A Thesis for the Degree of Ph.D. in Engineering

Enhancing Performance and Security of Virtual CPUs in Cloud Environments

February 2024

Graduate School of Science and Technology
Keio University

Kenta Ishiguro

Acknowledgement

I would like to thank my advisor, Prof. Kenji Kono. His constant guidance helped me in all the time of research. I would like to express my sincere gratitude to my collaborator: Dr. Pierre-Louis Aublin. I am also thankful to my colleagues in the sslab. Their surprising enthusiasm and skills have always inspired me. This dissertation would not have been possible without their advice and encouragement.

I am grateful to the members of my thesis committee as well: Prof. Masaaki Kondo, Prof. Baptiste Lepers, and Prof. Jianchen Shan. Their valuable feedback greatly improved this dissertation.

I appreciate the financial support from the Amano Scholarship, the Core Research for Evolutional Science, and the Support for Pioneering Research Initiated by the Next Generation.

Finally, I thank my family, parents, and sister for their support all these years. Without their support and encouragement, many accomplishments in my life, including this dissertation, would not have been possible.

Abstract

Hardware virtualization is widely used in cloud computing platforms. Multi-tenancy and oversubscribing hardware resources are leveraged in many types of public cloud computing to maximize their data center efficiency. Hypervisors play a crucial role in achieving these two features by multiplexing virtual machines on a single physical server. Despite continued hypervisor studies, commodity hypervisors still suffer from inefficiencies of virtual CPUs (vCPUs) and security concerns. The complexity and large code base of the commodity hypervisors make it challenging to uncover issues and apply the results of studies.

In this dissertation, we revisit the design and implementation of commodity hypervisors to uncover issues of CPU virtualization and address the issues with modest modifications. Our investigation shows the following two issues. First, the vCPU scheduling with ad-hoc optimizations is insufficient to mitigate excessive vCPU spinning, which occurs when a vCPU is waiting in a spin loop for an event from a descheduled vCPU. Second, ignorance of contexts in instruction emulation leads to a large attack surface.

To address excessive vCPU spinning, we identify three problems: 1) scheduler mismatch, 2) aggressive limitation of candidate vCPUs, and 3) inter-processor interrupt (IPI) context misuse. The first problem stems from the mismatch between the KVM vCPU scheduler and the Linux scheduler. The second and third problems come from failures in choosing candidate vCPUs to be scheduled next. Our in-depth analysis reveals simple modification to KVM (89 LoC) can mitigate excessive vCPU spinning. Our simple modification reduces excessive vCPU spinning by up to 96% and improves benchmark performance by up to 2.6×. Part of the proposed mitigation has been integrated with KVM from Linux KVM v5.13 onward.

To address the large attack surface of the instruction emulator, we propose

FWinst that is designed to identify illegitimate instructions and prevent them from being emulated based on emulation contexts. The key insight behind FWinst is that the instruction emulator needs to emulate only a small subset of instructions, depending on the underlying CPU micro-architecture and the hypervisor configuration. We have implemented a prototype of FWinst on KVM with modest modifications (279 LoC). Our experimental results demonstrate that FWinst defends against 14 real-world vulnerabilities in the KVM instruction emulator with negligible runtime overhead.

The contribution of this dissertation is twofold. First, we show that virtual machines running on KVM still suffer from excessive vCPU spinning. We uncover the three problems that incur frequent excessive vCPU spinning and propose mitigations with modest modifications. This improves the performance of vCPUs in cloud computing. Second, we show that the instruction emulator in the commodity hypervisor has a large attack surface. We design and implement FWinst to eliminate the need to emulate many instructions on CPUs with full-fledged support for virtualization by considering emulation contexts. This improves the security of vCPUs in cloud computing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Hypervisors play a crucial role in cloud computing. They enable the efficient sharing of cloud providers' physical resources among multiple virtual machines (VMs) by multiplexing hardware resources, including CPUs, memory, and I/O devices. This capability provides two fundamental characteristics of cloud computing: high resource efficiency and multi-tenancy.

High resource efficiency is critical to cloud computing, allowing cloud providers to enhance their data center efficiency regarding the price-per-performance ratio. Hypervisors employ a technique called CPU oversubscription to achieve this goal. CPU oversubscription enables multiple virtual CPUs (vCPUs) to share a single physical CPU, effectively increasing the number of VMs that can run on a given server. This approach improves resource utilization by allowing cloud providers to allocate idle resources to VMs that need them.

Multi-tenancy refers to the ability of a single physical server to host multiple VMs, each operating independently and securely. Hypervisors achieve multi-tenancy by isolating each VM, preventing them from interfering with each other's operations. This isolation is achieved through hypervisors' memory access control mechanisms and fair resource allocation methods. Hypervisors ensure that each VM has access to its resources while preventing any single VM from consuming excessive resources and accessing the memory contents of other VMs. Multi-tenancy allows cloud providers to offer cost-effective and scalable solutions to a wide range of users.

This dissertation focuses on CPU virtualization, which is at the heart of hy-

pervisors. While essential, CPU virtualization is a complex endeavor that demands careful implementation. CPU virtualization issues can compromise resource efficiency and multi-tenancy, two fundamental characteristics of cloud computing. Scheduling vCPUs is crucial for maintaining fairness among VMs and maximizing hardware utilization. However, suboptimal scheduling algorithms lead to resource wastage in guest VMs. Moreover, unfair resource allocation can undermine multi-tenancy, allowing a single VM to consume resources at the expense of others.

The point that CPU virtualization is implemented in the most privileged CPU mode introduces security concerns. Vulnerabilities in CPU virtualization can compromise multi-tenancy because attackers gain full system control once they exploit them. However, bug-free implementation of CPU virtualization is challenging because developers must consider various possible architectural states during implementation. Additionally, the trade-off between security and performance is a constant challenge in cloud computing. Ensuring hypervisor security without compromising resource efficiency requires careful consideration.

Early CPU virtualization relied on a software approach, leading to performance overhead without optimizations and implementation complexities. To overcome these problems, hardware virtualization extensions like Intel VT-x have emerged and become available for commodity environments. These extensions enable hypervisors to offload virtualization tasks, alleviating the implementation complexities.

The development of hardware virtualization extensions continues. The evolution of the hardware virtualization extensions provides room for resource efficiency improvements to the hypervisor and allows task offloading. For example, Intel pause-loop-exiting (PLE) [77] can improve resource efficiency in cloud computing. PLE is a later-introduced hardware assist that notifies when a vCPU is stuck in a pause loop for an extended period. The second-level address translation, like Intel extended page table (EPT), is also a later-introduced hardware assist that releases hypervisors from shadow page table management for guest VMs.

1.1 Motivation

Despite the efforts of many prior studies and hardware virtualization extensions, KVM [55], a widely adopted open-source commodity hypervisor [26, 33], continues to suffer from resource inefficiencies and security concerns of CPU virtualization. Applying the existing studies to commodity hypervisors is still challenging because modern commodity hypervisors, including KVM, are complex and integrated with an operating system (OS) kernel to leverage OS functionalities. Although leveraging OS functionalities simplifies the implementation and maintenance of the hypervisors, it makes it hard to modify those functionalities for hypervisor-specialized use cases while keeping the maintainability of the hypervisors.

Since prior studies aim to resolve general issues in hypervisors without considering applicability to the commodity hypervisors, they left two research questions: 1) what are the real-world issues that cause inefficiency and security concerns in KVM CPU virtualization, and 2) can the real-world issues be mitigated with minimal host modifications? Thus, exploring the real-world issues and mitigations against them is valuable to enhance the performance and security of cloud environments that leverage the commodity hypervisor.

1.2 Dissertation Contributions

This dissertation focuses on uncovering and resolving performance and security issues in CPU virtualization with minimal modifications. By revisiting the design and implementation of KVM, we gain a deeper understanding of its internal workings, enabling us to identify and address underlying limitations. This approach allows us to develop effective mitigation strategies without introducing significant changes to the existing infrastructure.

1.2.1 Mitigating Performance Issue of CPU Virtualization

As mentioned, cloud providers strive to oversubscribe hardware resources like CPUs to maximize hardware utilization. However, oversubscription comes at a cost: it requires multiplexing virtual CPUs (vCPUs) on physical CPUs (pCPUs).

