrence of the interrupt as the interrupt stack. A malicious guest kernel could set the stack pointer as an invalid address, leading to a triple fault when the CPU tries to push data. Third, it can forge interrupts and overwhelm the system with unnecessary interrupt requests, thereby degrading system performance or causing undefined behavior in the host kernel. CKI defends all of these attacks.

Prevent interrupt monopolizing. First, CKI allocates the IDT and the interrupt gate code in the KSM memory, making them non-modifiable by the guest kernel. Second, it uses the privilege deprivation mechanism (§4.1) to ensure that a guest kernel cannot disable interrupt handling nor modify IDTR (IDT base address register). Third, the guest kernel cannot change or remove the mapping of the IDT or the interrupt gate code, because the KSM maps its own memory in each activated page table (§4.3). With these mechanisms, CPU control flow always switches to the correct interrupt gate when an interrupt occurs.

Prevent interrupt stack manipulation. CKI leverages x86 interrupt stack table (IST) feature to ensure that CPU always uses the correct interrupt stack. Specifically, IST allows to set a specific interrupt stack and forces the CPU switch to the stack before pushing the interrupt context. The IST initialization is finished by the KSM (the guest kernel cannot execute the related privileged instructions) and the corresponding memory is also located in the KSM (the guest kernel cannot modify the corresponding memory data).

Prevent interrupt forgery. As the interrupt gate needs to access KSM memory and execute privileged instructions, it needs to first switch PKRS to zero when an interrupt happen in the guest kernel. If the switch is made by a *wrpkrs* instruction within the gate, a malicious guest kernel could directly jump to one of the interrupt gates and send a forged interrupt to the host kernel.

To prevent interrupt forgery, CKI extends the IDT configuration to further support switching PKRS register besides its original functionalities like switching interrupt stack. As described by the underlined blue text in Figure 8b, this minor hardware extension automatically sets the PKRS register to zero when a hardware interrupt occurs. Thus, there is no *wrpkrs* instruction in the interrupt gate. If the guest kernel jumps to the gate entry, the PKRS remains *PKRS_GUEST*, causing subsequent context switching to fail. Note that the application or the guest kernel may generate a software interrupt with an *int* instruction. The hardware extension switches PKRS only on hardware interrupts and keeps PKRS unchanged on software interrupts.

Additionally, the guest kernel also cannot abuse the hypercall gate for interrupt forgery because the host kernel can identify different exit reasons based on the information saved in the KSM (per-vCPU memory area).

5 Implementation

Guest kernel. We run Linux kernel as the guest kernel in the CKI container. We leverage the para-virtualization utilities in the Linux kernel (i.e., *pv_ops*) to hook privileged operations. We also add a new boot procedure in the Linux kernel to remove legacy initialization operations. We add ~2K lines of code (LoC) and modify fewer than 80 LoC.

Removing two-stage address translation does not require significant porting effort. A legacy kernel may depend on fixed low physical addresses for real mode startup. In contrast, CKI starts virtual CPUs (vCPUs) directly from long mode via para-virtualization. The traditional virtualization stack uses two-stage address translation to create MMIO regions that are mapped in the first stage but not in the second stage. We replace the MMIOs in the guest kernel (VirtIO frontend) with hypercalls.

In terms of compatibility, CKI can potentially support the same set of guest kernel features as software-based virtualization.

Hardware extensions. The performance evaluation in §7 is conducted on real hardware rather than in the simulator. In the evaluation, we use the *wrpkru* instruction to emulate the *wrpkrs* instruction. According to our evaluation based on Gem5 simulator [8], PKS permission-checking logic added to the privileged instructions incurs negligible overhead, so we use the unmodified instructions directly in the evaluation. We emulate the PKRS switching overhead during interrupt entry and *iret* by adding *wrpkru* instructions.

6 Security Analysis

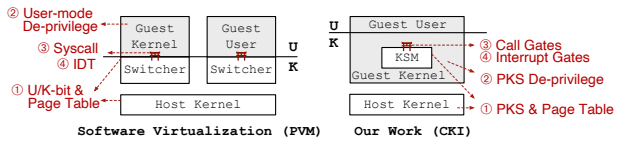

Figure 9. A comparison of isolation primitives in software-based virtualization (PVM) and CKI.

CKI can achieve the same security goals as software-based virtualization (PVM) because it implements the same set of isolation primitives (see Figure 9).

