

# PVM: Efficient Shadow Paging for Deploying Secure Containers in Cloud-native Environments

Hang Huang<sup>1†</sup>, Jiangshan Lai<sup>2†</sup>, Jia Rao<sup>3</sup>, Hui Lu<sup>3</sup>, Wenlong Hou<sup>2</sup>, Hang Su<sup>2</sup>, Quan Xu<sup>1</sup>, Jiang Zhong<sup>1</sup>, Jiahao Zeng<sup>1</sup>, Xu Wang<sup>2</sup>, Zhengyu He<sup>2</sup>, Weidong Han<sup>1</sup>, Jiang Liu<sup>1</sup>, Tao Ma<sup>1</sup>, Song Wu<sup>4</sup>

<sup>1</sup> Alibaba Group, <sup>2</sup> Ant Group

<sup>3</sup> The University of Texas at Arlington

<sup>4</sup> Huazhong University of Science and Technology

Email: {huanghang.huang, wutu.xq, zhongjiang.zj, zengjiahao.zjh, shaokang.hwd, liu.jiang, boyu.mt}@alibaba-inc.com, {jiangshan.ljs, houwenlong.hwl, darcy.sh, gnawux, zhengyu.he}@antgroup.com, {jia.rao, hui.lu}@uta.edu, {wusong}@hust.edu.cn

## Abstract

In cloud-native environments, containers are often deployed within lightweight virtual machines (VMs) to ensure strong security isolation and privacy protection. With the growing demand for customized cloud services, third-party vendors are turning to infrastructure-as-a-service (IaaS) cloud providers to build their own cloud-native platforms, necessitating the need to run a VM or a guest that hosts containers inside another VM instance leased from an IaaS cloud. State-of-the-art nested virtualization in the x86 architecture relies heavily on the host hypervisor to expose hardware virtualization support to the guest hypervisor, not only complicating cloud management but also raising concerns about an increased attack surface at the host hypervisor.

This paper presents the design and implementation of PVM, a high-performance guest hypervisor for KVM that is transparent to the host hypervisor and assumes no hardware virtualization support. PVM leverages two key designs: 1) a minimal shared memory region between the guest and guest hypervisor to facilitate state transition between different privilege levels and 2) an efficient shadow page table design to reduce the cost of memory virtualization. PVM has been adopted by Alibaba Cloud for hosting tens of thousands of secure containers on a daily basis. Our experiments demonstrate that PVM significantly outperforms current nested virtualization in KVM for memory virtualization, particularly for concurrent workloads, while maintaining comparable performance in CPU and I/O virtualization.

**CCS Concepts:** • Software and its engineering → Virtual machines; Virtual memory.

## ACM Reference Format:

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, Song Wu. 2023. PVM: Efficient Shadow Paging for Deploying Secure Containers in Cloud-native Environments. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23)*, October 23–26, 2023, Koblenz, Germany. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3600006.3613158>

## 1 Introduction

Containers have gained widespread popularity for building and deploying applications in cloud-native data centers owing to their flexibility, portability, minimal resource utilization, and excellent scalability. Unlike traditional virtual machine (VM)-based virtualization [23, 32], containers share the kernel with the host OS and therefore do not offer the same level of isolation between hosted applications, opening up opportunities for attackers to cause information leakage [53], privilege escalation [45], and denial of services [11]. Although there has been extensive research dedicated to improving container isolation and security, including operating system (OS)-level mechanisms such as *namespaces*, *capabilities* [34], *seccomp* [28], and *apparmor* [33], as well as user-level kernels like gVisor [10, 39], VM-based isolation methods such as Kata Containers [6] and FireCracker [15] remain the preferred and widely adopted choice in production systems. These VM-based solutions deploy containers with separate OS kernels within lightweight VMs, offering robust isolation and compatibility with legacy applications.

The additional layer of indirection at the VM *hypervisor*, however, incurs non-negligible overhead. Besides an enlarged per-container memory footprint, a primary source of overhead is the crossing of multiple layers of the virtualization stack (i.e., world switches) for CPU, memory, and I/O virtualization [19]. As the demand for user-defined, private cloud-native computing continues to grow, there is a need for running lightweight VMs within VM instances leased

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org). SOSP '23, October 23–26, 2023, Koblenz, Germany

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0229-7/23/10...\$15.00  
<https://doi.org/10.1145/3600006.3613158>

<sup>†</sup>These authors contributed equally to this work and should be considered joint first authors.

from public infrastructure-as-a-service (IaaS) clouds. For example, running Kata containers with Kubernetes [22] in the cloud requires nested VMs, allowing for greater isolation, more flexible, and elastic Kubernetes cluster management but resulting in even higher runtime overhead.

This paper analyzes state-of-the-art nested virtualization approaches and identifies a fundamental tradeoff – while hardware virtualization support, such as Intel VT-x [40] and AMD-V [47], offers superior performance and lower overhead compared with software-based emulation for single-level virtualization, it may result in sub-optimal performance and excessive (expensive) traps to the host hypervisor (the most privileged level) in 2-level nested virtualization, particularly memory virtualization. Hardware-assisted nested virtualization also complicates the design of the host hypervisor, expands its potential attack surface, and introduces new scalability issues.

In single-level memory virtualization, a VM’s virtual addresses need to be translated to VM physical addresses (level-1 or  $L_1$ ) and then to host physical addresses ( $L_0$ ). The two-level translations can be done entirely in software or assisted by hardware [25]. Without hardware support, the memory management unit (MMU) can only register a single page table. To run multiple VMs, the hypervisor maintains and registers with MMU a shadow page table (SPT) [35] that translates directly from VM virtual addresses to host physical addresses for each VM. Shadow paging requires that any changes to a VM page table must trap to the hypervisor to update the shadow page table accordingly. To assist memory virtualization, modern x86 processors add a second page table (EPT by Intel and NPT by AMD) to translate VM physical address to host physical address. This allows both levels of translation to be performed by hardware, thus eliminating the need for software emulation and hypervisor involvement.

There have been significant studies comparing software and hardware virtualization techniques and their associated overhead in single-level virtualization [13]. These studies have motivated the development of a hybrid approach that leverages the advantages of both software and hardware techniques. Nested virtualization, on the other hand, necessitates such a hybrid virtualization approach due to the limited availability of hardware support. Nested memory virtualization requires the multiplexing of three levels of address translation, i.e., from guest virtual addresses to guest physical addresses ( $L_2$ ), to VM physical addresses ( $L_1$ ), and eventually to host physical addresses ( $L_0$ ) on the 2-level paging hardware. One approach is to maintain an SPT at the  $L_1$  or guest hypervisor to handle the top two levels of translation. A more popular and the default approach in KVM [20] is to allow the  $L_2$  guest to use its own page table while the  $L_0$  hypervisor maintains a compressed EPT or NPT for translating  $L_2$  guest physical addresses to  $L_0$  physical addresses (the bottom two levels). Nevertheless, both approaches involve

an excessive number of world switches, many of which are unnecessary but expensive exits to the  $L_0$  hypervisor for assisting switches between the  $L_2$  guest and  $L_1$  guest hypervisor. As the number of  $L_2$  guests increases, the complexity to update and synchronize the consolidated page tables also increases, causing performance and scalability bottlenecks.

This paper explores the design space of a hybrid software-hardware approach for nested virtualization and argues that memory virtualization for an  $L_2$  guest should be exclusively performed by and within the  $L_1$  hypervisor, assuming no hardware support and being transparent to the  $L_0$  hypervisor. This design offers several advantages: **1)** decoupling nested virtualization from the host hypervisor does not add complexity, compromise security, or impact the underlying cloud management’s elasticity. **2)** Handling nested virtualization entirely within the guest hypervisor enables user-specific optimizations for security, performance, and scalability.

This paper presents PVM, a KVM guest hypervisor optimized for nested memory virtualization. PVM is a software paravirtualization solution within the  $L_1$  VM, requiring no changes to the  $L_0$  host hypervisor. To gain full control over  $L_2$  guest’s memory virtualization, PVM employs software-based shadow paging for translating  $L_2$  virtual addresses to  $L_1$  physical addresses and relies on hardware EPT or NPT for the remaining translation to  $L_0$  host physical addresses. This allows PVM to work with unmodified KVM host hypervisors and allows nested and ordinary VMs to coexist. Additionally, to trap all  $L_2$  guest page table updates while excluding the  $L_0$  hypervisor’s involvement, PVM avoids the use of hardware virtualization extensions on x86 processors, such as Intel VMX, when performing *world switches* [24] within the  $L_1$  VM. The nested CPU and memory virtualization based on software emulation effectively isolates an  $L_2$  guest within an  $L_1$  VM, but it also raises concerns about possible performance degradation compared to hardware-assisted methods.

PVM demonstrates that software-based world switches can be made as efficient (or with comparable performance) as hardware-assisted approaches for single-level virtualization. For nested memory virtualization, PVM’s combination of software (i.e., shadow paging and world switching within  $L_1$  VM) and hardware (EPT and VMX support at the  $L_0$  hypervisor) approaches significantly outperforms hardware (only)-assisted approaches. To achieve this, PVM involves three important designs: **1)** the placement of the user and kernel spaces of an  $L_2$  guest entirely in  $L_1$  VM’s privilege level 3 (Ring 3) to ensure all privileged instructions, system calls, and exceptions trap to PVM for emulation. This design is necessary in the latest AMD processors [27] and the upcoming Intel’s x86-s [9] architecture that both remove Ring 1 and 2 for a simplified CPU instruction set [43]. **2)** A piece of highly-efficient assembly code, called *switcher*, residing in a shared memory region between an  $L_2$  guest and the  $L_1$  VM to facilitate world switches among the user and kernel spaces of an  $L_2$  guest as well as PVM, the  $L_1$  hypervisor.

3) An efficient and scalable shadow page table design enabled by PVM’s nested virtualization. PVM features a parallel SPT and a pre-fault mechanism (and many optimizations) to accelerate guest page fault handling.

PVM is production-ready and has been adopted by Alibaba Cloud. It runs tens of thousands of secure containers in nested VMs daily with unmodified KVM host hypervisors. Extensive experiments with micro-benchmarks and real applications show that PVM achieves comparable performance in VM entry/exit, a key indicator of world switching overhead, compared to hardware-assisted CPU virtualization in single-level virtualization, and attains up to an order of magnitude performance improvement in memory-intensive, concurrent workloads.

In summary, this paper makes the following contributions:

- A comprehensive analysis of state-of-the-art nested virtualization that reveals the drawbacks of hardware-assisted approaches and motivates the hybrid software-hardware nested architecture, as well as decoupling nested VMs from the host hypervisor.
- The design and implementation of PVM, a guest hypervisor, and a general framework for nested virtualization that enables user-defined optimizations.
- Optimizations for world switches, concurrent shadow page table updates, and guest page fault handling.

## 2 Background and Motivation

As illustrated in Figure 1, nested virtualization enables a *host hypervisor* (denoted as the  $L_0$  hypervisor) to run another *guest hypervisor* (denoted as the  $L_1$  hypervisor) within a VM, which can in turn run additional VMs (denoted as the  $L_2$  guest). This 2-level nested virtualization facilitates the deployment of secure containers, which often run inside lightweight VMs for strong isolation, within cloud instances leased from IaaS providers. Unfortunately, current hardware (e.g., Intel VT-x [40] and AMD-V [47]) only has a single level of architectural support for virtualization. As a result, implementing nested virtualization requires complex and costly software efforts, especially in the  $L_0$  hypervisor. In this section, we compare 2-level nested virtualization with traditional single-level virtualization (based on Intel VT-x) and identify the primary sources of overhead in nested virtualization. This analysis motivates us to develop PVM, a more efficient nested virtualization approach.