Oversubscription violates an underlying assumption of the operating system (OS) design: OSes assume all of the CPUs to be active, and even if halted, they can respond to interrupts immediately. If pCPUs are oversubscribed, the execution of vCPUs is preempted by the hypervisor to schedule vCPUs, and vCPUs are not always active or cannot respond to interrupts immediately. Violating this OS design assumption results in the well-known problem of excessive vCPU spinning [2, 3, 20, 30, 97, 46, 50, 51, 73, 74, 42, 86, 89, 91, 95, 104, 105, 108, 114, 80]. This occurs when a vCPU waits in a tight loop for the completion of a short synchronous task by a descheduled vCPU. The waiting vCPU spins until the hypervisor schedules the descheduled vCPU. Excessive vCPU spinning originates from the following variants of the scheduling problem: 1) lock-holder preemption (LHP), 2) lock-waiter preemption (LWP), and 3) delayed response to interrupts.

Excessive vCPU spinning is difficult to solve in VM-agnostic hypervisors. The problem stems from a semantic gap between the hypervisor and guest OSes; the hypervisor is ignorant of the contexts in which a vCPU is running. Recent hardware support for virtualization detects long-running tight loops to mitigate the excessive vCPU spinning. When excessive spinning is detected, an Intel processor raises an event called Pause Loop Exit (PLE), and the control is transferred to the hypervisor. The hypervisor gains a chance to reschedule vCPUs to solve the root cause of excessive vCPU spinning.

Efforts have been devoted in KVM to mitigate excessive vCPU spinning. In this dissertation, we investigate the mitigations introduced into KVM. Once a PLE event occurs, KVM preempts the spinning vCPU and *boosts* the priority of another vCPU in the same VM. This mechanism was introduced in Linux v2.6.39. KVM selects and boosts a vCPU from candidate vCPUs in a round-robin fashion at every PLE. KVM optimizes the vCPU selection in four ways. The first three optimizations are for LHP and LWP, and were introduced in Linux v3.5, v3.9, and v4.13, respectively. The fourth optimization is for PLE events caused by delayed response to interrupts and was introduced in Linux v5.2. Unfortunately, these optimizations do not always reduce PLE events because they struggle to solve each problem in ad hoc ways.

Our in-depth analysis reveals that KVM suffers from 1) *scheduler mismatch*, 2) *aggressive candidate limiting*, and 3) *inter-processor interrupt (IPI) context misuse*.

Scheduler mismatch is peculiar to hypervisors that are integrated with the host OS where the host OS scheduler schedules other threads, based on its own policy, along with vCPUs without any distinction. The KVM vCPU scheduler gives a hint to the host OS scheduler, but scheduler mismatch occurs if the host OS scheduler eventually ignores the hint because it contradicts the host scheduling policy.

Aggressive candidate limiting and IPI context misuse are problems that occur when selecting vCPUs to boost. Due to the semantic gap between VM-agnostic KVM and guest OSes, KVM does not always correctly identify the exact root cause of excessive vCPU spinning. The KVM vCPU scheduler attempts to limit candidate vCPUs to avoid boosting vCPUs that are less likely to be a root cause. Aggressive candidate limiting occurs if the vCPU scheduler misjudges the root cause and excludes the root-cause vCPU from the candidates. However, the vCPU scheduler sometimes selects vCPUs as candidates that cannot be the root cause. Our analysis reveals that IPI context misuse is a major cause of this phenomenon.

We introduce three mitigations for the three aforementioned problems in KVM. First, the *hierarchical vCPU debooster* mitigates scheduler mismatch by lowering the priority of the vCPU preempted by a PLE. Because lowering the priority does not interfere with other threads in the host, the host OS scheduler does not ignore the hint to lower the priority. Second, *IPI-aware boost* mitigates IPI context misuse and partially mitigates aggressive candidate limiting by tracking IPI communications between vCPUs. IPI-aware boost enhances the vCPU selection by taking into account the information on IPI senders and receivers. Third, *relaxed boost* mitigates aggressive candidate limiting. This works as a safety net for vCPU selection by boosting all descheduled vCPUs after boosting all candidate vCPUs.

We have implemented the three above mitigations in Linux v5.6 in as few as 89 LoC. Our evaluation on 8- and 28-core machines with two different overcommit ratios and multicore benchmarks shows that our mitigations reduce PLE events resulting from spinlocks and delayed interrupt response. This reduction in PLE events improves the benchmark performance by up to 2.6×. Our evaluation further shows that the hierarchical vCPU debooster resolves scheduler mismatch while maintaining system fairness, and IPI-aware boost and relaxed boost

reduce PLE events caused by delayed interrupt response without compromising the existing mitigations for PLE events caused by spinlocks. Part of IPI-aware boost has been already integrated into Linux v5.13 [38, 60].

1.2.2 Mitigating Security Issue of CPU Virtualization

While vulnerabilities in hypervisors are crucial in multi-tenant clouds, there are many reported vulnerabilities in the hypervisors. As of November 2018, 111 CVEs are reported for KVM [55] and 275 vulnerabilities are in Xen Security Advisories (XSA) [110].

This dissertation focuses on vulnerabilities in instruction emulation in the hypervisors. Ideally, the hypervisors would only need to emulate a small subset of the instruction set. However, on x86 architecture, the hypervisors may be required to emulate most instructions [7, 11] for the following cases:

- **Port I/O (PIO):** When an I/O port is accessed, the port I/O instructions are interpreted to emulate the accessed I/O device.
- **Memory Mapped I/O (MMIO):** An access to an MMIO region is trapped by the hypervisor and the accessing instruction is interpreted by the instruction emulator to emulate the accessed I/O device.
- **Shadow Page Tables:** Prior to Nehalem micro-architecture, Intel CPUs did not support second level address translation. To keep the consistency between "shadow" and "guest" page tables, the hypervisor tracked changes of guest page tables by trapping and emulating VM writes to them.
- **Real Mode:** Prior to Westmere micro-architecture, Intel CPUs prevented real-mode code from running in guest-mode. Since CPUs boot in real-mode, hypervisors began with emulating the virtual CPU execution [15].
- **Migration:** To allow VM migration between Intel and AMD CPUs, some hypervisors trap and emulate vendor-specific instructions such as `sysenter` (specific to Intel). If `sysenter` is executed on AMD, the hypervisor traps and emulates it.

- **User-Mode Instruction Prevention (UMIP):** UMIP is a security feature introduced since Cannon Lake micro-architecture, which prevents some privileged registers from being read at user-level. Some hypervisors emulate UMIP on legacy (before Cannon Lake) micro-architectures by emulating the instructions that read those privileged registers.

Emulating most of the x86 instructions is complicated and error-prone. In fact, vulnerabilities in x86 emulators are not rare. To name a few, CVE-2016-9756 points out vulnerabilities in `far jump` and `far ret`. CVE-2017-2584 reports those in `fxrstor`, `fxsave`, `sgdt`, and `sidt`. CVE-2015-0239 and CVE-2017-2583 report vulnerabilities in `sysenter` and `mov SS`, respectively. CVE-2016-9756, CVE-2017-2584, CVE-2015-0239, CVE-2017-2583 are all related to vulnerabilities in the emulator. Making matters worse, Amit et al. [7] demonstrate any instructions can be forced to be emulated. This attack allows an attacker to exploit a vulnerability in any instructions.

For the security issue of CPU virtualization, this dissertation presents *FWinst* (derived from "Instruction Firewall"), which raises the bar for attacks on instruction emulation by narrowing the attack surface against it. The key insight behind *FWinst* is twofold. First, the emulator supports a wide range of x86 instructions only for backward compatibility. Recent x86 micro-architectures diminish the need for instruction emulation. For example, allowing real-mode in guest-mode eliminates the need for emulating real-mode code in the hypervisors. Supporting second level address translation eliminates the need for emulating VM writes to guest page tables.