First, PVM isolates the switcher and host kernel memory from the guest kernel with PTE U/K-bit and separate page tables. CKI isolates KSM and host kernel memory from the guest kernel with PKS and separate page tables (§4.3). Second, PVM prevents the execution of privileged instructions by the VM guest kernel by running it in user mode. CKI restricts privileged instructions from the guest kernel with an extension on PKS (§4.1). Third, PVM provides the VMs with a pre-defined syscall entry point to invoke the host kernel. CKI employs binary rewriting to eliminate *wrpkrs* instructions from the guest kernel, leaving only valid entry points to call KSM or the host kernel (§4.2). Fourth, in PVM, when a VM is interrupted by hardware interrupts, the CPU invokes the corresponding host kernel handler function defined in the IDT. CKI designs interrupt gates that redirects hardware interrupts to the host kernel, ensuring the gates cannot be broken or abused (§4.4).

7 Performance Evaluation

We evaluate the performance of CKI with microbenchmarks and application benchmarks, in both bare-metal (BM) and nested (NST) clouds. For experiments that yield similar results in both scenarios, we omit the BM/NST label in the results to save space.

Baseline. We compare CKI with OS-level containers (RunC [26]) and state-of-the-art VM-level secure containers (Kata Containers [15]) based on hardware-assisted virtualization (HVM) or software-based virtualization (PVM [45]). The guest kernels in CKI containers and Kata Containers are based on Linux 6.7.0-rc6.

Testbed. All evaluations are performed on an AMD server with an EPYC-9654 CPU running at 2.4GHz and 125GB of memory. For nested cloud evaluations, the containers are deployed in a hardware-assisted L1 VM with 16 vCPUs, 16GB of memory, and a VirtIO network interface. Both the L0 machine and L1 VM run Ubuntu 22.04 with Linux 6.7.0-rc6.

7.1 Microbenchmarks

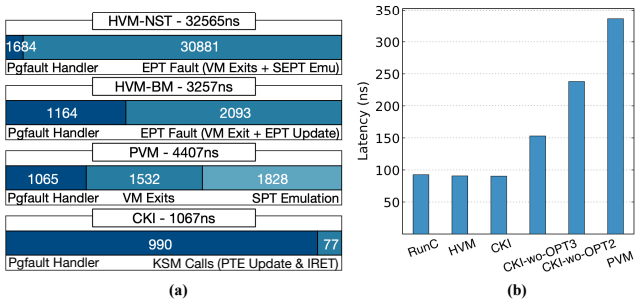

Figure 10. (a) Page fault latency, (b) system call latency.

Page fault. Figure 10a presents a breakdown of page fault latency in different secure containers. The latency is measured by allocating a large virtual memory region within the container and sequentially accessing each 4K page. For HVM, each page fault in the container triggers an additional EPT fault because the newly allocated gPA is not mapped in the EPT. This results in an overhead of 2,093ns (BM) and 30,881ns (NST). The high overhead in a nested cloud is due to the costly switches between the L0 kernel, the L1 kernel, and the L2 VM, as well as shadow EPT emulation. The page fault latency for PVM is four times that of RunC, due to VM exits (1,532ns) and shadow paging emulation (1,828ns). CKI eliminates the overhead of shadow paging by running the guest kernel in kernel mode and removing two-stage address translation. CKI only incurs 77ns overhead on a page fault due to the KSM calls for the PTE update and *iret*.

System call. Figure 10b shows the latency of a simple syscall (getpid) in different containers. For RunC, HVM, and CKI, the syscall latencies are the same (~90ns). The syscall latency in a PVM container is 336ns due to the additional page table and CPU mode switches.

Compared with PVM, CKI optimizes syscall latency through three optimizations: eliminating additional CPU mode switches (OPT1), eliminating page table switches (OPT2), and allowing *sysret* and *swapgs* to be directly executable in the guest kernel (OPT3). We further examine these optimizations by introducing two test settings, CKI-wo-OPT2 and CKI-wo-OPT3. The CKI-wo-OPT2 setting adds

two page table switches to the syscall path of CKI. The CKI-wo-OPT3 setting blocks *sysret* and *swapgs* in the guest kernel, resulting in two PKS switches in the syscall path of CKI (zero to *PKRS_GUEST* on entry, *PKRS_GUEST* to zero on exit). Comparing PVM with CKI-wo-OPT2, OPT1+OPT3 reduces the syscall latency from 336ns to 238ns. Comparing CKI-wo-OPT2 with CKI, OPT2 further reduces the latency to 90ns. Comparing CKI-wo-OPT3 with CKI, adding OPT3 based on OPT1+OPT2 reduces the latency from 153ns to 90ns.