### 2.1 Overview of nested virtualization

**Single-level virtualization.** Intel VT-x [46] offers virtualization extensions (VMX) to support an additional *non-root* operation mode, coexisting with the traditional *root* operation mode, each with privilege levels ranging from 0 to 3. In single-level virtualization, the host hypervisor operates at privilege level 0 of the root mode with full privilege, whereas the guest VM (kernel and user) operates at privilege level 0 and 3 of the non-root mode with reduced privilege. The

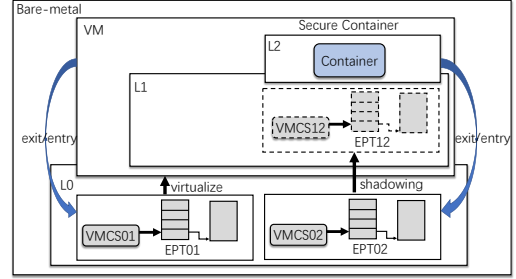


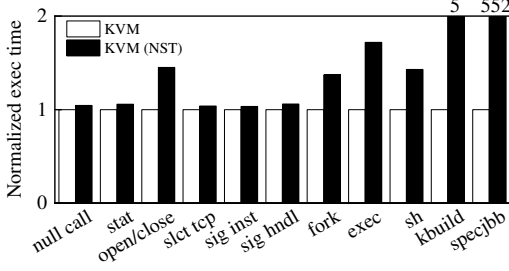
Figure 1. Secure containers with nested virtualization.

communication between a guest VM and the hypervisor occurs through a set of VMX instructions – e.g., VM Exit and VM Entry. In the non-root mode, a VM Exit occurs when a privileged instruction is executed or an interrupt is generated. This triggers the CPU to switch to the root mode, where the hypervisor handles the VM Exit via emulation. Afterward, the hypervisor uses the VM Entry instruction (i.e., VMLAUNCH/VMRESUME) to resume the guest VM’s execution. The hypervisor/guest transition involves the switching of both the execution modes (i.e., root and non-root mode) and the hypervisor and guest address spaces. Such context switches, referred to as *world switches*, introduce nontrivial overhead.

To preserve the execution contexts of a guest VM and the hypervisor during world switches, VMX maintains a VM control structure (VMCS) per virtual CPU (vCPU). Upon a VM Exit, the processor saves the guest VM’s CPU states (context) in the VMCS, while loading the hypervisor’s CPU states from the VMCS, and vice versa for VM entry.

**2-level nested virtualization.** The existing 2-level nested virtualization builds upon the hardware support for single-level virtualization described above but requires additional software effort. The  $L_0$  hypervisor operates at the most privileged level – level 0 of the root mode. It manages the underlying hardware and emulates VMX for the guest hypervisor  $L_1$ . With the emulated VMX, an unmodified hypervisor can run at  $L_1$  without the awareness of the lower  $L_0$ . Similar to a traditional hypervisor, an  $L_1$  hypervisor can create its own  $L_2$  guest VMs. Both  $L_1$  and  $L_2$  operate in the *non-root* mode.

Such a nested architecture involves complex and expensive transitions between multiple hypervisors and guest VMs. For example, in Figure 1, any privileged instruction from  $L_2$  will first be trapped by  $L_0$  (via a VM Exit). Since the privileged instruction needs to be handled by  $L_2$ ’s hypervisor, i.e.,  $L_1$ ,  $L_0$  forwards the trap to  $L_1$  and uses VM Entry to switch to  $L_1$ , which in turn handles the injected trap (originally from  $L_2$ ). Upon the completion of trap handling,  $L_1$  attempts to resume running  $L_2$  using the VM Entry instruction. As the VM Entry is a privileged instruction,  $L_1$  is trapped by  $L_0$  again, where the VM Entry instruction is executed in the root mode to finally resume  $L_2$ . As it can be seen, in 2-level nested virtualization, a transition between  $L_2$  and  $L_1$  (e.g., trap handling) requires an exit to  $L_0$ , doubling the number of world switches compared to that in single-level virtualization.



**Figure 2.** Overhead analysis of nested virtualization.

To reduce the overhead of world switches, nested virtualization employs a shadowing mechanism to maintain VMCS during switches. As depicted in Figure 1,  $L_0$  is responsible for maintaining  $L_1$ 's VMCS, denoted as  $VMCS_{01}$ , while  $L_1$  maintains  $L_2$ 's VMCS, denoted as  $VMCS_{12}$ . Since  $L_1$  works in the non-root mode, any operation it performs on  $VMCS_{12}$ , such as VMREAD or VMWRITE, will cause an exit to  $L_0$ . As studied in [49], to handle a single world switch from  $L_2$ ,  $L_1$  needs to access  $VMCS_{12}$  multiple times, leading to as many as 40-50 exits to  $L_0$ . To avoid excessive  $L_0$  exits, VMCS shadowing allows  $L_1$  to maintain a software-based  $VMCS_{12}$  that is not directly used for running  $L_2$ . Instead,  $L_0$  maintains a shadow  $VMCS_{02}$  that is created by merging and synchronizing  $VMCS_{01}$  and  $VMCS_{12}$ . As all reads/writes to VMCS are handled by  $L_0$  in the root mode, exits to  $L_0$  due to  $L_1$ 's accesses to VMCS are thus eliminated.

Despite these optimizations, nested virtualization remains expensive, especially for memory-intensive workloads. We conducted performance tests using *LMbench* [8], *Kbuild* [26], and *SPECjbb2005* [14] within one or multiple secure containers and compared the performance of single-level virtualization (the secure container running in a regular VM) with that of 2-level nested virtualization (the secure container running in an  $L_2$  guest). Note that each LMbench benchmark ran in a single container while Kbuild and SPECjbb2005 each used 16 containers to study the cost of nested virtualization for concurrent workloads. Figure 2 shows that while nested virtualization introduces additional world switches for handling privileged instructions, its overhead for workloads that do not involve intensive memory access is negligible. On the contrary, when dealing with memory-intensive workloads like frequent updates to page tables, virtual address translations, and especially simultaneous memory accesses, the performance slowdown is substantial, reaching up to two orders of magnitude. In what follows, we discuss memory virtualization and analyze its overhead.

## 2.2 Memory virtualization

Memory virtualization is currently facilitated by architectural features such as Extended Page Tables (EPT) in Intel VT-x or Nested Page Tables in AMD-V. Without loss of generality, we use EPT as an example to introduce both single and 2-level memory virtualization.

**Single-level memory virtualization** involves *two dimensional* page tables – 1) the regular guest page table (GPT)

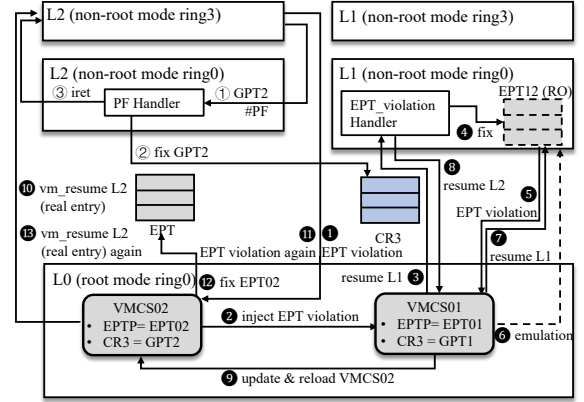
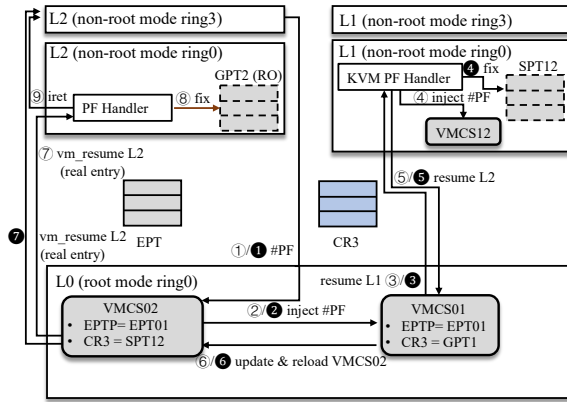
maps guest virtual addresses (GVA) to guest physical addresses (GPA); 2) the extended page table (EPT) maps GPA to host physical addresses (HPA). The hypervisor is responsible for maintaining the EPT while a guest manages its own GPT. Since GPT is visible to the memory management unit (MMU) and guest address translations are performed by hardware, guest page faults can be entirely handled by a guest kernel in the non-root mode without exits to the hypervisor. EPT violations, i.e., entries absent in the EPT, cause hypervisor page faults and are handled by the hypervisor in the root mode.

**SPT-on-EPT.** To support 2-level nested memory virtualization, *three dimensional* page tables are required: one for the  $L_2$  guest, one for the  $L_1$  guest hypervisor, and one for the  $L_0$  host hypervisor. One straightforward approach to achieve this is by using shadow page tables (SPT) in  $L_1$ , while still allowing  $L_0$  to use the EPT hardware, called *SPT-on-EPT*.

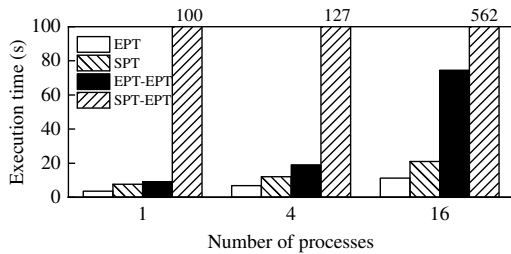
As illustrated in Figure 3(a),  $L_2$  manages its guest page table, GPT2, which maps  $GVA_{L_2}$  to  $GPA_{L_2}$ .  $L_1$  manages its guest page table, GPT1, which maps  $GPA_{L_2}$  to  $GPA_{L_1}$ .  $L_0$  uses EPT01 to map  $GPA_{L_1}$  to HPA. As hardware MMU only supports two page tables,  $L_1$  combines GPT1 and GPT2 to create shadow page tables, SPT12, mapping  $GVA_{L_2}$  directly to  $GPA_{L_1}$ . To keep SPT12 synchronized with GPT2, every update made to GPT2 by  $L_2$  must trigger a page fault, and trap to the  $L_1$  hypervisor. To achieve this, GPT2 is made *read-only* to  $L_2$ , but *writable* to  $L_1$ . When  $L_2$  is running,  $L_0$  provides both SPT12 and EPT01 to MMU for address translation, where MMU first uses SPT12 to translate  $L_2$ 's GVA to  $L_1$ 's physical address; further, MMU uses EPT01 to translate  $L_1$ 's physical address to  $L_0$ 's HPA. Despite using the EPT hardware, SPT-on-EPT incurs significant overhead, because each page fault in  $L_2$  requires  $L_1$  to update the shadow page tables. As described in Section 2.1, these  $L_2$  page faults/GPT2 updates must first trap to  $L_0$  and then be forwarded to  $L_1$ , resulting in excessive and expensive world switches.