Second, a legitimate subset of instructions to be emulated depends on the *emulation context* in which the emulator is invoked. If the emulator accepts only the *legitimate* set of instructions in each context, the attack surface is narrowed because the attacker cannot exploit vulnerabilities in the emulation of instructions that are not legitimate in the current context. *FWinst* identifies six contexts: 1) PIO context, 2) MMIO context, 3) shadow page table context, 4) real-mode context, 5) migration context, and 6) UMIP context, and is given a list of legitimate instructions for each context. For example, in the migration context, `sysenter`, which is specific to Intel CPU, is emulated only on AMD CPUs; its emulation is denied on Intel CPUs. In the MMIO context, the emulator denies `jmp` instruction because an MMIO region is accessed only through memory access instructions

such as `mov`.

To narrow the attack surface, *FWinst* uses a hypervisor's configuration and determines which context is valid. When the hypervisor invokes the emulator to emulate an instruction, *FWinst* checks if the current context is valid. If it is not, no instruction is emulated. For example, if second level address translation is enabled, no instruction should be emulated in the shadow page table context. If the current context is valid, *FWinst* passes only the legitimate instruction to the emulator. For example, in the MMIO context, the legitimate set of instructions are memory-access instructions. Emulation of, for instance, `jmp` instruction, is denied.

We have implemented a prototype of *FWinst* on KVM (Linux version 4.8.1), which runs on Intel Westmere and Skylake micro-architectures with the full-fledged support for virtualization turned on. Our experiment demonstrates *FWinst* can defend against several attacks on vulnerabilities in the emulation of `sysenter`, `far jump`, `far ret`, `mov SS`, `fxrstor`, `fxsave`, `sgdt`, `sidt`, `clflush` and `hint-nop` in KVM (Linux version 4.8.1). It also shows the performance overheads of *FWinst* is negligible. Furthermore, the code size of *FWinst* is small (279 LoC) and unlikely to introduce new security holes.

1.3 Organization

This dissertation is organized as follows. Chapter 2 shows our in-depth analysis of KVM vCPU scheduler and our mitigations against the issues uncovered by our analysis. Chapter 3 introduces FWinst. This chapter shows FWinst effectively enhances the security of the KVM instruction emulation. Chapter 4 concludes this dissertation and disscusses the future directions.

Chapter 2

Mitigating excessive virtual CPU spinning

This chapter aims to demonstrate our three mitigations, *debooster*, *relaxed boost*, and *IPI-aware boost*, against excessive vCPU spinning that cause resource inefficiency in cloud environments. First, Section 2.1 describes excessive vCPU spinning that degrades guest VM performance running on KVM and the KVM approach against it. Second, Section 2.2 provides a quantitative analysis of the KVM vCPU scheduler, revealing three issues that enlarge excessive vCPU spinning in KVM. Then, Section 2.3 and 2.4 describe the details of our mitigations against the uncovered issues. The evaluation in Section 2.5 shows that our mitigations effectively reduce excessive vCPU spinning and improve guest VM performance.

2.1 Background and Motivation

In this section, we introduce the problem of excessive vCPU spinning in virtualized systems. Then, we present the current KVM approach to alleviate the performance degradation caused by excessive vCPU spinning.

2.1.1 Excessive Virtual CPU Spinning

CPU spinning is common in OSes when waiting for another CPU to complete a short task. In virtualized environments, this approach causes severe performance

degradation due to *excessive vCPU spinning*. This occurs when a vCPU spins to wait for an event from a descheduled vCPU, in which causes the waiting vCPU to not make any progress. Guest OSes are given the illusion their vCPUs are running continuously, but in reality, the hypervisor preempts them to schedule other vCPUs. Hence, the short task on a vCPU descheduled by the hypervisor can take a long time to complete while another vCPU is spinning and waiting for the descheduled vCPU.

Lock holder preemption (LHP) is a widely known problem that causes excessive vCPU spinning [97, 30]. If a vCPU holding a spinlock is scheduled out by the hypervisor, another vCPU waiting for the spinlock cannot make any progress. Older versions of Linux (until version 4.1) supported ticket spinlocks in which a lock is acquired in the requesting order. The ticket spinlock amplifies the problem of vCPU spinning because a vCPU waiting for a ticket spinlock cannot make any progress until all vCPUs preceding it in the ticket requesting order are scheduled. This problem is called Lock waiter preemption (LWP). Linux has dropped support for ticket spinlock to avoid LWP.

To mitigate vCPU spinning, hardware-level support for virtualization provides a function to detect excessive vCPU spinning and enable the hypervisor to re-schedule vCPUs. Modern processors are equipped with a special instruction (PAUSE in Intel x86) that gives a hint to the processor that the code is in a spinning loop. The use of PAUSE instructions in spin-wait loops is strongly recommend by Intel to improve the performance of spin-wait loops [36].

Pause Loop Exit (PLE) [77] in Intel x86 processors checks the interval between consecutive PAUSE instructions performed in kernel mode. If the interval is shorter than *PLE_gap*, a pre-defined parameter, the vCPU is considered to be excessively spinning. If the spinning continues beyond another pre-defined parameter, *PLE_window*, a VMExit is triggered to transfer control to the hypervisor, which deschedules the spinning vCPU and schedules another vCPU. KVM dynamically adjusts the *PLE_window* value to reduce false positives in excessive spinning identification based on the frequency of PLE events. The *PLE_window* value grows when PLE happens and shrinks when a vCPU is scheduled. AMD supports Pause Filter (PF) [6], which is essentially the same as PLE. This work focuses on PLE but can be applied to PF as well. Our work relies on PLE to detect excessive vCPU spinning, so user-level synchronization primitives like those

used in OpenMP [72] are out of the scope.

2.1.2 Revisiting VM-agnostic KVM vCPU Scheduler

This work provides a quantitative analysis of all changes made to the VM-agnostic KVM vCPU scheduler. A quantitative analysis of each change reveals that each attempt to resolve some of the root causes of excessive spinning but introduces new issues to be addressed. Because of the semantic gap, each change cannot resolve the root cause perfectly. Relaxed boost (described in Section 2.3.2) is complementary to all the changes, and tries to mitigate excessive spinning that the existing and proposed solutions fail to resolve.

To mitigate excessive vCPU spinning, the hypervisor needs to schedule the root-cause vCPU that causes another vCPU to spin. Modern hypervisors leverage hardware assistance such as Intel's PLE to detect excessive vCPU spinning. The hypervisor reschedules vCPUs every time a PLE event occurs.

A straightforward rescheduling policy postpones scheduling the pause-loop-exiting (PLE-ing) vCPU for a certain period with the expectation that the root-cause vCPU was scheduled during the period. The Xen credit scheduler and the initial KVM vCPU scheduler implement this policy. The current KVM vCPU scheduler has been enhanced; KVM deschedules the PLE-ing vCPU and selects another vCPU to schedule, expecting the scheduled vCPU to be the root cause.

The current KVM vCPU scheduler has **directed yield** [99], which allows a PLE-ing vCPU to yield to another vCPU directly. This mechanism was introduced in Linux v2.6.39. KVM is integrated into the Linux kernel so that the KVM vCPU scheduler cooperates with the Linux scheduler. KVM gives the Linux scheduler a hint regarding which vCPU should be boosted. To decide which vCPU is boosted, the KVM vCPU scheduler selects candidate vCPUs from all descheduled vCPUs. This mechanism is called *candidate vCPU selection*. Note that the Linux scheduler decides which thread to run in the end. The boosted vCPUs are not always scheduled immediately by the Linux scheduler.

Candidate vCPU selection is vital for mitigating excessive vCPU spinning. KVM attempts to boost the root-cause vCPU but does not know exactly which vCPU is the root cause because of the semantic gap. KVM selects a candidate vCPU to be boosted in two rounds. After boosting all candidates in the first

round, KVM rebuilds its candidate set in the second round. To avoid boosting vCPUs that are unlikely to be the root cause, the KVM vCPU scheduler has been enhanced with the following changes:

Change-A: Skips boosting lock-waiters in the first round. Introduced in Linux 3.5 to mitigate LHP and LWP. KVM assumed all vCPUs that have caused PLEs are lock-waiters. In the first round, KVM skips boosting the other PLE-ing vCPUs because they are unlikely to be lock-holders. In the second round, the skipped vCPUs are boosted as well as the other candidates because KVM favors boosting the lock-waiters. KVM expects the lock-holders to have released their locks in the first round. Consequently, all not-running vCPUs are the candidates in the second round and later. PLE-ing vCPUs are excluded from the candidates in the first round. Note that Change-A assumes PLE-ing vCPUs are lock-waiters, but PLE-ing vCPUs can be waiters of synchronous IPIs (described in Section 2.2.1).