VM exit in nested cloud. In a nested cloud, both CKI and PVM support direct VM exits from the L2 VM to the L1 kernel without the intervention of the L0 kernel. Thereby, an empty hypercall takes only 390ns for CKI and 486ns for PVM. In contrast, for HVM, an empty hypercall takes 6,746ns due to the intervention of the L0 kernel.

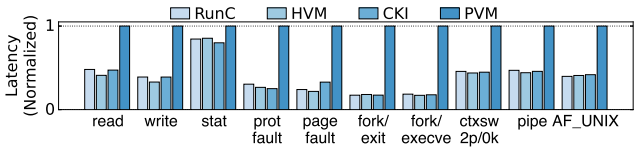

Figure 11. Container performance on lmbench.

lmbench. We evaluate the containers using lmbench microbenchmarks [17], with the results shown in Figure 11. The syscall redirection overhead in PVM containers is significant for syscalls with short execution times, e.g., doubling the latency of a read syscall. PVM shadow paging also imposes a large overhead on memory management operations, such as page fault handling and process creation. Additionally, context switching and inter-process communication are slow on PVM because the kernel need to invoke a hypercall to switch process page tables. CKI adds KSM calls to memory management operations and process switches, but the end-to-end overhead is small as PKS switches are fast. HVM achieves performance comparable to RunC on lmbench, as these test cases do not involve VM exits.

7.2 Memory-intensive Applications

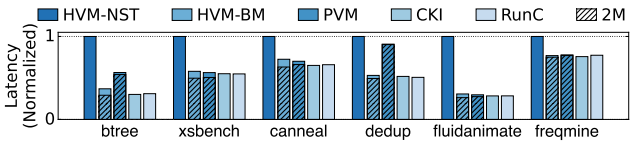

Figure 12. Latencies of memory-intensive (page-fault-intensive) applications, 2M: enabling huge page mapping for (container) VM memory.

Figure 12 shows the latencies of memory-intensive (page-fault-intensive) applications from PARSEC [68] and vmitosis-workloads [59]. Compared with HVM-NST, HVM-BM, and PVM, CKI reduces latencies by 24%~72%, 1%~18% and 2%~47%, respectively. CKI incurs low overhead (<3%) compared with RunC.

We also evaluate PVM and HVM-BM with 2MB huge page mapping enabled for the (container) VM memory. The overhead of HVM-BM becomes minor since each EPT fault handles a huge page, allowing the cost to be amortized. Compared with PVM, CKI still reduces the latencies of btree and dedup applications by 44% and 42%, respectively, as each page fault in the container triggers VM exits. HVM-NST is not evaluated with huge page enabled because it fails to run.

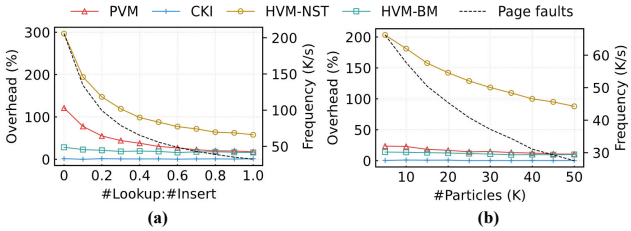

Figure 13. Overhead of secure containers on page-fault-intensive applications, (a) BTree, (b) XSBench.

Figure 13 analyzes the overhead of secure containers (compared with RunC) with the BTree and XSBench applications. The BTree application first inserts a group of entries into a BTree key-value store and then performs a series of search operations on it. The insertion operation is more time-consuming since of triggering new memory allocation and page table modification. Therefore, the overhead decreases as the lookup/insert ratio increases. The XSBench application simulates Monte Carlo neutron transport and consists of an initialization phase for data generation, followed by a calculation phase for simulating each particle. The overhead in this case mainly stems from data generation, resulting in higher overhead when the calculation phase is shorter (fewer particles).

The overhead of CKI remains low across different parameters due to the small per-page-fault overhead it incurs. Additionally, we analyze the page fault frequency of these applications to validate our overhead analysis.

Table 4. Finish time of TLB-miss-intensive applications, HVM-BM results: EPT huge page disabled/enabled.