As an example, in Figure 3(a), a page fault from  $L_2$  can trigger two phases of page table updates: 1) The first phase updates  $L_2$ 's GPT2; and 2) the second phase updates  $L_1$ 's GPT1 and SPT12. More specifically, in the first phase, an access to an absent  $GVA_{L_2}$  in GPT2 triggers an  $L_2$  page fault and an exit to  $L_0$  (①), which injects the page fault to  $L_1$  by writing  $VMCS_{01}$  (②) and resumes executing  $L_1$  (③). The  $L_1$  hypervisor recognizes that this is an  $L_2$  page fault and injects it to  $L_2$ 's  $VMCS_{12}$  (④) and attempts to resume  $L_2$ . Resuming  $L_2$  needs to trap back to  $L_0$  (⑤), which updates  $VMCS_{02}$  (⑥) before finally resuming  $L_2$  using  $VMCS_{02}$  (⑦). The page fault handler in  $L_2$  updates GPT2 with new page table entries mapping  $GVA_{L_2}$  to  $GPA_{L_2}$  (⑧) and finally returns to user (⑨). To add more complexity to this process – since GPT2 is *read-only*,

Without loss of generality, we assume  $L_1$  is warmed up and EPT01 has been established (i.e., no EPT violations from EPT01) in SPT-on-EPT and all other memory virtualization approaches.



**Figure 3.** Two approaches, (a) SPT-on-EPT and (b) EPT-on-EPT, for 2-level nested memory virtualization.



**Figure 4.** The performance comparison between EPT and SPT w/wo nested virtualization.

every update to GPT2 needs assistance from  $L_1$ . Hence, step ⑧ could further cause multiple rounds of switches between  $L_2$  and  $L_1$  via  $L_0$  (not illustrated in Figure 3(a)), depending on the number of page table levels. For example, a 4-level GPT2 needs four rounds of such switches. In the first phase,  $L_0$  can be involved up to  $2n + 2$  times with  $4n + 4$  world switches (where  $n$  is the number of  $L_2$  page table levels).

The following access to  $\text{GPA}_{L_2}$ , which was added to GPT2 during the first phase, triggers a page fault in  $L_1$  since  $\text{GPA}_{L_2}$  is absent from GPT1 and hence SPT12. The second phase handles this  $L_1$  page fault and updates SPT12. As shown in Figure 3(a), the second phase follows the same steps as the first phase (❶–❷), except for two differences. First, at step ❹ the  $L_1$  kernel handles the page fault and updates SPT12. Second, step ❷ returns directly to  $L_2$  user space without involving the  $L_2$  guest kernel. In the second phase, there are *four* more world switches, including trapping to  $L_0$  *twice*.

In summary, in the worst-case scenario, in which both GPT2 and SPT12 need to be updated, and assuming an  $n$ -level GPT2, an  $L_2$  page fault can lead to  $4n + 8$  world switches and  $2n + 4$  exits to  $L_0$ .

**EPT-on-EPT.** The state-of-the-art nested memory virtualization approach for KVM leverages the architectural support for 2-dimensional page tables in EPT.

As illustrated in Figure 3 (b),  $L_2$  has its own GPT2, which maps  $\text{GVA}_{L_2}$  to  $\text{GPA}_{L_2}$ .  $L_0$  exposes EPT capabilities to  $L_1$ ,

which creates and maintains an EPT12 table mapping  $\text{GPA}_{L_2}$  to  $\text{GPA}_{L_1}$ . Same as SPT-on-EPT,  $L_0$  manages EPT01, mapping  $\text{GPA}_{L_1}$  to HPA. Because MMU only supports one EPT table,  $L_0$  compresses EPT01 and EPT12 into one, EPT02, mapping  $\text{GPA}_{L_2}$  to HPA. Therefore,  $L_0$  can run  $L_2$  using  $L_2$ 's GPT2 and EPT02. The EPT-on-EPT approach allows  $L_2$  to freely update GPT2 without any traps to  $L_0$  (①-③), but needs multiple world switches to build and update EPT02 (①-⑬).

The update of EPT02 also involves two phases: The first phase updates EPT12 and EPT01 (①-⑩). The update of EPT12 is emulated by  $L_0$  – by making EPT12 *read-only* to  $L_1$ . Hence, step ⑤-⑦ may repeat multiple times – depending on the number of page table levels in EPT12. Thus, the first phase traps to  $L_0$   $n + 2$  times with  $2n + 4$  world switches (where  $n$  is the number of EPT12 page table levels). The second phase updates EPT02 (⑪-⑬), by compressing EPT01 and EPT12 (built in the first phase). It adds *one* more  $L_0$  trap and *two* more world switches. In summary, a page fault from  $L_2$  triggers  $2n + 6$  world switches and exit to  $L_0$  for  $n + 3$  times.

Compared to SPT-on-EPT, EPT-on-EPT is more efficient in terms of fewer number of world switches and  $L_0$  traps [13]. The performance benefit of EPT-on-EPT can be observed from Figure 4. We used a memory-intensive micro-benchmark running in  $L_2$  to sequentially allocate 1 MB memory spaces and access pages within the allocated regions one by one with a total working set size of 4 GB. We varied the number of benchmark instances from 1 to 16. Figure 4 shows that EPT-on-EPT significantly outperformed SPT-on-EPT in all cases. However, a considerable performance gap persisted between EPT-on-EPT and single-level memory virtualization (EPT only), and this gap widened as concurrency levels increased. We quantified the cost of world switches in EPT-on-EPT and EPT only. For fair comparison, we measured the time needed to perform an  $L_2$ -to- $L_1$  world switch and that due to an  $L_1$ -to- $L_0$  switch in single-level virtualization. While both are world switches between a VM and its immediate

hypervisor, the  $L_2$ -to- $L_1$  switch involves exits to  $L_0$  to handle state transition or fault injection. Our measurements showed that world switches in nested virtualization (1.3  $\mu$ s) is an order of magnitude more expensive than those in single-level virtualization (0.105  $\mu$ s).

### 2.3 Challenges and opportunities

While EPT-on-EPT is the state-of-the-art nested memory virtualization approach, it has several drawbacks that prevent it from being adopted in production cloud systems.

First, EPT-on-EPT still leads to a large number of costly world switches between various levels of hypervisors and VMs. For example, given a 4-level page table, a page fault from  $L_2$  can trigger up to 14 world switches and trapping to  $L_0$  7 times. The high rate of world switches inevitably leads to high performance overhead in nested virtualization. Second, nested virtualization upon EPT-on-EPT needs to extend hardware virtualization capabilities to  $L_1$  VMs. This leads to increased complexity and decreased flexibility of the cloud stack. Once an  $L_2$  guest is running,  $L_1$  can no longer be migrated, saved, or loaded, significantly impacting the cluster management. Further, many cloud providers do not support nested virtualization for cloud VM instances [31], or have many limitations in supporting nested virtualization [1, 54]. For example, nested virtualization is incompatible with emerging technologies [30] such as AMD SEV [2] and Intel TDX [5]. Last, nested virtualization upon EPT-on-EPT heavily relies on the  $L_0$  hypervisor (e.g., maintaining EPT02 and forwarding traps between  $L_2$  and  $L_1$ ). A “fat” host hypervisor increases security risks by expanding the attack surface for cloud providers.

These drawbacks of nested virtualization stem from its original design goal of supporting unmodified guest hypervisors ( $L_1$ ) [19, 31]. To achieve this, the host hypervisor ( $L_0$ ) uses a combination of hardware (e.g., EPT and VMCS) and software approaches (EPT-on-EPT and shadowing VMCS) to emulate VMX. However, the goal of using nested virtualization for secure containers is entirely different. It simply needs a strong-yet-lightweight isolation mechanism to make secure containers “deployable” in any existing IaaS clouds. This motivates us to develop a more efficient nested virtualization framework, which assumes no support from hardware virtualization and supports unmodified  $L_0$  host hypervisor.

## 3 PVM: Design and Implementation

We propose, PVM, a *high-performance, pagetable-based* nested virtualization framework built upon the KVM hypervisor. The over-arching goals of PVM are to completely decouple secure container hosting from the host hypervisor and hardware virtualization support to 1) enable nested virtualization with any IaaS clouds without affecting the security, flexibility, and complexity of the cloud platform; 2) avoid costly exits to the host hypervisor and devise efficient world switching mechanisms. To achieve this, PVM is developed based on the following design choices:

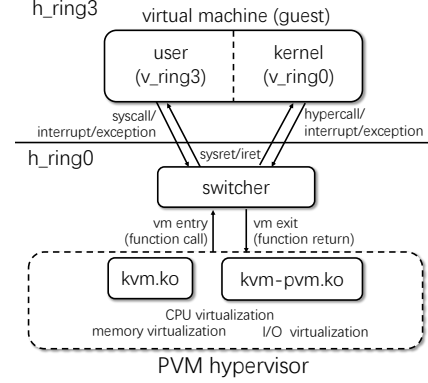


Figure 5. The PVM architecture.

1) Unlike the existing nested virtualization approaches that place  $L_2$  on privilege 3 (for the guest user) and privilege 0 (for the guest kernel) in the non-root mode, PVM’s  $L_1$  *de-privileges*  $L_2$  to completely operate in the least privilege level 3, or Ring 3, while providing isolation between the guest user and kernel through the use of separate page tables. This way, PVM effectively isolates secure containers while allowing for more efficient world switching (Section 3).

2) De-privileging  $L_2$  to Ring3 further facilitates PVM to enable  $L_1$  to capture all privileged operations from  $L_2$  and serve them (via emulation) without involving  $L_0$ , thus reducing the number of world switches. PVM further achieves *fast world switches* between  $L_1$  and  $L_2$  via a switcher, a piece of highly-efficient assembly code/data (Section 3.2).

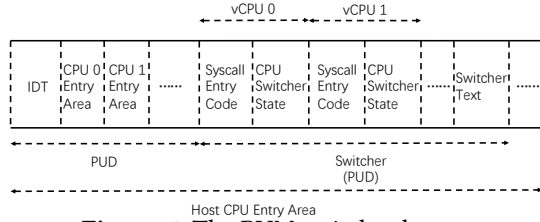
3) Upon fast world switches, PVM devises a *highly-efficient shadow paging* approach with a range of optimizations, leading to fewer and faster world switches than hardware-assisted approaches, such as EPT-on-EPT (Section 3.3).

The primary contribution of PVM is a software-based nested virtualization approach that operates independently of the host ( $L_0$ ) hypervisor and assumes no hardware support. PVM can co-exist with other ordinary VMs on the same host and relies on hardware virtualization support for VM-exit/entry between  $L_1$  and  $L_0$ . If not otherwise stated, the following discussions on  $L_2$ -to- $L_1$  switches are entirely handled by PVM without any involvement of the  $L_0$  hypervisor.