Change-B: Avoids boosting HLT-ing vCPUs. Introduced in Linux 3.9 to boost the lock-holder vCPU more quickly than Change-A alone. If a vCPU has been halted with HLT instruction, KVM skips boosting it in both the first and second rounds because it is unlikely to be a spinlock-holder. HLT is not issued inside a critical section protected by a spinlock. After introducing Change-B, PLE-ing and HLT-ing vCPUs are excluded from the candidates in the first round, and HLT-ing vCPUs are excluded in the second round and later.

Change-C: Avoids boosting vCPUs in user mode. Introduced in Linux 4.13 to boost the lock-holder vCPU more quickly than Change-A and -B. If a vCPU is in user mode, KVM skips boosting it in both the first and second rounds because it cannot hold a spinlock in kernel code; PLE happens only when the vCPU is executing a pause-loop in kernel mode. After introducing Change-C, vCPUs in user mode are excluded from the candidates as well as PLE-ing and HLT-ing vCPUs in the first round, and HLT-ing vCPUs and vCPUs in user mode are excluded in the second round and later.

Change-D: Boosts HLT-ing IPI receivers. Introduced in Linux 5.2 to deal with delayed response to IPIs. To respond to an IPI quickly, the recipient vCPU should be scheduled as soon as possible even if it is halted with HLT. If not scheduled immediately, the vCPU waiting for the response to the IPI causes a PLE. However, Change-B excludes HLT-ing vCPUs from the candidates. Change-D

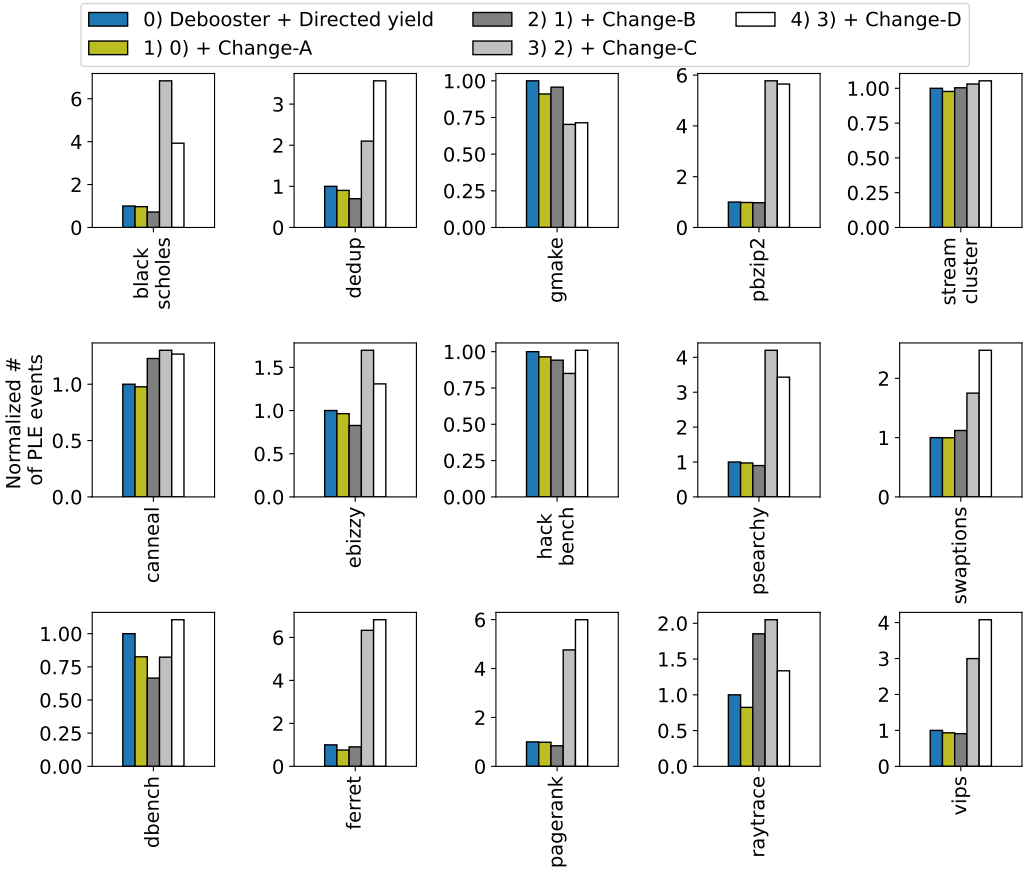


Figure 2.1. Normalized number of PLE events with Change-A to -D.

mitigates the negative effect due to Change-B for responses to IPI, while it maintains the positive effect for spinlock-holders. If a HLT-ing vCPU is an IPI receiver, it is considered a vCPU candidate to be boosted in the first round and later. After introducing Change-D, PLE-ing vCPUs, HLT-ing non-IPI-receivers, and vCPUs in user mode are excluded from the candidates in the first round, and HLT-ing non-IPI receivers and vCPUs in user mode are excluded from the candidates in the second round and later.

KVM has been optimizing its rescheduling policy for over a decade. However, the changes may introduce additional PLE events in some workloads like Change-B. Figure 2.1 shows the normalized number of PLE events of the benchmarks listed in Table 2.1 with Change-A to -D. The experimental setup is described in Section 2.2.3. Figure 2.1 indicates that both the expansion and shrink-

age of the candidate set with the changes cause an increase in PLE events depending on the benchmark. A more detailed analysis of Figure 2.1 is in Section 2.2.3. We address the increase in PLE events with relaxed boost that we describe in Section 2.3.2.

2.1.3 CPU Throttling

In commercial cloud environments, the pay-per-use model is standard. Hypervisors throttle the resources of each VM depending on the customers' payment. CPU throttling is one of the fundamental requirements to prevent each VM from exceeding its reserved resources.

KVM leverages cgroups to throttle CPU usage on each VM via `cpu.shares`, to which one cgroup is assigned. Throttling CPU usage with cgroups affects the KVM solution to excessive vCPU spinning because it changes the scheduling algorithm of Linux CFS. The group scheduling of Linux CFS is enabled to throttle the CPU usage of each VM. Run queues in the group scheduling are organized as a hierarchical tree. An internal node of the tree corresponds to one cgroup and has its run queue. A leaf node corresponds to a scheduling entity (vCPU or host thread), whose parent node is a cgroup to which the entity belongs. If the group scheduling is not enabled, a scheduling entity with the highest priority is chosen to run. Otherwise, CFS chooses an internal node (cgroup) in each hierarchy from top to bottom, and then schedules an entity with the highest priority in the leaf. All parent nodes account for the CPU usage in the leaf node.

This dissertation investigates the effectiveness of KVM mitigations of vCPU spinning with the hierarchical Linux CFS.

2.2 Analysis of KVM Behaviors

As described in Section 2.1, the KVM optimizations do not always reduce PLE events. This section shows the experimental results on Linux/KVM v5.6.0 which suggest that PLE events stem from the scheduler mismatch problem (Section 2.2.2) and the issues in the candidate vCPU selection (Section 2.2.3).

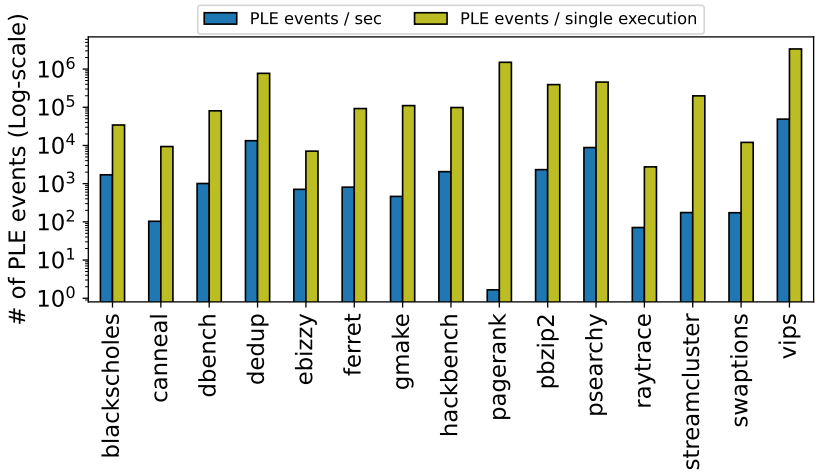


Figure 2.2. Number of PLE events in each benchmark (in log-scale). The left bar shows the number of PLE events per second. The right bar shows the average number of PLE events during a single execution.