	RunC-BM	HVM-BM	PVM-BM	CKI-BM
GUPS	54.9	67.8 / 67.1	54.9	55.1
BTree-Lookup	22.6	24.1 / 24.2	21.7	22.6

Table 4 shows the latencies of memory-intensive (TLB-miss-intensive) applications in bare-metal. Compared with HVM-BM, CKI-BM avoids two-dimensional page table walk, reducing the latency of GUPS [10] and BTree lookup (45GB

memory) by 19% and 6%. We also evaluate HVM-BM with 2MB huge page mapping enabled in the EPT, and the results are similar.

7.3 I/O-intensive Applications

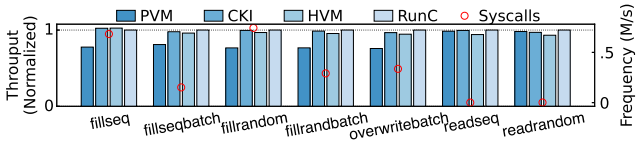

Figure 14. Container performance on SQLite benchmark.

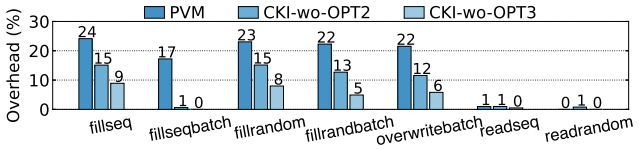

Figure 15. Break down the syscall optimizations in CKI with SQLite benchmark. OPT2/OPT3: see §7.1.

We evaluate the SQLite relational database engine [28] using sqlite-bench [5], which measures operation throughput with various access patterns. The database file is stored in an in-memory filesystem (tmpfs). We ignore the EPT fault overhead of HVM by running the test cases twice. Figure 14 shows the throughput of different containers and the syscall frequency for each test case. The syscall redirection overhead of PVM is correlated with syscall frequency. For database writing, PVM throughput is 19%~24% lower compared with RunC, as the SQLite engine interacts intensively with the filesystem. In contrast, for database reading, the low syscall frequency means that PVM does not incur significant overhead. Both CKI, HVM, and RunC achieve similar throughput since they support native syscalls, and the evaluation does not involve virtualized I/O (VM exits) because it uses tmpfs.

Figure 15 shows an analysis of the syscall optimizations introduced by CKI. Comparing PVM with CKI-wo-OPT2, removing the additional CPU mode switches in PVM reduces overhead by up to 10%. Comparing CKI-wo-OPT2 with CKI, eliminating page table switches further decreases overhead by up to 15%. Comparing CKI-wo-OPT3 with CKI, making *swapgs* and *sysret* executable in the guest kernel reduces overhead by up to 9%.

We evaluate two in-memory key-value stores, memcached [18] and Redis [23], using memtier_benchmark [19] with an 1:1 read/write ratio and a data size of 500 bytes. Figure 16 presents the results for different numbers of clients. Compared with HVM-NST, CKI-NST eliminates L0 intervention on VM exits, achieving 6.8× throughput for memcached and 2.0× throughput for Redis. Compared with

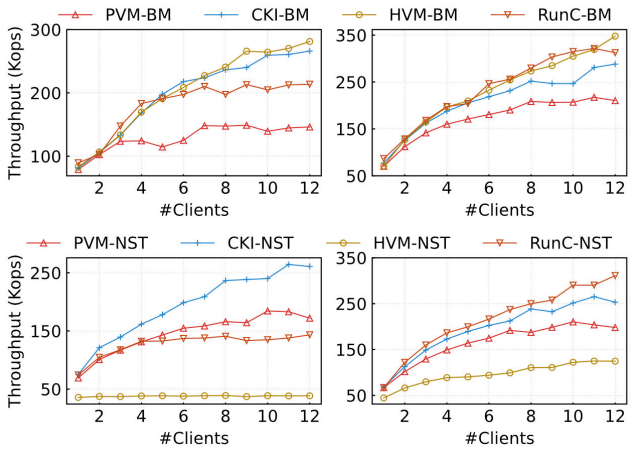

Figure 16. Throughput of key-value stores in containers with different number of clients. (a) memcached, (b) Redis.

PVM-BM/PVM-NST, CKI-BM/CKI-NST achieves 1.8×/1.5× throughput for memcached, and 1.4×/1.3× throughput for Redis. The optimization of CKI over PVM partly results from the elimination of syscall redirection. We emulate PVM syscall latency on CKI, resulting in a throughput decrease of up to 4.4% compared with unmodified CKI. The remaining performance improvements stem from the simpler VirtIO implementation in CKI, such as replacing MMIOs with hypercalls.