### 3.1 Architecture Overview

Figure 5 depicts the architecture of PVM, which comprises three key components: the  $L_2$  guest, the switcher, and the  $L_1$  guest hypervisor, or the PVM hypervisor. 1) The  $L_2$  guest operates entirely within the hardware Ring 3 (i.e.,  $h\_ring3$  in the *non-root* mode) with the least privilege. It features secure containers in virtual Ring 3 (i.e.,  $v\_ring3$ ) and the para-virtualized  $L_2$  guest kernel in virtual Ring 0 (i.e.,  $v\_ring0$ ). To distinguish between  $v\_ring0$  and  $v\_ring3$ , both residing at  $h\_ring3$ , PVM uses *separate page tables* for isolating the  $L_2$  guest user and kernel. 2) The switcher is responsible for “switching” the worlds between the  $L_1$  guest hypervisor, the  $L_2$  guest user, and the  $L_2$  guest kernel. The world switches can be triggered by a set of events, such as system calls,





**Figure 6.** The PVM switcher layout.

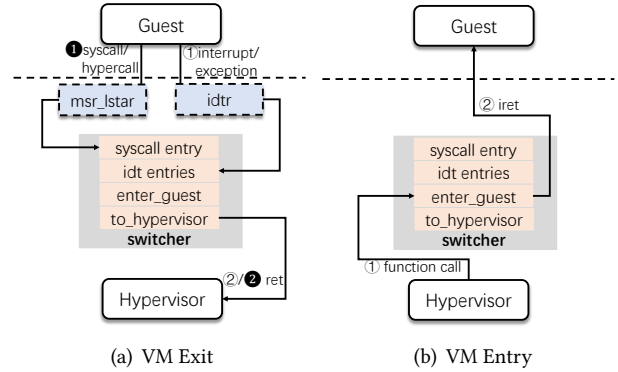
hypercalls, and interrupts/exceptions. The Switcher comprises efficient and concise assembly code and data structures that map to identical addresses within the  $L_1$  hypervisor,  $L_2$  user, and  $L_2$  kernel address spaces. 3) Finally, the PVM hypervisor comprises two (loadable) kernel modules: *kvm.ko*, which incorporates traditional KVM core functionality to ensure compatibility with the upper software stack, and *kvm-pvm.ko*, which incorporates the core PVM functionality, i.e., a customized virtualization implementation using a function pointer vector of KVM. Note that PVM can support both nested virtualization and bare-metal machines (i.e., running as an  $L_0$  host hypervisor). The discussion here and hereafter focuses on a 2-level nested virtualization scenario, where secure containers are deployed in  $L_2$  guests managed by the PVM hypervisor running in an  $L_1$  VM.

### 3.2 Switcher

Early 32-bit paravirtualized guests in Xen place the guest user and kernel in Ring 3 and Ring 1, respectively, to trap access to privileged states and use segmentation to isolate the guest and host space. In contrast, recent paravirtualized Xen guests in 64-bit x86-64 place guest user and kernel in Ring 3 and use separate page tables for isolation [43]. This design is necessary due to the possible removal of Ring 1 in the upcoming x86-s architecture [9] and the limited support of segmentation in x86-64. PVM draws inspiration from these paravirtualization approaches, such as Xen and Lguest [7], and adopts a switcher design to enable efficient world switches between the  $L_2$  user and kernel (Ring 3), and the  $L_1$  hypervisor (Ring 0) in the non-root mode.

As shown in Figure 5, switcher is located between the  $L_1$  hypervisor and the  $L_2$  guest (user and kernel) and facilitates state transitions between them. To be executed across domains during the switchover process (i.e., involving switching page tables and address spaces), switcher must be located at *identical* virtual addresses in the  $L_2$  user,  $L_2$  kernel, and  $L_1$  guest hypervisor. Previous methods [7] place a switcher in a high address space (e.g.,  $0xFFC00000/0xFFE00000$ ); both the user and kernel can access it using the *same* page table.

On the other hand, PVM adopts a more secure approach by utilizing the Linux Kernel Page Table Isolation (KPTI) design, which helps to mitigate certain types of security vulnerabilities, such as those exploited by the Meltdown and



**Figure 7.** VM Exit and Entry.

Spectre attacks. Similar to KPTI, PVM leverages separate page tables for the  $L_1$  host kernel (i.e., the guest hypervisor) and the  $L_2$  guest and consolidates the switcher’s code/data (e.g., syscall entry, IDT, TSS, trampoline stack, LDT, etc.) accessed by both the  $L_1$  hypervisor and the  $L_2$  guest in a *per-CPU entry area*. Note that, PVM already uses two separate page tables for the  $L_2$  guest user and kernel. Therefore, the switcher, or *the per-CPU entry area*, needs to be mapped to three page tables for the  $L_1$  host kernel,  $L_2$  guest user, and  $L_2$  guest kernel, respectively at an identical, unused virtual address range.

**Switcher layout.** As illustrated in Figure 6, there are three key elements in switcher: 1) a per-CPU syscall entry to handle syscall requests between the  $L_1$  guest hypervisor, the  $L_2$  guest user, and the  $L_2$  kernel; 2) a per-CPU switcher state, similar to VMCS, which saves and restores  $L_2$  guest and  $L_1$  host states during world switches; and 3) customized interrupt descriptor table (IDT) handlers for capturing interrupts/exceptions: PVM modifies the entries of the IDT in the  $L_2$  guest (both user and kernel) address space to point to the switcher’s customized interrupt handlers. This allows switcher to capture all external interrupts or exceptions, even during world switches. To ensure efficient accesses to and the execution of switcher, PVM sets the entire switcher to be global, preventing its TLB entries from being flushed.

**VM exit/entry:** Without hardware support, an  $L_2$  guest traps to the  $L_1$  hypervisor in two ways: syscall/hypercall and interrupt/exception. In PVM, hypercalls in the  $L_2$  guest kernel are implemented through the invocation of syscalls (i.e., via unique hypercall numbers).

As illustrated in Figure 7(a), when an  $L_2$  guest executes a syscall instruction or an interrupt/exception occurs, the CPU transitions to  $h\_ring0$  and enters the switcher through a pre-defined handler in CPU registers (e.g., MSR\_LSTAR or IDTR). Inside the switcher, the *to\_hypervisor* function performs the world switch from the  $L_2$  guest user (or kernel) to the  $L_1$  hypervisor. The switch function saves the  $L_2$  guest states and restores the  $L_1$  host states to/from the per-CPU switcher state (shown in Figure 6). Once the exit has been completed (e.g., via emulation as discussed in Section 3.3.1),

The function pointer vector, or the vCPU run structure, is a data structure that contains function pointers to various KVM operations, such as context switching, interrupt handling, and device emulation.

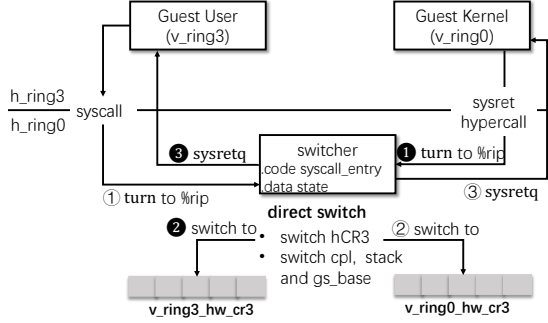


Figure 8. The direct switching within switcher.

the `enter_guest` function in the  $L_1$  hypervisor rebuilds the  $L_2$  guest context by saving the  $L_1$  host states and restoring the  $L_2$  guest states to/from the per-CPU switcher state, as shown in Figure 7(b). This way, the CPU switches from the  $L_1$  host ( $h\_ring0$ ) to the  $L_2$  guest ( $h\_ring3$ ). Additionally, to mitigate the security risk that arises due to mapping switcher into different address spaces, PVM clears all general purpose registers except for RSP and RAX during a VM-exit to prevent speculative use of a guest’s CPU states. This prevents an  $L_2$  guest from accessing saved CPU states of another guest or the guest hypervisor.

**Direct switch:** PVM has also devised a novel direct switch mechanism in the switcher, as shown in Figure 8. This direct-switch mechanism allows  $L_2$  guests to achieve fast user/kernel switches due to syscalls without hypervisor intervention.

Specifically, when an  $L_2$  guest user invokes a syscall from  $h\_ring3$ , the CPU transitions to  $h\_ring0$  and enters switcher. Switcher emulates the syscall instruction by 1) saving the state of the  $L_2$  guest user; 2) restoring the state of the  $L_2$  guest kernel; and 3) constructing a syscall frame where the  $L_2$  guest kernel can invoke the requested syscall function. At the end of the syscall emulation, switcher directly switches to the  $L_2$  guest kernel for executing the syscall. By default, the kernel returns a syscall via the `sysret` instruction. As `sysret` is a privileged instruction while the  $L_2$  guest kernel is running in  $h\_ring3$ , the execution of `sysret` by the  $L_2$  guest kernel will trap to  $h\_ring0$ . To enable direct switch, PVM uses a `sysret` hypercall in the  $L_2$  guest kernel to return a syscall. The `sysret` hypercall will enter the switcher (no traps to the  $L_1$  hypervisor), which switches the execution worlds and directly returns the  $L_2$  guest user.

### 3.3 PVM Hypervisor

**3.3.1 CPU virtualization.** PVM relies entirely on the software emulation for CPU virtualization. PVM virtualizes a vCPU by maintaining its state in the vCPU data structure (similar to VMCS) stored in the per-CPU entry area (see Figure 6) in switcher. As stated in Section 3,  $L_2$  guest vCPUs are restricted to operate solely on  $h\_ring3$  for security considerations. The PVM hypervisor simulates  $v\_ring0$  and  $v\_ring3$  using a switcher state flag for the  $L_2$  guest. As  $L_2$  guest vCPU works on  $h\_ring3$ , access to privileged registers

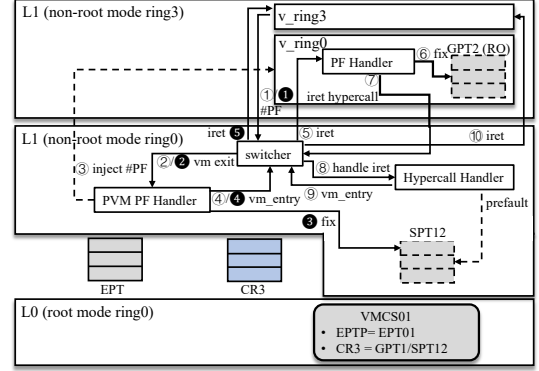


Figure 9. PVM-on-EPT.

and segment states, and the execution of privileged CPU instructions must be intercepted by the PVM hypervisor. PVM adopts a hybrid approach based on instruction emulation and paravirtualization for CPU virtualization. It currently focuses on supporting the 64-bit mode with paging for  $L_2$  guest vCPUs, as this is the most frequently used mode.

Specifically, when an  $L_2$  vCPU executes a privileged instruction, it leads to a general *protection exception* and triggers an exit to the  $L_1$  hypervisor. The PVM hypervisor employs an instruction simulator to emulate instruction execution for the  $L_2$  guest. As the process of *trap and emulation* is expensive, PVM uses hypercalls to handle a total of 22 frequently invoked privileged instructions (e.g., `iret`, `write_msr`, and `read_msr`). Non-privileged but sensitive instructions also need to be restricted because of their capability to modify/observe privileged host states. PVM uses the Linux paravirtualization (PV) mechanism – initially proposed by Xen [18] – to manage both sensitive and privileged instructions via `pv_cpu_ops`, `pv_mmu_ops`, and `pv_irq_ops` interfaces. These interfaces enable PVM to capture and handle any sensitive instructions even though x86 is not fully virtualizable [42].