2.2.1 Analysis of PLE Events

We evaluate the number of PLE events on Linux/KVM v5.6.0 with the default PLE parameters (PLE_gap set to 128 and PLE_window dynamically adjusted by KVM). Experiments are conducted on a machine containing a 2.10 GHz 8-pCPU Intel Xeon Silver 4110 processor with 128 GB of RAM. Hyperthreading is turned off.

In the experiments, two VMs run simultaneously on the machine. Each VM runs Ubuntu 18.04 LTS with Linux kernel 4.15 as the guest OS. We allocate eight vCPUs and 16 GB of RAM for each. One VM executes one of the benchmark from: `mosbench`, `parsec3.0`, `pagerank from CloudSuite`[28], `pbzip2`, `dbench`, `ebizzy`, and `hackbench` (detailed in Table 2.1). The other co-runner VM executes the CPU-intensive `swaptions`. All experiments are executed ten times, and the results are averaged.

PLE events in VM-agnostic KVM

Figure 2.2 shows the number of PLE events per second and the total number of PLE events in each benchmark in log scale. The seven benchmarks out of 15 show more than 1,000 PLE events per second. In particularly, `vips`, `dedup`, and

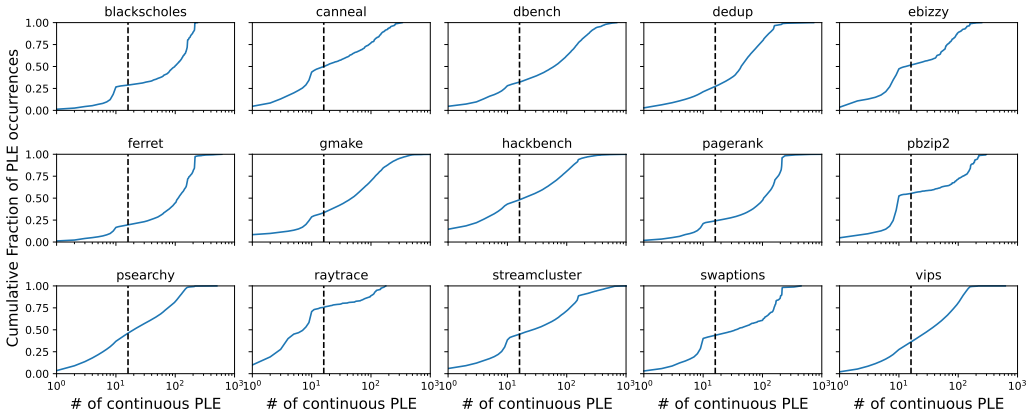


Figure 2.3. CDF of the length of continuous PLE events. The vertical dotted line shows the 16 (= 8 vCPUs × 2 rounds) continuous PLE events.

`psearchy` show 48,000, 13,000, and 8,800 PLE events per second, respectively. In `vips`, a PLE event occurs every 45,000 cycles, corresponding to 21 μs on our machine. In contrast, a PLE event occurs every 5,700 μs in `swaptions` which rarely causes PLE events.

Interestingly, PLE events occur *continuously* once one occurs. They occur at the same code location on the same vCPU without being interleaved with PLE events from other code locations. Figure 2.3 shows the CDF of the length of continuous PLE events in each benchmark. The vertical dotted line is drawn on 16 (= 8 vCPUs × 2 rounds) continuous PLE events because the KVM vCPU scheduler is expected to resolve the root cause by boosting all of the vCPUs in the 8-vCPU VM in two rounds. However, most PLE events are not resolved within 16 continuous PLE events. Moreover, 10 to 56% of PLE events come from continuous PLE events whose length exceeds 100.

PLE Reason

Spinning loops with PAUSE instructions are ubiquitous in the kernel and used for various purposes. To understand which kernel code causes PLE events, we trace the vCPUs' instruction pointers during the experiments. For instance, if a PLE event occurs while a vCPU executes `native_queued_spin_lock_slowpath` function (a spinlock implementation in Linux), the vCPU may be waiting for the completion of a lock-

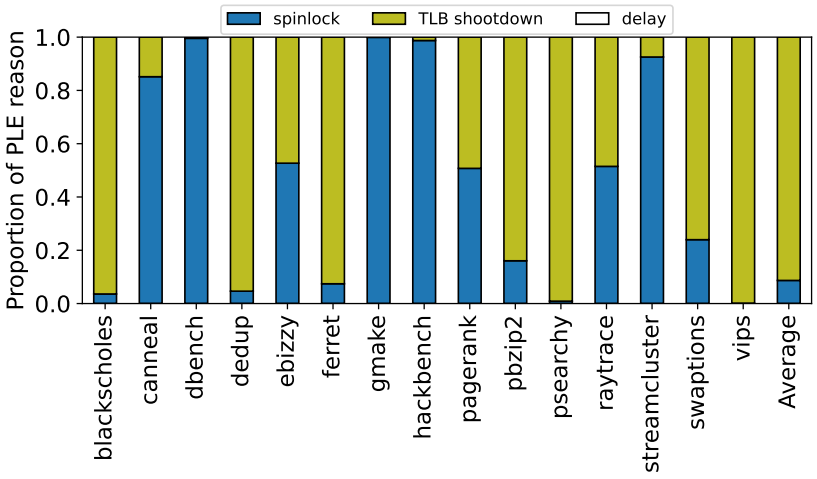


Figure 2.4. PLE reasons (intentional delays are rare).

holder vCPU's execution in a critical section.

In the experiments, we identified 28 functions that cause PLE events in the Linux kernel and categorized them into *spinlock*, *TLB shootdown*, and *intentional delay*. Note that PLE events related to TLB shootdown are caused by `smp_call_function_many`, which sends an IPI to multiple cores. This function is not solely for TLB shootdown, but we determine that all of the calls to the function that caused PLE events are for TLB shootdown. TLB shootdown causes a PLE event because a vCPU waits in a spinning loop for all the other vCPUs to flush their own TLBs.

Figure 2.4 shows the proportion of PLE reasons in each benchmark. Although the main PLE reason differs from benchmark to benchmark, spinlock and TLB shootdown are the two major causes of PLE events. More than 95% of PLE events in `vips`, `dedup`, and `psearchy` are caused by TLB shootdown because these benchmarks invoke many system calls to shrink the heap size that is shared address space among threads [2, 21]. In `dbench`, 99% of PLE events are caused by spinlock and more than 1,000 PLE events occur per second. The intentional delay is negligible (less than 0.1%).

Although the KVM vCPU scheduler takes both spinlock and TLB shootdown into account, they incur long continuous PLE events due to the scheduler mismatch problem (Section 2.2.2) and problems related to the candidate vCPU selec-

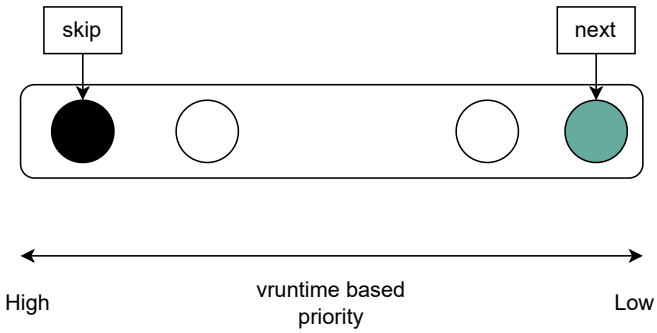


Figure 2.5. Example of scheduler mismatch problem (group scheduling disabled). Although KVM directs vCPU B to be boosted, CFS schedules vCPU A because vCPU B's priority is much lower than vCPU A's.