8 Other Related Work

Intra-kernel isolation. Table 5 shows prior work on intra-kernel domain isolation. Nested Kernel [36] and NICKLE [60] create a single isolation domain to deprivilege the entire kernel and monitor security-critical operations. UnderBridge [40], LVD [57], SILVER [67], and BULKHEAD [41] establish multiple domains for isolating kernel extensions. Unlike UnderBridge, which relies on MPK, the number of domains in CKI is not limited by the number of keys because it integrates PKS isolation with page table isolation. BULKHEAD proposes a similar technique that isolates different groups of kernel extensions in different address spaces.

Some prior work does not support page table updates within domains (LVD, UnderBridge) or relies on costly shadow paging (NICKLE). In contrast, CKI enables secure and efficient page table management within secure containers. Nested Kernel and BULKHEAD isolate privileged instructions with binary rewriting, but they can only eliminate part of the privileged instructions because x86 does not enforce alignment of instructions. LVD and UnderBridge isolate all privileged instructions with hardware virtualization extension, but this extension may be unavailable for secure containers in nested clouds. LVD, UnderBridge, and BULKHEAD protect interrupts by redirecting them to the core kernel, but they overlook potential interrupt forgery

Table 5. Comparison between CKI and prior work on intra-kernel isolation domains.

Aspect	Nested Kernel	LVD	UnderBridge	NICKLE	SILVER	BULKHEAD	CKI
Scalable Isolation Domains		✓			✓	✓	✓
Secure & Efficient Pgtbl. Mgmt.	✓				✓	✓	✓
No Reliance on Virt. HW	✓			✓		✓	✓
Complete Priv. Inst. Isolation		✓	✓				✓
Interrupt Redirection		✓	✓			✓	✓
Interrupt Forgery Prevention							✓

attacks. For example, LVD allows a malicious kernel module to invoke an interrupt handler of the core kernel by jumping to the *vmfunc*-based switch gate.

The concept of our hardware extension shares similarities with x86 ring-1 [14], which provides an additional privilege level where privileged instructions are disabled. CKI revives this concept for secure containers, which is the first attempt. Inspired by VTx for VMs, we investigate hardware extensions to accelerate container virtualization.

Nested virtualization. Turtles [34] was the first implementation of hardware-assisted nested virtualization on x86. Intel adds a hardware extension for nested virtualization called VMCS shadowing [14], which allows the L1 kernel to read-/write VMCS without trapping into the L0 kernel. NEVE [51] add a similar extension on ARM platform. TLFS [11] paravirtualizes the VMCS accesses in L1 to reduce VM exits to L0. Despite these optimizations, hardware-assisted nested virtualization is still inefficient due to L0 intervention and shadow EPT management. DVH [52] enables the L0 kernel to directly provide virtual hardware to L2 VMs without the intervention of the L1 kernel, reducing the overhead of VM exits. It optimizes I/O-intensive applications, but the memory management overhead is still high. Meanwhile, DVH relies on modification in the L0 kernel, which is not feasible for an IaaS tenant. DMT [69] proposes a hardware extension for three-stage translation, avoiding shadow paging. CKI eliminates the overhead of shadow paging with a simpler hardware extension (PKS). SVT [65] reduces the overhead of nested VM exits by running the hypervisor and the VMs on different hardware threads and replacing VM exits with simple thread stall and resume events.

9 Conclusion and Future Work

CKI is a hardware-software co-design for building efficient secure containers. By leveraging and extending lightweight CPU features for intra-kernel isolation, CKI efficiently constructs a new privilege level for container kernels, harmonizing the performance overhead and security isolation and outperforming state-of-the-art secure container designs.

While our PKS extensions are not available in current CPUs, we believe they will enable new opportunities for future use cases including the following: *Sandboxing untrusted kernel drivers*: Directly isolating drivers within ring-0, eliminating the need to deprivilege them to ring-3 as in microkernel designs, thus avoiding additional performance overhead on user-kernel or inter-process communication. *Kernel-level syscall optimization*: Running syscall-intensive applications within the kernel to achieve better performance by eliminating the traditional syscall overhead. We plan to explore these directions in our future work.

10 Acknowledgment

We sincerely thank our shepherd Anton Burtsev and the anonymous reviewers, whose reviews, feedbacks, and suggestions have significantly strengthened our work. This research was supported in part by the National Natural Science Foundation of China (No. 62432010,62202292), the Fundamental Research Funds for the Central Universities, and research grants from Huawei Technologies and Intel. Corresponding author: Jinyu Gu (gujinyu@sjtu.edu.cn).