**3.3.2 Memory virtualization.** PVM seeks to handle memory virtualization of an  $L_2$  guest exclusively by and within the  $L_1$  hypervisor and employs a software-based approach, i.e., shadow page tables (SPT). As stated in Section 3, PVM uses separate page tables for the  $L_2$  guest user and  $L_2$  kernel, preventing any access from the guest user to the guest kernel address. Accordingly, PVM constructs separate shadow page tables for the  $L_2$  guest kernel and user, simulating KPTI for  $L_2$  guests at the  $L_1$  hypervisor level, achieving strong isolation.

However, as discussed in Section 2.2, both SPT (single memory virtualization) and SPT-on-EPT (2-level nested memory virtualization) incur significant poor overhead due to a large number of expensive world switches. To address this challenge, PVM devises a new and highly-efficient SPT approach with various optimizations, called *PVM-on-EPT*, as illustrated in Figure 9. Compared with the EPT-on-EPT approach (Section 2.2), PVM-on-EPT leads to *fewer* and *cheaper* world switches, making it possible for a software-based SPT



approach to achieve comparable (or better) performance as (than) the hardware-assisted approach.

**PVM-on-EPT.** Figure 9 shows that, the  $L_2$  guest operates its guest page table (GPT2), mapping  $GVA_{L_2}$  to  $GPA_{L_2}$ , while the  $L_1$  hypervisor maintains the guest shadow page table (SPT12) for  $L_2$ , mapping  $GVA_{L_2}$  to  $GPA_{L_1}$ . Note that,  $L_1$  maintains two shadow page tables – one for the  $L_2$  guest user and one for the  $L_2$  guest kernel. To ensure that any updates from GPT2 are synchronized with SPT12, GPT2 is made *read-only* to  $L_2$ . Finally, to run the  $L_2$  guest, the underlying host  $L_0$  hypervisor simply uses both SPT12 and EPT01 (which maps  $GPA_{L_1}$  to HPA) to translate  $GVA_{L_2}$  to HPA. Since EPT01 for an  $L_1$  VM appears identical to that of other regular VMs, the  $L_0$  hypervisor remains unaware of the nested memory virtualization within  $L_1$ , and thus, requires no modifications to support PVM-on-EPT.

We use the handling process of a page fault from  $L_2$  to demonstrate the world switches due to PVM-on-EPT. First, an  $L_2$  page fault occurring from the guest user space traps to the  $L_1$  hypervisor via the switcher (①-②) with *one* world switch. The page fault handler of the PVM hypervisor injects the page fault back to  $L_2$  (③) and switches to the  $L_2$  kernel via *one* world switch (④-⑤). Then, the  $L_2$  kernel starts updating GPT2 (⑥). As GPT2 is read-only, each update needs the assistance of  $L_1$ , causing  $2n$  (where  $n$  is the number of GPT2 levels) world switches between the  $L_2$  kernel and  $L_1$  (not depicted in Figure 9). After all related page table entries in GPT2 have been updated, the  $L_2$  guest kernel returns (⑦ via the *iret* hypercall). Instead of returning to the  $L_2$  guest user via direct switch, PVM adopts a *prefault* optimization and switches to  $L_1$  (with *one* world switch), which proactively updates SPT12 (⑧), preventing future traps due to subsequent page faults on SPT12 due to the missing of the corresponding  $GVA_{L_2}$  recently updated in GPT2. Finally,  $L_1$  returns to  $L_2$  guest user with *one* more world switch. In summary, an  $L_2$  page fault causes  $2n + 4$  world switches under PVM-on-EPT, fewer than EPT-on-EPT which triggers  $2n + 6$  world switches. Note that, PVM-on-EPT exclusively handles of  $L_2$  page faults within  $L_1$ , while EPT-on-EPT heavily relies on  $L_0$ . Although PVM-on-EPT only saves two world switches compared to EPT-on-EPT, the world switches in PVM-on-EPT are substantially cheaper. The world switches in PVM entail state transitions within switcher and ring level changes in the non-root mode while those in EPT-on-EPT require state transitions across the non-root and root mode. Our measurements show that it takes on average  $0.179 \mu s$  to perform a world switch in PVM, almost an order of magnitude cheaper than that in EPT-on-EPT ( $1.3 \mu s$ ) and close to that in single-level virtualization ( $0.105 \mu s$ ).

**Optimizations.** In addition to the *prefault* optimization, we devise two other important optimizations in PVM-on-EPT: 1) a process context identifiers (PCID) mapping mechanism that eliminates TLB flushes for  $L_2$ ; and 2) a fine-grained locking mechanism for fast SPT updates.

In a traditional SPT approach, the hardware uses hierarchical address space ID for TLB (e.g., EPTP, VPID, and PCID), and all processes running in the same  $L_2$  guest share the same higher granular VPID. This means that any TLB flushes to the  $L_2$  (user or kernel) cause the higher granular VPID to be flushed rather than the specific PCID, resulting in a costly cold-start penalty as all TLB entries associated with the  $L_2$  guest are flushed whenever there is a  $L_2$  flush request. To eliminate these TLB flushes for  $L_2$ , PVM employs a PCID mapping mechanism. It assigns unused PCID values (e.g., 32-63) from  $L_1$  to the  $L_2$  guest (e.g., 32-47 for guest  $v\_ring0$  and 48-63 for guest  $v\_ring3$ ) and maps them with the  $L_2$  guest’s PCID. This allows the TLB hardware to recognize individual SPTs for each process in  $L_2$ , thereby eliminating TLB flushes during world switches.

In addition, a traditional SPT approach relies on a global read-write lock, namely “*mmu\_lock*”, to serialize updates to SPT data structures. In contrast, PVM employs a fine-grained locking mechanism that provides more precise data protection and enables greater concurrency in handling page faults. First, PVM identifies tasks that can be processed without holding the “*mmu\_lock*” or can be delayed (such as walking shadow pagetables), thereby reducing the sequential execution time caused by “*mmu\_lock*”. Further, PVM categorizes SPT related data into three sub-groups: 1) inter-shadow pages (e.g., shadow page collections or parent/child relationships), 2) intra-shadow pages (e.g., shadow page-table entries), and 3) reverse mappings (between a guest frame number and its SPT entry) allowing for faster location of the SPT entries given a guest frame number. PVM uses separate locks to protect each data group – i.e., a “meta-lock” for inter-shadow pages, a per-shadow page “*pt\_lock*” for intra-shadow pages, and a per-guest frame number, per-page “*rmap\_lock*” for reverse mappings.

**3.3.3 Interrupt virtualization.** Unlike CPU and memory virtualization that can be entirely handled by PVM within  $L_1$ , interrupt virtualization requires the involvement of the  $L_0$  hypervisor. Whenever an interrupt for an  $L_2$  guest occurs, regardless of KVM or PVM, an  $L_2$  guest will first need to exit to the  $L_0$  hypervisor because in both scenarios the interrupt will cause a VM exit from the non-root mode to the root mode at  $L_0$ . This first world switch is enabled by VMCS and handled (automatically and transparently to software) by the CPU hardware. Subsequently, the  $L_0$  hypervisor injects the interrupt to the  $L_1$  VM, from where PVM handles the interrupt differently from the existing approaches. In KVM, since the  $L_1$  VM is unable to manage the VMCS for the  $L_2$  guest and the  $L_0$  hypervisor manages a shadow copy  $VMCS_{02}$  for  $L_2$  (as discussed in Section 2.1), the following interrupt handling requires multiple exits to  $L_0$ . In contrast, after interrupt injection to the  $L_1$  VM, PVM uses a customized IDT to handle the interrupt entirely between  $L_1$  and  $L_2$  without any involvement of  $L_0$ .

PVM relies on  $L_0$  solely for interrupt injection to  $L_1$  and adopts software-based interrupt virtualization afterwards. If an interrupt occurs while an  $L_2$  guest is running, the default interrupt descriptor table (IDT) handler for this  $L_2$  guest, which is pointed by the hardware IDTR register located in the  $L_2$  guest address space, cannot handle it appropriately. To address this issue, PVM maps a customized IDT at the address to where the IDTR points in the  $L_2$  address space. The customized IDT initiates a transition to the PVM hypervisor (equivalent to a VM exit from  $L_2$  to  $L_1$ ) to handle the interrupt. Note that as PVM shifts the starting address of the guest’s CPU entry area back by one PUD size, both the customized IDT and the  $L_2$  guest’s original IDT can co-exist. Within the  $L_1$  VM, PVM reuses the interrupt controller (APIC) virtualization in KVM to convert the interrupt to a virtual interrupt and injects it back to the  $L_2$  guest, from where the virtual interrupt handle is identical to that in a regular VM.

To capture all interrupts for an  $L_2$  guest, the PVM hypervisor enables hardware interrupts at `h_ring3`, where the  $L_2$  guest operates. This requires setting the `RFLAGS.IF` flag in the `iret` frame for each VM entry in `switcher`. However, read/write to the interrupt configuration from within an  $L_2$  guest does not cause an exit to the  $L_1$  hypervisor and thus the latter does not know whether a virtual interrupt can be injected into an  $L_2$  guest, i.e., whether the interrupt is enabled in  $L_2$ . To overcome this, PVM introduces an 8-byte shared data structure between an  $L_2$  guest and the  $L_1$  hypervisor to virtualize `RFLAGS.IF`. As such, the  $L_1$  hypervisor is able to directly query interrupt configuration inside an  $L_2$  guest.

## 4 Evaluation

We have thoroughly evaluated the effectiveness of PVM. As PVM does not require hardware virtualization support and is compatible with KVM’s virtualization interface, it can run both as 1) an  $L_1$  guest hypervisor for nested virtualization and 2) an  $L_0$  hypervisor on bare-metal hardware for single-level virtualization. We compared PVM with two existing approaches: hardware-assisted and shadow page table-based (software) approaches in both single-level (bare-metal) and nested (2-level) virtualization. Specifically, we first used micro-benchmarks to investigate the performance of PVM’s `switcher` in performing world switches and PVM’s shadow paging (i.e., PVM-on-EPT) in handling various types of guest page table updates. Further, we used real-world applications to assess the overall performance and scalability of PVM.

Without loss of generality, we conducted the experiments using Intel processors. We used two Intel x86 instances from Alibaba Cloud – 1) a bare-metal instance and 2) a general-purpose (VM) instance with identical software and hardware configurations. Each instance was configured with two Intel Xeon Platinum CPUs (i.e., model 8269CY with 26 cores at 2.50GHz and hyperthreading enabled), 385 GB RAM, and a 500 GB SATA SSD. The bare-metal instance was equipped with Intel virtualization technology (VT), supporting VMX

Configurations	kvm (BM)	pvm (BM)	kvm (NST)	pvm (NST)
Hypercall	0.46/0.46	0.54/0.54	7.43/7.87	0.48/0.48
Exception	1.66/1.65	1.67/1.65	9.20/9.01	2.21/2.2
MSR access	0.87/0.87	2.53/2.51	8.18/8.47	2.88/2.86
CPUID	0.54/0.54	0.60/0.59	7.10/7.16	0.51/0.51
PIO	3.79/3.39	4.91/4.54	29.34/28.27	12.94/12.03

**Table 1.** Average round-trip latency ( $\mu$ s) of VM exits/entries with KPTI enabled/disabled.

root and non-root modes, EPT, and VMCS Shadowing. The VM instance had no hardware virtualization support and did not colocate with other VMs. Hence, its performance was solely affected by nested virtualization.