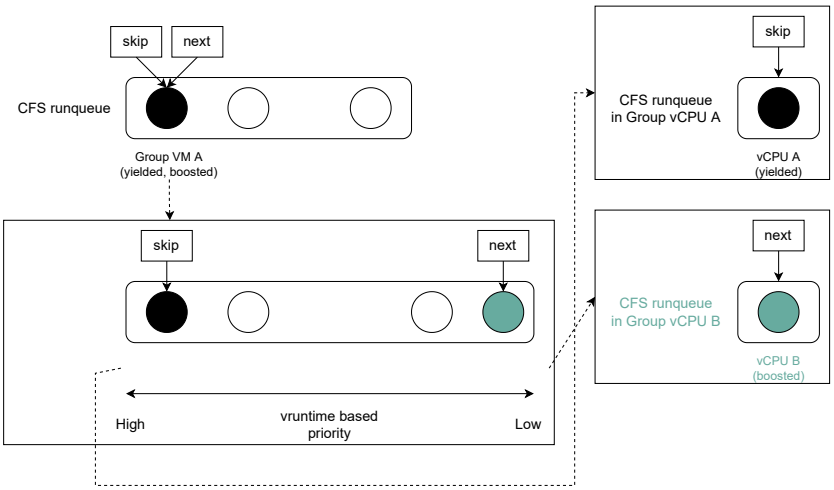


Figure 2.6. Example of scheduler mismatch problem with hierarchical scheduling. The KVM vCPU scheduler asks CFS to yield vCPU A and boost vCPU B so that all groups including vCPU A are labeled *skip* and all groups including vCPU B is labeled *next*.

tion (Section 2.2.3).

2.2.2 Scheduler Mismatch

The scheduler mismatch problem occurs when a scheduling hint from the vCPU scheduler is ignored by the host OS scheduler. When a vCPU causes a PLE event,

the KVM vCPU scheduler selects a candidate vCPU (a vCPU to be boosted) and conveys these vCPU threads as a scheduling hint to the Linux scheduler. The default Linux scheduler, Completely Fair Scheduler (CFS), computes the *virtual runtime* based on the actual *execution time* (as well as the nice value) for each vCPU thread or other host OS thread and schedules the one with the lowest virtual runtime. However, CFS has two special labels, *next* and *skip*, for threads in its run queue for temporary changes in priority. If a thread is labeled *next*, CFS schedules the thread as soon as possible, whereas if a thread is labeled *skip*, CFS usually postpones scheduling the thread. For instance, when the KVM vCPU scheduler asks CFS to change the priority of the vCPUs, CFS labels the PLE-ing vCPU thread as *skip* and the boosted vCPU thread as *next*. However, CFS does not always schedule the *next* thread immediately to maintain fairness among threads. CFS schedules threads regardless of the *skip* and *next* labels if the virtual runtime difference between threads is above the threshold defined by CFS because it is considered too unfair to schedule the *next* thread or to not schedule the *skip* thread.

If the *skip* vCPU happens to be in the same run queue as the *next* vCPUs, rescheduling the *skip* vCPU leads to continuous PLE events because the virtual runtime of the *skip* vCPU tends to be short due to the preemption. In contrast, the virtual runtime of the *next* vCPU tends to be long because it used the entire time slice especially in LHP. If the *skip* and the *next* vCPUs are in the same run queue, the *skip* vCPU is scheduled again but will be preempted immediately due to another PLE event. Figure 2.5 shows the worst case, i.e., when the yielded and boosted vCPUs are in the same run queue and the virtual runtime difference is above the threshold. Suppose vCPU B holds a spinlock and has used up the entire time slice. Then, vCPU A is scheduled and tries to acquire the lock but immediately causes a PLE event. The KVM vCPU scheduler gives a hint to CFS to yield vCPU A and boost vCPU B. Since vCPU A's virtual runtime is much shorter than vCPU B's, CFS prioritizes vCPU A and schedules it again.

The scheduler mismatch problem remains when the group scheduling is enabled to throttle CPU usage. The difference from the case without group scheduling is the CFS run queue where the scheduler mismatch occurs. As described in Section 2.1.3, the CFS run queue is organized hierarchically when the group scheduling is enabled. When CFS labels a vCPU thread with *skip* or *next*, all an-

cestor groups are also labeled *skip* or *next* to control CPU usage through cgroups (recall each cgroup corresponds to one group in the hierarchy). Figure 2.6 illustrates the scheduler mismatch problem with the group scheduling. The leaf nodes correspond to vCPU A (*skip*) and B (*next*), and their parent node groups them into a VM group. While the KVM vCPU scheduler notifies the vCPU threads that are the leaf nodes in Figure 2.6 to CFS, scheduler mismatch occurs due to the large virtual runtime difference between the group vCPU A and B, not between the vCPU A and B.

Because of the semantic gap between KVM and the host OS scheduler, it is impossible for the host OS scheduler to distinguish vCPUs from other scheduling entities. The host OS scheduler treats all vCPUs completely in the same way as normal threads. Therefore, the host OS scheduler adheres to its own scheduling policy, even if KVM requests it to boost a vCPU.

2.2.3 Issues in Candidate vCPU Selection

The candidate vCPU selection is the other important factor in reducing PLE events. We evaluate the multiple candidate selection algorithms with debooster to discern the effectiveness of vCPU selection.

As shown in Figure 2.1, the changes for the KVM vCPU scheduler do not consistently reduce PLE events. For example, while Change-C is effective for `gmake` and `hackbench`, it causes an increase in PLE events by more than 2× among 8 out of 15 benchmarks. On the other hand, Change-A reduces PLE events for all benchmarks. In the rest of this section, we quantitatively analyze the changes and define the two problems, *aggressive candidate limiting* and *IPI context misuse*.

Quantitative Analysis of Candidate vCPU Selection

Change-A reduces PLE events for all benchmarks in this experiment. Change-A primarily targets LHP and LWP. Although the ticket spinlock is not used in the experiment to avoid LWP, Change-A can reduce PLE events caused by LHP without negative side effects. Theoretically, the vCPU selection with Change-A delays boosting the root-cause vCPU if the PLE reason is not spinlock because it skips boosting other PLE-ing vCPUs in the first round. However, the skipped

vCPUs are eventually boosted in the second round. Consequently, Change-A reduces PLE events thanks to more LHP reduction than an increase in PLE events caused by TLB shootdown.

Change-B increases PLE events by more than 20% in `canneal` and `raytrace`. To resolve LHP quickly, Change-B excludes HLT-ing vCPUs from the candidates in the first round and later. This leads to *underboost*, in which the root-cause vCPU is excluded from the candidate vCPUs. The underboost happens when PLE events are caused by TLB shootdown, which uses IPIs synchronously. A vCPU that sends an IPI for TLB shootdown waits in a pause-loop for acknowledgements from the IPI receivers. Since HLT-ing vCPUs can receive IPIs, IPI receivers must be boosted even if it is a HLT-ing vCPU. If a HLT-ing IPI receiver is not boosted, the IPI sender triggers PLE events repeatedly until the HLT-ing IPI receiver is eventually scheduled. Thus, underboost causes long continuous PLE events once it occurs. Since PLE events caused by TLB shootdown occur in both `canneal` and `raytrace` as shown in Figure 2.4, underboost makes continuous PLE events dominant in these benchmarks.

Change-C increases PLE events by more than 2× in 8 benchmarks while it is effective for `gmake` and `hackbench`. Change-C leads underboost as well as Change-B when the PLE event is caused by TLB shootdown because it skips boosting vCPUs in user mode in the first round and later. Since a preempted vCPU in user mode may receive the synchronous IPI for TLB shootdown, IPI receivers must be boosted regardless of the vCPU mode.

Change-C is effective for LHP because Intel x86 raises PLE events only when the vCPU is running in kernel mode. As shown in Figure 2.1, Change-C reduces PLE events in `gmake` and `hackbench`, which are spinlock-intensive. The penalty of boosting vCPUs in user mode outweights the benefit of not boosting vCPUs in user mode. Underboost due to Change-C increases PLE events more significantly than Change-B because preempted vCPUs in user mode tend to run out CPU time, whereas HLT-ing vCPUs voluntary yield before running out CPU time.