References

[1] Apparmor. https://apparmor.net.
[2] capabilities(7) — linux manual page. https://man7.org/linux/man-pages/man7/capabilities.7.html.
[3] The container security platform | gvisor. https://gvisor.dev.
[4] CVE-2022-0185 in linux kernel can allow container escape in kubernetes. https://www.aquasec.com/blog/cve-2022-0185-linux-kernel-container-escape-in-kubernetes/.
[5] db_bench_sqlite3.cc. https://github.com/google/leveldb/blob/main/benchmarks/db_bench_sqlite3.cc.
[6] Dirty COW (CVE-2016-5195). https://dirtycow.ninja.
[7] The dirty pipe vulnerability. https://dirtypipe.cm4all.com.
[8] gem5: The gem5 simulator system. https://www.gem5.org.
[9] gvisor - platform guide. https://gvisor.dev/docs/architecture_guide/platforms/.
[10] HPCCHALLENGE - RandomAccess. https://hpcchallenge.org/projectsfiles/hpcc/RandomAccess.html.
[11] Hypervisor top level functional specification. https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/tlfs/tlfs.
[12] Indirect branch predictor barrier. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-predictor-barrier.html.
[13] Indirect branch restricted speculation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html.
[14] Intel 64 and ia-32 architectures software developer manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[15] Kata containers - open source container runtime software. https://katacontainers.io.

[16] Linux kernel vulnerability: Escaping containers by abusing cgroups. https://www.aquasec.com/blog/new-linux-kernel-vulnerability-escaping-containers-by-abusing-cgroups/.

[17] Lmbench - tools for performance analysis. https://lmbench.sourceforge.net.

[18] memcached - a distributed memory object caching system. https://memcached.org.

[19] memtier_benchmark. https://github.com/RedisLabs/memtier_benchmark.

[20] namespaces(7) — linux manual page. https://man7.org/linux/man-pages/man7/namespaces.7.html.

[21] Page table isolation (pti). https://docs.kernel.org/next/x86/pti.html.

[22] Quark: A secure container runtime with oci interface. https://github.com/QuarkContainer/Quark.

[23] Redis - the real-time data platform. https://redis.io.

[24] Releasing systrap - a high-performance gvisor platform. https://gvisor.dev/blog/2023/04/28/systrap-release/.

[25] Retpoline: A branch target injection mitigation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/retpoline-branch-target-injection-mitigation.html.

[26] runc. https://github.com/opencontainers/runc.

[27] seccomp(2) — linux manual page. https://man7.org/linux/man-pages/man2/seccomp.2.html.

[28] Sqlite home page. https://www.sqlite.org.

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

[30] Anjali, Tyler Caraza-Harter, and Michael M. Swift. Blending containers and virtual machines: a study of firecracker and gvisor. In Santosh Nagarakatte, Andrew Baumann, and Baris Kasikci, editors, *VEE '20: 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, virtual event [Lausanne, Switzerland], March 17, 2020*, pages 101–113. ACM, 2020.

[31] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. Scone: Secure linux containers with intel sgx. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 689–703, USA, 2016. USENIX Association.

[32] 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 Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 164–177, New York, NY, USA, 2003. Association for Computing Machinery.

[33] Jonathan Behrens, Anton Cao, Cel Skeggs, Adam Belay, M. Frans Kaashoek, and Nickolai Zeldovich. Efficiently mitigating transient execution attacks using the unmapped speculation contract. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 1139–1154. USENIX Association, 2020.

[34] 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 Remzi H. Arpaci-Dusseau and Brad Chen, editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 423–436. USENIX Association, 2010.

[35] Xiangdong Chen, Zhaofeng Li, Tirth Jain, Vikram Narayanan, and Anton Burtsev. Limitations and opportunities of modern hardware isolation mechanisms. In *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'24, USA, 2024. USENIX Association.

[36] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram S. Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, pages 191–206. ACM, 2015.

[37] Shulin Fan, Zhichao Hua, Yubin Xia, Haibo Chen, and Binyu Zang. Isa-grid: Architecture of fine-grained privilege control for instructions and registers. In Yan Solihin and Mark A. Heinrich, editors, *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA 2023, Orlando, FL, USA, June 17-21, 2023*, pages 15:1–15:15. ACM, 2023.