We ran benchmarks and workloads using Kata containers, namely the *secure containers* where regular containers are deployed and running in lightweight VMs, controlled by KVM or PVM. A secure container can run as an  $L_2$  guest on the general-purpose VM instance or directly on the bare-metal instance. All secure containers were managed by the RunD [36] runtime. Both the  $L_1$  guest hypervisor and  $L_0$  host hypervisor ran Linux kernel 4.19. For I/O virtualization, we used paravirtualization: `virtio-blk` for disk and `vhost-net` for network. All Linux kernels had Kernel Page Table Isolation (KPTI) enabled, if not otherwise stated.

We considered five scenarios for the deployment of secure containers: 1) Bare-metal (BM) instance with full hardware virtualization support, denoted as *kvm-ept (BM)*. This is the single-level virtualization case with the hardware VMX and EPT support. 2) Bare-metal instance with shadow paging (*kvm-spt (BM)*). This is the baseline for software-based memory virtualization. 3) PVM on bare-metal hardware (*pvm (BM)*). 4) 2-level nested (NST) virtualization with full hardware support (*kvm-ept (NST)*). This is a state-of-the-art approach for deploying secure containers in VM instances. 5) PVM within VM instances with nested virtualization (*pvm (NST)*). Note that we exclude the shadow-on-shadow (SPT-on-SPT) scenario as it is not a viable approach in practical systems due to unacceptable performance.

### 4.1 Micro-benchmarks

**VM exit and entry** cause world switches between a VM and its underlying hypervisor(s). With VMX hardware support, VM exits directly trap to ring-0 in root mode, i.e., the  $L_0$  hypervisor in nested virtualization. In contrast, PVM uses hypercalls or instruction emulation to trap privileged instructions due to the absence of hardware support, but VM exits only trap to the  $L_1$  guest hypervisor, i.e., PVM (instead of the  $L_0$  hypervisor). We evaluated the effectiveness of PVM in handling VM exits compared to the hardware-assisted approach in single-level and nested virtualization. As both *kvm-spt* and *kvm-ept* use VMX for CPU virtualization, their results are similar and are collectively referred to as *kvm*.

We used a set of micro-benchmarks to perform privileged guest operations to trigger VM exits and measured the round-trip time until VM entries, both with KPTI enabled/disabled.

Configurations	Optimization	Syscall ( $\mu$ s)
kvm-ept (BM)		0.22/0.06
kvm-spt (BM)		2.09/0.06
pvm (BM)	none	1.91/1.91
	direct-switth	0.29/0.29
kvm (NST)		0.23/0.06
pvm (NST)	none	1.93/1.93
	direct-switth	0.3/0.3

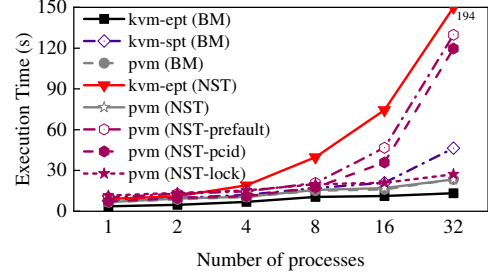
**Table 2.** Execution time ( $\mu$ s) of syscall `get_pid` with KPTI enabled/disabled.

Table 1 lists the average round-trip latency under five privileged guest operations: no-op hypercall, invalid opcode exception, MSR access of `MSR_CORE_PERF_GLOBAL_CTRL`, `CPUID`, port-mapped I/O (PIO). For each case, the average round-trip latency was determined based on ten million operations.

In *single-level* virtualization, VM exits/entries were efficiently handled by hardware: *pvm (BM)* offered performance comparable to *kvm (BM)* in most guest operations, except for MSR access. It is due to KVM’s capability to directly access MSR registers in non-root mode, avoiding VM exits. In *nested* virtualization, PVM significantly outperformed hardware-assisted approaches: *pvm (NST)*, compared to *kvm (NST)*, reduced VM exit/entry latency by an average of over 75%. This clearly demonstrates the benefit of PVM’s software-based VM exit/entry: *kvm (NST)* triggered two expensive exits to the  $L_0$  hypervisor for each privileged  $L_2$  guest operation, while PVM needed *one and less expensive* VM exit to the  $L_1$  guest hypervisor. From these results, we did not discern any considerable impact from KPTI on VM exits/entries.

**System calls**, or syscalls, cause the transition between guest user and kernel space. Unlike other privileged operations, most syscalls can be handled without VM exits in hardware-assisted, nested virtualization – i.e., no exits to the  $L_1$  or  $L_0$  hypervisor. In contrast, without the VMX hardware support, PVM emulates the syscall instruction through its *switcher*, where a guest syscall causes an exit to the PVM hypervisor.

To gauge PVM’s efficiency in handling guest syscalls, we measured the execution time of the `get_pid` syscall. This syscall is simple and mostly involves a user/kernel transition. In Table 2, regardless of single-level or nested virtualization, *kvm-ept (BM)* and *kvm (NST)* achieved similar and superior performance because the syscall can be handled exclusively and quickly by the guest. We then analyzed two PVM variants: one with direct switching and the other without. Compared with hardware-assisted approaches, PVM noticeably slowed down syscall execution – e.g., by up to 7x without direct switching (with KPTI enabled). The direct switching feature can reduce PVM’s overhead, narrowing the syscall execution time gap between PVM and its hardware-assisted counterparts to around 1.3x (with KPTI enabled). *kvm-spt* was the least efficient of all tested methods because a user-kernel transition in the guest requires a costly trap to the hypervisor to switch the corresponding shadow page tables.



**Figure 10.** The performance of guest page faults handling.

We observed that with KPTI disabled, both *kvm-ept (BM)* and *kvm (NST)* executed the `get_pid` syscall faster due to reduced world switches between the guest user and kernel, e.g., the syscall latency dropped from 0.22 to 0.06 microseconds. However, this improvement was not observed in PVM, e.g., the latency remained around 0.30 microseconds (with direct switching). The main reason is that disabling KPTI in PVM does not reduce its *world switches*: After a system call completes via `sysret`, there remains the necessity to exit either to switcher or to the PVM hypervisor. Both change the ring level from `h_ring3` to `h_ring0` in non-root mode. We are in the process of developing an advanced direct switching optimization that can return `sysret` at `h_ring3`. This can save one exit to `h_ring0` during syscalls and achieve comparable syscall latency as that in the KVM baselines without KPTI.

**Page faults.** In nested virtualization, PVM is transparent to the  $L_0$  hypervisor. Consequently, any violations in EPT01 – i.e., responsible for translating  $L_1$  VM’s physical addresses to  $L_0$  host’s physical addresses – are handled by the  $L_0$  hypervisor (out of PVM’s control). Thus, we assume that the  $L_1$  VM has been sufficiently warmed up and there are very few EPT violations. This is a reasonable assumption as general-purpose VM instances usually stay up for hours, if not for days. We focus on evaluating guest page faults, i.e., those due to launching short-lived secure containers.

We used a micro-benchmark to repeatedly allocate and release 1MB of memory in a guest’s virtual address space and access the allocated data at page granularity. This process continued until the accessed data reached 4GB. This benchmark caused frequent updates to the guest page table and thus stress-tested memory virtualization. To examine scalability, we adjusted the number of benchmark processes from 1 to 32 within a single container. Figure 10 illustrates the performance of various approaches, among which *kvm-ept (BM)* achieved the best performance and scalability. This is because the existing 2-level paging hardware can efficiently handle guest page table updates. While *pvm (BM)* shows similar scalability, its performance lags. This is expected for a software-based approach. Notably, *pvm (NST)* significantly outperformed *kvm-ept (NST)*, and the performance gap widened with increasing concurrency. This clearly demonstrates the benefits of PVM’s shadow paging (i.e., PVM-on-EPT) and its optimizations – prefault, PCID mapping, and fine-grained page table locking. As depicted in Figure 10, by applying

Config	#C	null I/O	stat	open close	slct TCP	sig inst	sig hndl	fork proc	exec proc	sh proc
kvm-ept (BM)	1	0.27	0.72	25.07	2.16	0.29	1.01	82.13	422.33	1.6k
	32	6.5	14.6	289.5	2.16	0.3	4.005	513.5	4.62k	7.64k
kvm-spt (BM)	1	2.15	2.66	32.63	4.09	2.16	2.95	504.67	1.5k	4.2k
	32	6.325	14.2	266	4.115	2.17	3.995	32k	76k	171k
pvm (BM)	1	0.33	0.78	30.83	2.18	0.32	1.71	436.67	1.29k	3.58k
	32	6.475	14.95	314	2.18	0.33	4.25	5.38k	12.5k	29.5k
kvm-ept (NST)	1	0.28	0.76	36.43	2.24	0.30	1.07	113.00	725.67	2.29k
	32	7.03	16	578.5	2.495	0.35	4.295	812.5	14k	28k
pvm (NST)	1	0.33	0.79	25.73	2.22	0.33	1.72	466.33	1.33k	3.55k
	32	5.96	10.45	335.5	2.25	0.33	4.52	7.04k	15k	32.5k

**Table 3.** LMBench: Processes - time in  $\mu$ s (smaller is better).

Config	0K File create/delete	10K File create/delete	Mmap	Prot Fault	Page Fault	100fd select
kvm-ept(BM)	86.83/55.37	139.37/59.43	85.93k	0.66	0.15	2.02
kvm-spt(BM)	115.97/73.10	190.77/80.57	125.9k	3.17	0.93	3.96
pvm(BM)	112.83/73.83	172.30/73.57	115.20k	2.65	0.88	2.05
kvm-ept(NST)	148.73/98.87	221.63/98.77	128.03k	0.69	0.19	2.09
pvm(NST)	108.03/69.33	161.20/68.50	118.20k	2.69	1.01	2.08

**Table 4.** File & VM system latencies in  $\mu$ s (smaller is better).

the fine-grained page table locking optimization alone, PVM achieves superior scalability; the use of prefault and PCID mapping further enhances PVM’s performance.

## 4.2 System Benchmarks

Beyond our hand-crafted micro-benchmarks, we also used system benchmarks targeting different guest subsystems to more comprehensively evaluate PVM. We selected 32 benchmarks in LMBench [8], including process management, file systems, and network I/O. Table 3 and 4 present the results for process management and file system I/O, respectively. For process management, we present results with a single process and 32 processes. Networking results are similar to that of file systems and thus not presented.

Table 3 shows that *pvm (BM)* consistently outperformed *kvm-spt* in most cases and achieved close yet slightly worse performance than that of *kvm-ept (BM)* except for fork, exec, and sh. We found that these three benchmarks intensively created new page tables without actually accessing them. Therefore, these benchmarks caused guest page faults that never needed to update the  $L_0$  hypervisor’s EPT. In such cases, hardware-assisted approaches are always more efficient and page faults can be handled solely by the guest. A similar pattern was observed for nested virtualization – *pvm (NST)* was consistently better than *kvm-ept (NST)* except for the three tests for the same reason. Table 4 shows the performance of the file system I/O and virtual memory management. As PVM largely relies on KVM for I/O virtualization, its performance on file and network I/O was close to KVM, whether in single-level or nested virtualization. The only exceptions were observed in the two page fault benchmarks – similar to the fork benchmark, guest page faults occurred without updates in the hypervisor-managed page table. We

also performed tests on network latency and bandwidth and obtained similar results as those in the file system tests.