Change-D increases PLE events in `vips` by 36% while it reduces PLE events in `canneal`, `ebizzy`, etc. Change-D has been introduced to mitigate the underboost caused by Change-B. If a HLT-ing vCPU is an IPI receiver, it is considered a candidate for vCPU selection. Unfortunately, Change-D causes *overboost*,

which boosts vCPUs that should not be boosted. A typical overboost scenario with Change-D is as follows. First, vCPU *A* sends an IPI to HLT-ing vCPU *B* and waits for a reply from vCPU *B*. Then, *before* vCPU *A* causes a PLE event, another vCPU *C* triggers a PLE event caused by LHP. In this scenario, vCPU *B* becomes a candidate for vCPU selection, although running vCPU *B* never resolves the root cause of vCPU *C* causing PLE events.

Each change reduces PLE events only in the specific benchmark for which it has been designed. However, the changes may cause underboost or overboost in unexpected situations and result in a significant increases PLE events in some benchmarks. We define the two problems, aggressive candidate limiting and IPI context misuse, as the root causes of these underboost and overboost.

Aggressive Candidate Limiting

Limiting candidates to boost is effective for benchmarks focused on, as shown the results with Change-A, B, and C. However, aggressive candidate limiting incurs underboost as described in the cases of Change-B and -C in Section 2.2.3.

Change-B and -C both attempt to narrow down candidates of vCPU selection to boost the root-cause vCPU more quickly. Unfortunately, due to the semantic gap between hypervisors and guest VMs, it is almost impossible to identify which vCPU is the root cause of PLE events. Change-B and -C have been introduced mainly to deal with LHP, but they falsely narrow down vCPU candidates if PLE events are triggered due to reason other than LHP. In the worst case, the root-cause vCPU never becomes a candidate for selection.

The most significant difference between Change-A and others is that Change-A boosts all not-running vCPUs in the second round. Although Change-A intends to boost lock-waiters in the second round, this algorithm eventually works as a safety net and avoids underboost.

IPI context misuse

Leveraging the IPI context is effective in resolving PLE events caused by TLB shootdowns, as shown in the results with Change-D in Figure 2.1. This is because the PLE-ing vCPU can be waiting for the HLT-ing IPI receivers. However, IPI context misuse for the candidate vCPU selection incurs overboost. Boosting IPI

receivers regardless of context degrades the effectiveness of the optimizations for the vCPU selection against LHP.

There are two cases of IPI context misuse. In the first case, as described in Section 2.2.3, the IPI receiver vCPU is the candidate even if the PLE-ing vCPU has not sent the IPI to the receiver vCPU. In the second case, an IPI is not synchronous. If the IPI is not synchronous, the IPI sender does not wait for a response from the receiver. Thus, asynchronous IPI receivers need not be boosted to resolve PLE events caused by TLB shootdown. Since the need for boosting IPI receivers depends on the IPI context, the context in which an IPI is sent must be deliberately taken into account.

2.3 Design

This section describes the design of our mitigations for excessive vCPU spinning. We introduce the vCPU hierarchical debooster to mitigate the scheduler mismatch problem, and IPI-aware boost and relaxed boost to improve the vCPU selection.

2.3.1 vCPU Hierarchical Debooster

As described in Section 2.2.2, scheduler mismatch occurs when the priority of the *skip* vCPU is much higher than that of the *next* vCPU. The *hierarchical debooster* alleviates the scheduler mismatch problem by *deboosting*, i.e., lowering the priority of, the *skip* vCPUs that are preempted by PLEs so that the *next* vCPU is scheduled before the *skip*. Since the priority difference between the deboosted *skip* vCPU and the *next* vCPU is set to be lower than the predefined threshold, scheduling the *next* before the *skip* does not violate the scheduling policy of the host OS.

The hierarchical debooster deboosts a *skip* vCPU only when it is in the same run queue as the *next*. In this case, setting the priority difference lower than the threshold can enforce the scheduling order in which the *next* is scheduled prior to the *skip*. If they are in different run queues, the scheduling order cannot be enforced because the *skip* and the *next* are scheduled independently of each other.

The hierarchical debooster can be applied to group scheduling for CPU throttling. When the group scheduling turned on, the debooster traverses the hierarchy of run queues to find the lowest common run queue of the *skip* and the *next*. In Figure 2.6, the lowest common queue is in Group VM A. Then, it lowers the priority of the *skip* in the lowest common queue if it is too high to switch to the *next*.

The priority of vCPUs is calculated from virtual runtime as described in Section 2.2.2. If the difference in virtual runtimes between the *skip* and the *next* is larger than a predefined threshold, CFS ignores the hint from the vCPU scheduler. The debooster sets the virtual runtime of the *skip* to (the virtual runtime of the *next*) − (the threshold). This adjustment makes the difference of virtual runtimes of the *skip* and *next* equal to the threshold, and CFS obeys the hint from the vCPU scheduler. Debooster does not undermine fairness between the *skip* and the *next* vCPUs because debooster upholds the priority of the *skip* as high as possible. Since the *skip*'s virtual runtime is still lower than the *next*'s virtual runtime (i.e. regarding virtual runtime, the *skip* has higher priority than the *next*) after adjusting their runtime, the *skip* will likely be scheduled after the *next* is scheduled. The current debooster design is similar to the design of the adjustment by CFS for woke-up threads/processes. Both designs keep the target thread priority as high as possible while lowering the target thread priority that has too much low virtual runtime. CFS updates a woke-up thread's virtual runtime equal to the minimum virtual runtime of the threads waiting to be scheduled if the woke-up thread has lower virtual runtime than the minimum virtual runtime in the run queue. This prevents threads that sleep a lot from having too much low virtual runtime compared to other threads in the run queue. Also, CFS ensures that the woke-up thread is the highest priority in the run queue. Debooster prevents *skip* vCPUs from having too much low virtual runtime but ensures the *skip* vCPUs have the possible highest priority.

This design does not violate the fairness of vCPU scheduling. CFS can maintain fairness as usual among VMs and other threads on the host because the debooster does not raise the priority of any vCPUs. Furthermore, the debooster contributes to more efficient utilization of CPU time. Without the debooster, the CPU time for the *skip* is wasted until CFS considers the boost to be fair because the *skip* cannot make any progress until the root cause is resolved. With the de-

booster, the CPU time is allocated to other VMs or threads because the execution of the *skip* is deferred.

Note that vCPU pinning is effective against scheduler mismatch. It ensures that each per-core run queue contains at most one vCPU of the same VM. The vCPU pinning reduces the degree of oversubscription, hence it is usually used for high-performance settings [115]. It also requires careful configuration of vCPU affinity. While mitigating the scheduler mismatch, the debooster enables the hypervisor to delegate management of load-balancing to the host OS scheduler.

2.3.2 Candidate Selection Improvement

As shown in Section 2.2.3, the candidate vCPU selection in the current KVM involves two issues, aggressive candidate limiting and IPI context misuse. We introduce *IPI-aware boost* to mitigate IPI context misuse and underboost caused by Change-C, and *relaxed boost* to mitigate aggressive candidate limiting.

IPI-Aware Boost

IPI-aware boost tracks IPI communication between vCPUs to select vCPU candidates to be boosted (or *next* in CFS). Tracking IPI communication enables the vCPU scheduler to take the IPI contexts into account in the vCPU selection. In IPI-aware boost, the vCPU scheduler checks if a vCPU that caused a PLE event has sent an IPI since the previous PLE. If an IPI has been sent, only the vCPU (either in user or kernel mode) that is supposed to receive the IPI becomes a candidate for boost. The other vCPUs, to which no IPI has been sent, are not included in the candidate vCPU to be boosted.

IPI-aware boost mitigates IPI context misuse and underboost caused by Change-C. In underboost, a receiver of a synchronous IPI is excluded from the vCPU candidates if it is in user mode. IPI-aware boost brings back an IPI receiver in user mode to the vCPU candidate. In IPI context misuse, IPI-aware boost does not boost a vCPU to which no IPI has been sent from the PLE-ing vCPU by tracking IPI communication. IPI-aware boost also deals with asynchronous IPIs. If IPI is asynchronous, an IPI sender will not trigger a PLE event because it does not wait for a reply from the IPI receiver. In IPI-aware boost, no IPI receiver becomes a candidate for boosting if the IPI sender does not trigger a PLE event.