[38] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Íñigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. Making kernel bypass practical for the cloud with junction. In Laurent Vanbever and Irene Zhang, editors, *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*, pages 55–73. USENIX Association, 2024.

[39] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. Agile paging: Exceeding the best of nested and shadow paging. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, pages 707–718. IEEE Computer Society, 2016.

[40] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 401–417, July 2020.

[41] Yinggang Guo, Zicheng Wang, Weiheng Bai, Qingkai Zeng, and Kangjie Lu. BULKHEAD: secure, scalable, and efficient kernel compartmentalization with PKS. *CoRR*, abs/2409.09606, 2024.

[42] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In Dahlia Malkhi and Dan Tsafrir, editors, *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 489–504. USENIX Association, 2019.

[43] Alexander Van't Hof and Jason Nieh. BlackBox: A container security monitor for protecting containers on untrusted operating systems. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 683–700, Carlsbad, CA, July 2022. USENIX Association.

[44] Zhichao Hua, Yang Yu, Jinyu Gu, Yubin Xia, Haibo Chen, and Binyu Zang. Tz-container: protecting container from untrusted OS with ARM trustzone. *Sci. China Inf. Sci.*, 64(9), 2021.

[45] Hang Huang, Jiangshan Lai, Jia Rao, Hui Lu, Wenlong Hou, Hang Su, Quan Xu, Jiang Zhong, Jiahao Zeng, Xu Wang, Zhengyu He, Weidong Han, Jiang Liu, Tao Ma, and Song Wu. PVM: efficient shadow paging for deploying secure containers in cloud-native environment. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 515–530. ACM, 2023.

[46] Hang Huang, Honglei Wang, Jia Rao, Song Wu, Hao Fan, Chen Yu, Hai Jin, Kun Suo, and Lisong Pan. vkernel: Enhancing container isolation via private code and data. *IEEE Transactions on Computers*, 2024.

[47] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas

Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[48] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 376–394, New York, NY, USA, 2021. Association for Computing Machinery.

[49] Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. Lock-in-pop: Securing privileged operating system kernels by keeping on the beaten path. In Dilma Da Silva and Bryan Ford, editors, *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 1–13. USENIX Association, 2017.

[50] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. Rund: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In Jiri Schindler and Noa Zilberman, editors, *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 53–68. USENIX Association, 2022.

[51] 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, Shanghai, China, October 28-31, 2017*, pages 201–217. ACM, 2017.

[52] Jin Tack Lim and Jason Nieh. Optimizing nested virtualization performance using direct virtual hardware. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 557–574. ACM, 2020.

[53] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. Dirtycred: Escalating privilege in linux kernel. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 1963–1976. ACM, 2022.

[54] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. Meltdown: Reading kernel memory from user space. *Commun. ACM*, 63(6):46–56, may 2020.

[55] Lukas Maar, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. Dope: Domain protection enforcement with pks. In *Proceedings of the 39th Annual Computer Security Applications Conference*, ACSAC '23, page 662–676, New York, NY, USA, 2023. Association for Computing Machinery.

[56] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery.

[57] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight kernel isolation with virtualization and VM functions. In Santosh Nagarakatte, Andrew Baumann, and Baris Kasikci, editors, *VEE '20: 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, virtual event [Lausanne, Switzerland], March 17, 2020*, pages 157–171. ACM, 2020.

[58] Zhenhao Pan, Qing He, Wei Jiang, Yu Chen, and Yaozu Dong. Nestcloud: Towards practical nested virtualization. In *2011 International Conference on Cloud and Service Computing, CSC 2011, Hong Kong, December 12-14, 2011*, pages 321–329. IEEE Computer Society, 2011.

[59] Ashish Panwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K. Gopinath, and Jayneel Gandhi. Fast local page-tables for virtualized NUMA servers with vmitosis. In Tim Sherwood, Emery D. Berger, and Christos Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 194–210. ACM, 2021.

[60] Ryan D. Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In Richard Lippmann, Engin Kirda, and Ari Trachtenberg, editors, *Recent Advances in Intrusion Detection, 11th International Symposium, RAID 2008, Cambridge, MA, USA, September 15-17, 2008. Proceedings*, volume 5230 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2008.

[61] Michael Roitzsch, Till Miemietz, Christian von Elm, and Nils Asmussen. Software-defined CPU modes. In Malte Schwarzkopf, Andrew Baumann, and Natacha Crooks, editors, *Proceedings of the 19th Workshop on Hot Topics in Operating Systems, HOTOS 2023, Providence, RI, USA, June 22-24, 2023*, pages 23–29. ACM, 2023.