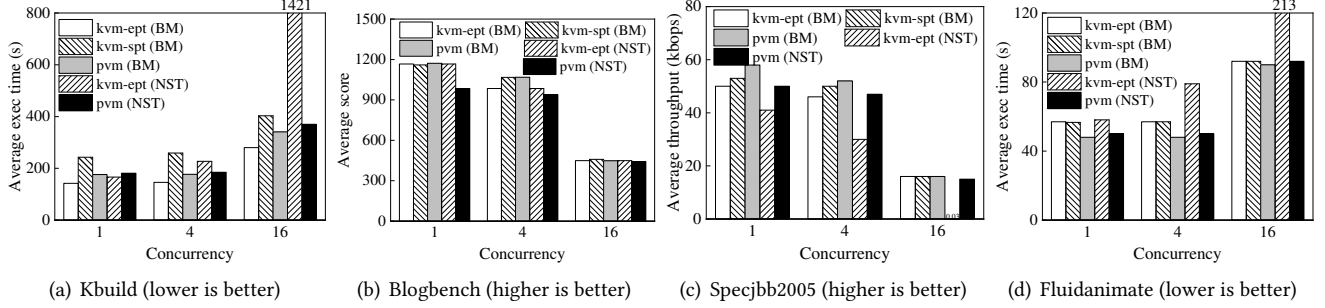
## 4.3 Real-world Applications

We continued the evaluation of PVM using four representative real-world applications with distinct characteristics: 1) *Kbuild* [26] builds the Linux kernel from the source involving a mixture of compute and file I/O; 2) *Blogbench* [16] is a filesystem benchmark that reproduces the load of a busy file server; 3) *SPECjbb2005* [14] is a Java benchmark involving the use of a Java virtual machine (JVM); 4) *Fluidanimate* is a benchmark with a large dataset selected from the PARSEC benchmark suite [21]. We ran multiple instances of the same benchmark, each in a separate secure container, and varied the concurrency level from 1 to 16.

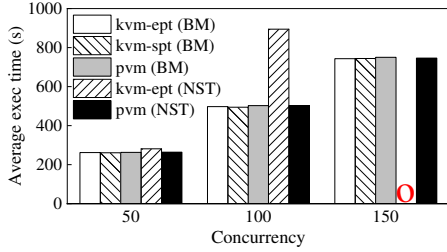
We made the following observations from the results as shown in Figure 11. First, for all applications, PVM offered performance close to the hardware-assisted approach for single-level virtualization. Further, the performance of *kvm-ept (NST)* collapsed in all cases when concurrency was high, suggesting that the  $L_0$  hypervisor became the bottleneck. This discovery reinforces that only utilizing hardware-assisted approaches to traverse multiple layers in the virtualization stack is neither efficient nor adaptable. In comparison, PVM achieved consistently good performance and, in many cases, close to that of single-level virtualization. PVM even outperformed the hardware-assisted approach in *fluidanimate* due to more efficient handling of the HALT instruction. PVM executes the HALT instruction via a hypercall and performs the sleep and wakeup process without a switch between the non-root and root modes, leading to higher overall CPU utilization for parallel programs with blocking synchronization.

PVM achieves superior concurrent performance due to two main factors. First, PVM incurs only one VM exit to  $L_0$  when handling an interrupt, eliminating the need for VM exits to  $L_0$  when managing  $L_2$  guest page faults. As a result, there is less strain on  $L_0$  by minimizing the frequent switching between non-root and root modes. Instead of overwhelming  $L_0$  with these tasks, PVM enables interrupts and  $L_2$  guest page faults to be individually and concurrently handled by their corresponding  $L_1$  hypervisors. Second, PVM devises  $L_0$ -transparent optimizations for page fault handlings, such as prefault, PCID mapping, and fine-grained shadow page table locks. These mechanisms together achieve up to two orders of magnitude performance improvements when compared to hardware-assisted virtualization with multiple  $L_2$  guests.

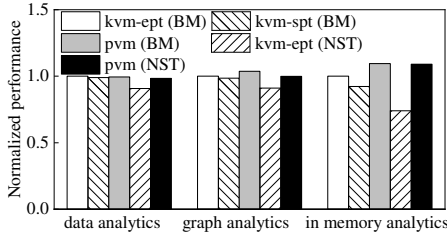
We further increased the density of secure container deployment to the maximum capacity the two cloud instances can handle. Figure 12 shows the performance of *fluidanimate* – the most memory-intensive one among the four applications. Interestingly, under high load conditions, all approaches converged to similar performance except for *kvm-ept (NST)*, which crashed due to a failure to connect to the RunD container runtime. Last, we evaluated PVM with



**Figure 11.** The performance of real-world applications under different levels of concurrency.



**Figure 12.** The performance of fluidanimate under high load.



**Figure 13.** The performance of cloud benchmarks.

three representative workloads with large datasets from the Cloud Bench Suite [4]. This experiment stress-tested PVM’s capability of executing data-intensive applications at a relatively low concurrency level. Figure 13 shows that PVM achieved close performance to bare-metal approaches and significantly outperformed *kvm-ept (NST)*.

To summarize, PVM is currently the most practical solution for *nested* virtualization across various workloads, delivering comparable performance to single virtualization.

#### 4.4 Cloud Adoption

PVM has been adopted by Alibaba Cloud, a major IaaS cloud provider, as an alternative to bare-metal instances for hosting secure containers. Presently, PVM operates more than 100K secure containers with over 400K vCPUs on a daily basis. These secure containers are deployed for a range of tasks, such as user-defined serverless functions, data analytics via Spark, and offline batch jobs. PVM has effectively supported users in addressing traffic spikes by promptly launching general-purpose instances with nested virtualization.

The adoption of PVM has continued to grow over the past year. It has led to a 36% user shift from bare-metal instances to general-purpose instances for secure container hosting and substantial cost-savings for cloud tenants. Notably, there has been a steady migration of Spark-based

in-memory data processing and analytics workloads from bare-metal servers to PVM servers with nested virtualization. These PVM servers have shown an average performance boost of 22.6% for Spark workloads. It is worth noting that the PVM servers are equipped with newer generation processors than the bare-metal ones in Alibaba Cloud. We expect PVM to offer comparable performance with bare-metal servers if both were deployed on the same platform.

## 5 Discussions and Future Work

**Security of PVM.** Secure containers running in PVM share the common isolation assumptions as traditional VMs: We trust hardware-based protection (e.g., page tables), which ensure strong isolation between processes, or “worlds” in PVM; we focus on deficiencies in software (e.g., kernels and hypervisors) that could be exploited through exposed interfaces like system calls or hypercalls. Hence, the threat model assumes that one malicious tenant (running in  $L_2$ ) may attempt to breach the isolation boundaries by compromising the exposed interfaces and exploiting vulnerabilities in its host kernels or hypervisors (running in  $L_1$ ), resulting in information leakage, privilege escalation, and denial of services. Two main metrics are used to evaluate the attack surface: 1) the size of the exposed interfaces and 2) the extent of code accessible through the interfaces.

Secure containers running with PVM provide stronger isolation (i.e., more secure) than traditional containers that are deployed directly on the ( $L_1$ ) host and share the same host kernel. There are two main reasons: 1) Narrow attack surface: secure containers interact with the ( $L_1$ ) host kernel via a minimal set of hypercalls, typically around 10s. In contrast, traditional containers rely on a much larger system-call interface, e.g., 250+ system calls under the default seccomp configuration. 2) Defense in depth: a malicious user in secure containers needs to comprise both the ( $L_2$ ) kernel and ( $L_1$ ) hypervisor before being able to comprise the ( $L_1$ ) host kernel, making it challenging for attackers to breach the isolation boundaries and escalate privileges. Moreover, PVM leads to a thin L0 hypervisor compared to hardware-based nested virtualization, which further reduces the security risks and minimizes the attack surface for IaaS clouds.



**Limitations of PVM.** As a software-based approach for nested virtualization, PVM could introduce considerable overhead for certain workloads, such as fork and the allocation of many small memory areas. These workloads lead to a high number of  $L_2$  guest page faults that the PVM hypervisor needs to handle. In contrast, a guest can handle these page faults exclusively in hardware-assisted approaches. Even though these access patterns are rare in real-world applications, we aim to address this issue by allowing PVM’s *switcher* to distinguish between guest and shadow page table faults. By doing this, the former can be directly injected back to an  $L_2$  guest, saving one exit to PVM.

PVM ensures isolation between the guest user and kernel by using separate page tables for each, resulting in dual shadow page tables. We are continuously refining PVM in two directions. First, write protection (WP) is currently employed by PVM to synchronize the guest and shadow page tables (GPT and SPT). However, WP-induced VM exits cause a significant overhead with dual shadow page tables. We are exploring the possibility of removing WP by allowing the guest and hypervisor to collaboratively construct these page tables through an efficient synchronization approach. Second, we are working towards implementing a Xen-like “direct paging” solution on KVM by mapping the GPA->HPA relationship to the guest.

## 6 Related Work

**Secure containers** [6, 10, 15] host containerized workloads in lightweight VMs. They can be crafted with both hardware support [3, 12, 46] and software optimizations [15, 51, 52], leading to minimized performance overhead. For instance, Intel CPUs support Virtualization Technology (VT-x) [3, 46] and Virtualization Technology for Direct I/O (VT-d) [12] to reduce the performance overhead of CPU, memory, and I/O virtualization. AWS’s Firecracker [15] tailors a VM’s guest kernel and virtual machine monitor (VMM) with minimal components and a simplified I/O model. NEC’s LightVM [51] links a hosted application into a tiny unikernel image under a single address space for improved efficiency. With such support/optimizations, secure containers allow applications to run efficiently with high-density deployment and high-concurrency startup [36]. PVM introduces a novel high-performance nested virtualization framework to run securing containers in VMs more efficiently.

**Nested virtualization.** There has been significant research in performance improvement via both software and hardware approaches. Turtles [20] and Neve [37] are two early implementations of nested virtualization on x86 and ARM. DVH [38] provides four mechanisms (e.g., virtual passthrough, virtual timers, virtual inter-processor interrupts, and virtual idle) enabling  $L_0$  to directly expose virtual hardware to  $L_2$  curtailing exit multiplication. SVT [48] runs different VMs and hypervisors on separate hardware threads, reducing the costly context switches of VM traps via simple thread stall. NestCloud [41] introduces three optimizations that include

guest page fault bypassing, virtual EPT, and PV VMCS, to cut down the overhead of nested guests. BrFusion [17] and Hostlo [17] address two networking issues of nested virtualization. nOSV [44] presents a lightweight VMM based on the microkernel that can house multiple Xen hypervisors with minimal runtime overhead. In contrast, PVM introduces a pure software-based design and implementation for nested virtualization with significantly improved performance.