[62] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 121–135, New York, NY, USA, 2019. Association for Computing Machinery.

[63] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, page 275–287, New York, NY, USA, 2007. Association for Computing Machinery.

[64] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: secure, efficient in-process isolation with protection keys (MPK). In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1221–1238. USENIX Association, 2019.

[65] Lluís Vilanova, Nadav Amit, and Yoav Etsion. Using SMT to accelerate nested virtualization. In Srilatha Bobbie Manne, Hillery C. Hunter, and Erik R. Altman, editors, *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 750–761. ACM, 2019.

[66] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, pages 199–211. ACM, 2018.

[67] Xi Xiong and Peng Liu. SILVER: fine-grained and transparent protection domain primitives in commodity OS kernel. In Salvatore J. Stolfo, Angelos Stavrou, and Charles V. Wright, editors, *Research in Attacks, Intrusions, and Defenses - 16th International Symposium, RAID 2013, Rodney Bay, St. Lucia, October 23-25, 2013. Proceedings*, volume 8145 of *Lecture Notes in Computer Science*, pages 103–122. Springer, 2013.

[68] Xusheng Zhan, Yungang Bao, Christian Bienia, and Kai Li. PARSEC3.0: A multicore benchmark suite with network stacks and SPLASH-2X. *SIGARCH Comput. Archit. News*, 44(5):1–16, 2016.

[69] Jiyuan Zhang, Weiwei Jia, Siyuan Chai, Peizhe Liu, Jongyul Kim, and Tianyin Xu. Direct memory translation for virtualized clouds. In Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir, editors, *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pages 287–304. ACM, 2024.

A Artifact Appendix

A.1 Abstract

The artifact contains the source code of our prototype system, the scripts for conducting experiments presented in the paper, and the applications used for those experiments. It also includes a prebuilt VM image for running the experiments in a nested cloud setting (containers running inside the VM).

A.2 Description & Requirements

A.2.1 How to access. The artifact is available at https://doi.org/10.5281/zenodo.14931956.

A.2.2 Hardware dependencies. This artifact requires AMD CPUs and has been tested on AMD EPYC-9654 and AMD EPYC-7T83 machines. The machine should have at least 8 GB of memory and 32 GB of free disk space.

A.2.3 Software dependencies. To run the experiments with the prebuilt images, the machine should operate on a Linux OS (Ubuntu 22.04 with Linux 6.7.0-rc6 tested) and have the following dependencies installed: qemu-system-x86_64 (v6.2.0 tested), expect, redis-cli, nc, Docker, Python 3, matplotlib, and numpy.

A.2.4 Benchmarks. The Dockerfiles for building the container applications are available in the `apps` directory. Memcached and Redis are evaluated using memtier_benchmark, which can be downloaded from Docker Hub.

A.3 Set-up

The user needs to download the prebuilt images, fill out a configuration file, and create a tap device on the host OS. Please refer to the README for details.

A.4 Evaluation workflow

A.4.1 Major Claims.

- *C1*: Compared with HVM-NST and PVM, CKI reduces the latencies of page-fault-intensive applications (from PARSEC/vmitosis-workloads) by up to 72% and 47%, respectively. This is proven by experiment *E1* whose results are reported in Figure 12.
- *C2*: Compared with PVM, CKI increases the throughput of sqlite benchmark by up to 24%. This is proven by experiment *E2* whose results are reported in Figure 14.
- *C3*: Compared with HVM-NST, CKI-NST obtains 6.8× throughput for memcached and 2.0× throughput for Redis. This is proven by experiment *E3* whose results are reported in Figure 16.

A.4.2 Experiments.

- Experiment *E1* (20 compute-minutes): Run script `scripts/run_fig12.sh`, which executes the page-fault-intensive applications using HVM-NST, PVM, and CKI. The latency results will be displayed in `plots/fig12.pdf`.
- Experiment *E2* (20 compute-minutes): Run script `scripts/run_fig14.sh`, which executes the sqlite benchmark using HVM, PVM, and CKI. The throughput results will be displayed in `plots/fig14.pdf`.
- Experiment *E3* (40 compute-minutes): Run script `scripts/run_fig16.sh`, which evaluates the throughput of Redis and Memcached using HVM-NST, PVM-NST, and CKI-NST with different numbers of clients. The results will be displayed in `plots/fig16.pdf`.