**Software-based virtualization.** Without hardware support, the hypervisor must implement VM entry/exit and memory virtualization via software [7, 13, 24, 25, 29, 50]. By combining trap-and-emulate techniques with binary translation, VMware Workstation 1.0 [24] provides an efficient way to virtualize x86 architectures and enable the operation of multiple virtual machines on a single physical host. Lguest [7] employs shared memory to enable the mapping of code in the host and guest address spaces simultaneously, facilitating VM entry/exit. HVX [29] leverages instruction replacements and simulation to enhance virtualization. Xen-blanket [50] provides blanket hypercalls for guests, replacing the original vmcall-based hypercalls, thus avoiding the execution of privileged vmcall instructions. These approaches rely on the shadow paging mechanism for memory virtualization. However, recent widely adopted secure container technologies rely heavily on the KVM hypervisor, which can effectively utilize Linux subsystems by co-existing with the Linux kernel and providing support for many advanced cloud-native features (e.g., hotplugging, memory balloon, large pages, and virtio). PVM, as a practical approach for cloud-native environments, is built upon the KVM hypervisor and seamlessly integrates with the existing software stack of secure containers. In addition, PVM’s PVM-on-EPT significantly outperforms existing paging-based approaches.

## 7 Conclusion

This paper presents PVM, a minimal, high-performance, and software-based hypervisor for nested virtualization. PVM does not assume any hardware virtualization support and thus is portable to any cloud systems based on KVM. As a self-contained nested virtualization approach, PVM is transparent to the underlying host hypervisor and does not affect the security, complexity, or management of cloud platforms. We have demonstrated that PVM is the only viable approach with superior performance than hardware-assisted nested virtualization. PVM has been adopted in Alibaba Cloud for hosting secure containers, providing a more performant and cost-effective option for cloud users.

## 8 Acknowledgements

We thank the anonymous reviewers and our shepherd, Edouard Bugnion, for their constructive feedback and suggestions. This work is supported by Alibaba Innovative Research, Ant Group Research Fund, and National Science Foundation of China under grant No.62032008. The corresponding authors are Jia Rao, Hui Lu, Jiangshan Lai, Tao Ma, and Song Wu.

## References

- [1] About nested virtualization. <https://cloud.google.com/compute/docs/instances/nested-virtualization/overview>.
- [2] Amd secure encrypted virtualization (sev). <https://www.amd.com/en/developer/sev.html#:~:text=AMD%20Secure%20Encrypted%20Virtualization%2DEncrypted,to%20a%20CPU%20register%20state>.
- [3] Amd virtualization (amd-v). <https://www.techtarget.com/searchitoperations/definition/AMD-V-AMD-virtualization>.
- [4] CloudSuite. <https://github.com/parsa-epfl/cloudsuite>.
- [5] Intel® trust domain extensions (intel® tdx). <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html>.
- [6] Kata Containers. <https://github.com/kata-containers>.
- [7] Lguest: The simple x86 hypervisor. <http://lguest.ozlabs.org/>.
- [8] LMBench - Tools for Performance Analysis. <https://lmbench.sourceforge.net/>.
- [9] X86-s external architectural specification. <https://cdrdv2-public.intel.com/776648/x86s-EAS-v1-4-17-23-1.pdf>.
- [10] gVisor. <https://gvisor.dev/>, 2023.
- [11] ABAL, I., BRABRAND, C., AND WASOWSKI, A. 42 variability bugs in the linux kernel: a qualitative analysis. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE)* (2014), pp. 421–432.
- [12] ABRAMSON, D., JACKSON, J., MUTHRASANALLUR, S., NEIGER, G., REGNIER, G., SANKARAN, R., SCHOINAS, I., UHLIG, R., VEMBU, B., AND WIEGERT, J. Intel virtualization technology for directed i/o. *Intel technology journal* 10, 3 (2006).
- [13] ADAMS, K., AND AGESEN, O. A comparison of software and hardware techniques for x86 virtualization. *ACM Sigplan Notices* 41, 11 (2006), 2–13.
- [14] ADAMSON, A. Specjbb2005-a year in the life of a benchmark. In *2007 SPEC Benchmark Workshop* (2007).
- [15] AGACHE, A., BROOKER, M., IORDACHE, A., LIGUORI, A., NEUGEBAUER, R., PIWONKA, P., AND POPA, D.-M. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 419–434.
- [16] AHUJA, S. P., AND DEVAL, N. System level benchmarks for the cloud. *Computer and Information Science* 8, 2 (2015), 58.
- [17] BACOU, M., TODESCHI, G., HAGIMONT, D., AND TCHANA, A. Nested virtualization without the nest. In *Proceedings of the 48th International Conference on Parallel Processing* (2019), pp. 1–10.
- [18] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. *ACM SIGOPS operating systems review* 37, 5 (2003), 164–177.
- [19] BEN-YEHUDA, M., DAY, M. D., DUBITZKY, Z., FACTOR, M., HAR’EL, N., GORDON, A., LIGUORI, A., WASSERMAN, O., AND YASSOUR, B.-A. The turtles project: Design and implementation of nested virtualization. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)* (Vancouver, BC, Oct. 2010), USENIX Association.
- [20] BEN-YEHUDA, M., DAY, M. D., DUBITZKY, Z., FACTOR, M., HAR’EL, N., GORDON, A., LIGUORI, A., WASSERMAN, O., AND YASSOUR, B.-A. The turtles project: Design and implementation of nested virtualization. In *Osdi* (2010), vol. 10, pp. 423–436.
- [21] BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques* (2008), pp. 72–81.
- [22] BREWER, E. A. Kubernetes and the path to cloud native. In *Proceedings of the sixth ACM symposium on cloud computing* (2015), pp. 167–167.
- [23] BUGNION, E., DEVINE, S., GOVIL, K., AND ROSENBLUM, M. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 412–447.
- [24] BUGNION, E., DEVINE, S., ROSENBLUM, M., SUGERMAN, J., AND WANG, E. Y. Bringing virtualization to the x86 architecture with the original vmware workstation. *ACM Transactions on Computer Systems (TOCS)* 30, 4 (2012), 1–51.
- [25] BUGNION, E., NIEH, J., TSAFRIR, D., AND MARTONOSI, M. *Hardware and software support for virtualization*. Springer, 2017.
- [26] CANILLAS, J. M. Kbuild: the linux kernel build system. *Linux Journal* 2012, 222 (2012), 1.
- [27] DEVICES, A. M. Amd64 architecture programmer’s manual volume 2: System programming. 2023 (2023).
- [28] EDGE, J. A seccomp overview. *Linux Weekly News* (2015).
- [29] FISHMAN, A., RAPOPORT, M., BUDILOVSKY, E., AND EIDUS, I. HVX: Virtualizing the cloud. In *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 13)* (San Jose, CA, June 2013), USENIX Association.
- [30] GE, X., KUO, H.-C., AND CUI, W. Hecate: Lifting and shifting on-premises workloads to an untrusted cloud. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (2022), pp. 1231–1242.
- [31] GOPALAN, K., KUGVE, R., BAGDI, H., HU, Y., WILLIAMS, D., AND BILA, N. Multi-Hypervisor virtual machines: Enabling an ecosystem of hypervisor-level services. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, July 2017), USENIX Association, pp. 235–249.
- [32] GOVIL, K., TEODOSIU, D., HUANG, Y., AND ROSENBLUM, M. Cellular disco: Resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the seventeenth ACM symposium on Operating systems principles* (1999), pp. 154–169.
- [33] GRUENBACHER, A., AND ARNOLD, S. Apparmor technical documentation, 2007.
- [34] HALLYN, S. E., AND MORGAN, A. G. Linux capabilities: Making them work.
- [35] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium* (2007), vol. 1, Dttawa, Dntorio, Canada, pp. 225–230.
- [36] LI, Z., CHENG, J., CHEN, Q., GUAN, E., BIAN, Z., TAO, Y., ZHA, B., WANG, Q., HAN, W., AND GUO, M. RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 53–68.
- [37] LIM, J. T., DALL, C., LI, S.-W., NIEH, J., AND ZYNGIER, M. Neve: Nested virtualization extensions for arm. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 201–217.
- [38] LIM, J. T., AND NIEH, J. 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* (2020), pp. 557–574.
- [39] LINGAYAT, A., BADRE, R. R., AND GUPTA, A. K. Integration of linux containers in openstack: An introspection. *Indonesian Journal of Electrical Engineering and Computer Science* 12, 3 (2018), 1094–1105.
- [40] NEIGER, G., SANTONI, A., LEUNG, F., RODGERS, D., AND UHLIG, R. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal* 10, 3 (2006).
- [41] PAN, Z., HE, Q., JIANG, W., CHEN, Y., AND DONG, Y. Nestcloud: Towards practical nested virtualization. In *2011 International Conference on Cloud and Service Computing* (2011), IEEE, pp. 321–329.
- [42] POPEK, G. J., AND GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. *Communications of the ACM* 17, 7 (1974), 412–421.
- [43] PRATT, I., FRASER, K., HAND, S., LIMPACH, C., WARFIELD, A., MAGENHEIMER, D., NAKAJIMA, J., AND MALLICK, A. Xen 3.0 and the art of virtualization. In *Linux symposium* (2005), vol. 2, Citeseer, pp. 65–78.

- [44] REN, J., QI, Y., DAI, Y., XUAN, Y., AND SHI, Y. Nosv: A lightweight nested-virtualization vmm for hosting high performance computing on cloud. *Journal of Systems and Software* 124 (2017), 137–152.
- [45] TUNDE-ONADELE, O., HE, J., DAI, T., AND GU, X. A study on container vulnerability exploit detection. In *2019 IEEE International Conference on Cloud Engineering (IC2E)* (2019), IEEE, pp. 121–127.
- [46] UHLIG, R., NEIGER, G., RODGERS, D., SANTONI, A. L., MARTINS, F. C., ANDERSON, A. V., BENNETT, S. M., KAGI, A., LEUNG, F. H., AND SMITH, L. Intel virtualization technology. *Computer* 38, 5 (2005), 48–56.
- [47] VAN DOORN, L. Hardware virtualization trends. In *ACM/Usenix International Conference On Virtual Execution Environments: Proceedings of the 2 nd international conference on Virtual execution environments* (2006), vol. 14, pp. 45–45.
- [48] VILANOVA, L., AMIT, N., AND ETSION, Y. Using smt to accelerate nested virtualization. In *Proceedings of the 46th International Symposium on Computer Architecture* (2019), pp. 750–761.
- [49] WASSERMAN, O. Nested virtualization: shadow turtles. <https://www.linux-kvm.org/images/e/e9/Kvm-forum-2013-nested-virtualization-shadow-turtles.pdf>.
- [50] WILLIAMS, D., JAMJOOM, H., AND WEATHERSPOON, H. The xen-blanket: Virtualize once, run everywhere. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, Association for Computing Machinery, p. 113–126.
- [51] WILLIAMS, D., KOLLER, R., LUCINA, M., AND PRAKASH, N. Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2018), SoCC '18, Association for Computing Machinery, p. 199–211.
- [52] YOUNG, E. G., ZHU, P., CARAZA-HARTER, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. The true cost of containing: A gvisor case study. In *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)* (2019).
- [53] ZAHKA, D., KOCOŁOSKI, B., AND KEAHEY, K. Reducing kernel surface areas for isolation and scalability. In *Proceedings of the 48th International Conference on Parallel Processing (ICPP)* (2019), pp. 1–10.
- [54] ZHANG, X., ZHENG, X., WANG, Z., YANG, H., SHEN, Y., AND LONG, X. High-density multi-tenant bare-metal cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (2020), pp. 483–495.