Relaxed Boost

Relaxed boost mitigates aggressive candidate limiting. Because of the semantic gap between hypervisors and guest VMs, the vCPU scheduler cannot always determine which vCPU is the root cause of PLE events. Once the root-cause vCPU is overlooked, the PLE event repeatedly occurs until the root-cause vCPU is scheduled without boosting.

The relaxed boost boosts all not-running vCPUs in the second round and later. Not boosted vCPUs in the first round are back to the vCPU candidate in the second round by the relaxed boost. The relaxed boost behaves as a safety net for vCPU selection in the second round, while the vCPU selection benefits by limiting candidates in the first round. The key insight behind this design is that limiting candidates is effective for the specific benchmarks, but overlooking the root-cause vCPU increases a significant number of PLE events. Although the relaxed boost may delay scheduling root-cause vCPUs, it prevents root-cause vCPUs from not being scheduled for a long time.

To summarize, our mitigation works as follows. In the first round, it focuses on boosting either a spinlock holder or a TLB shootdown request receiver. In the second round, all descheduled vCPUs are candidates to be boosted. This design enables us to schedule root-cause vCPUs quickly in the first round and avoids overlooking root-cause vCPUs in the second round.

2.4 Implementation

We have implemented our mitigations on Linux/KVM with Linux kernel version 5.6.0. Our implementation modifies less than 100 lines of code in KVM and does not require code modifications in the guest. [1] Regarding portability, our mitigations do not require additional modification to port them from Linux kernel version 5.6.0 to 5.12-rc5.

Hierarchical vCPU debooster is implemented without modifying the KVM vCPU scheduler or the Linux scheduler core. The Linux scheduler, designed to work with the KVM vCPU scheduler, provides the `yield_to_task` interface that takes two scheduling entities, one for yielding (*skip*) and the other for

[1] Our patch is available at `https://github.com/sslab-keio/ple-kvm`.

boosting (*next*). This interface is implemented for each Linux scheduler. [2] We implement the hierarchical vCPU debooster inside `yield_to_task`.

The hierarchical vCPU debooster identifies the lowest common run queue to which *skip* and *next* vCPUs belong. It uses `for_each_sched_entity`, a helper function in CFS, to traverse the ancestor scheduling entities and find the lowest common run queue. If necessary, the hierarchical vCPU debooster adjusts the virtual runtime of the yielded vCPU group as described in Section 2.3.1.

IPI-aware boost is implemented in the KVM vCPU scheduler and the KVM virtual IPI handler. Every time a vCPU issues an IPI to another vCPU, the virtual IPI handler in KVM is invoked to virtualize the IPI. To keep track of IPI communication, IPI-aware boost records IPI senders and receivers in the handler. The records are cleared when the IPI receiver is scheduled. The KVM vCPU scheduler refers to the records when selecting vCPU candidates for boosting. For further optimization, we can make use of the IPI convention in the guest OS. For example, KVM can detect that an IPI is for TLB shootdown and synchronous by checking the least significant 8 bits in the IPI request value in Linux. This optimization can be introduced without any modifications to the guest OS if KVM can know in advance what guest OS is running.

Relaxed boost is implemented in the KVM vCPU scheduler. To record a vCPU that has been skipped for boosting in the first round, an extra field is added to the vCPU management structure. If this extra field indicates that the vCPU is skipped in the first round, the vCPU scheduler selects it for boosting and labels it as *next* in the second round. The field is cleared after the vCPU is scheduled by the host.

2.5 Evaluation

We evaluate our mitigations on a wide range of benchmarks on 8-core and 28-core servers with two different VM overcommit ratios. Our evaluation demonstrates that our mitigations reduce the number of PLE events in every combination of setups and improve the performance on the benchmarks. In addition, our mitigations maintain the system fairness and do not degrade the performance of

[2] To the best of the authors' knowledge, only CFS implements this interface.

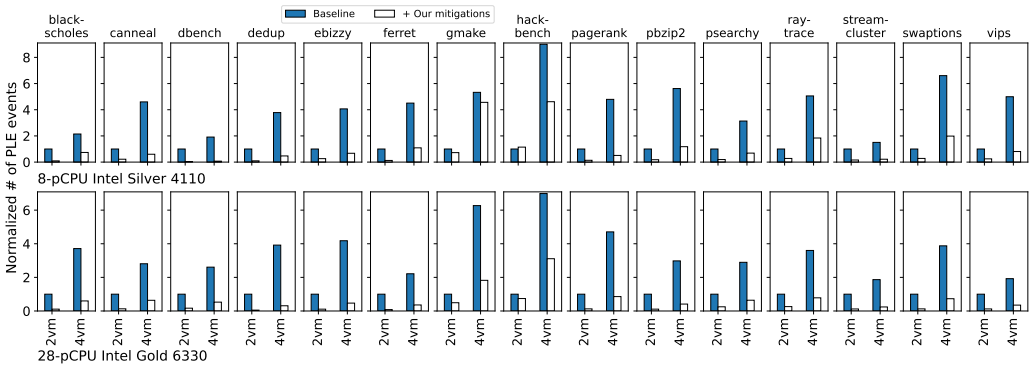


Figure 2.7. Reduction in number of PLE events, normalized with baseline KVM running two VMs.

co-runner VMs.

2.5.1 Experimental Settings

The baseline for our experiments is Linux/KVM v5.6.0. We use the default PLE parameters. PLE_gap is set to 128 and PLE_window is dynamically adjusted by KVM. We use two servers: the first one has a 2.10 GHz 8-pCPU Intel Xeon Silver 4110 processor with 128 GB of RAM, the second one has a 2.00 GHz 28-pCPU Intel Xeon Gold 6330 processor with 256 GB of RAM. Hyperthreading is turned off. Each VM runs Ubuntu 18.04 LTS with Linux kernel 4.15 as the guest OS with the same number of vCPUs as pCPUs in the server. For each VM, 16 GB of memory is allocated. Our experiments are conducted without static pinning.

Two or four VMs run simultaneously on each server so that the overcommit ratio is 2 to 1 or 4 to 1, respectively. One of the VMs executes a CPU-intensive benchmark, `swaptions`, and all the other VM(s) execute(s) a wide range of benchmarks: `mosbench`, `parsec3.0`, `PageRank from CloudSuite`, `pbzip2`, `dbench`, `ebizzy`, and `hackbench`. Table 2.1 describes each benchmark. We use these benchmarks to test KVM with a wide range of parallel workloads. The average of 10 executions is shown for all experiments.

Table 2.1. Multi-threaded benchmarks

Benchmark name	Workload
mosbench.gmake	Parallel build system
mosbench.psearchy	In-memory parallel search & indexer
parsec.blackscholes	Financial analysis with Black-Scholes
parsec.canneal	Minimizing the routing cost of a chip design
parsec.dedup	Compression with deduplication
parsec.ferret	Content-based similarity search
parsec.raytrace	Real-time raytracing
parsec.streamcluster	Online clustering of input stream
parsec.swaptions	Pricing of a portfolio of swaptions
parsec.vips	Image processing
CloudSuite.pagerank	Grapn analytics with PageRank
pbzip2	Data compressor
dbench	Filesystem I/O
ebizzy	Common web application servers
hackbench	Unix-socket or pipe stress

2.5.2 PLE Reduction

We measure the total number of PLE events for each benchmark. Figure 2.7 shows the normalized number of PLE events for each benchmark on the 8-pCPU and 28-pCPU server with the 2 to 1 (2-VM) and 4 to 1 (4-VM) overcommit ratios. Each result is normalized with the baseline KVM running two VMs. Our mitigations reduce the PLE events on average by 72.0%/73.9% and 80.0%/80.4% in 8-pCPU (2VM/4VM) and 28-pCPU (2VM/4VM), respectively.

The reduction rate of PLE events is higher on the 28-pCPU than on the 8-pCPU server. In addition, the rate is slightly higher on the 4-VM setting than on the 2-VM. Selecting vCPU candidates for boosting becomes more difficult with a greater number of vCPUs. Recall each VM has the same number of vCPUs as pCPUs in the server. If a root-cause vCPU is skipped by accident, PLE events continue to be triggered because it takes more time to schedule the root-cause

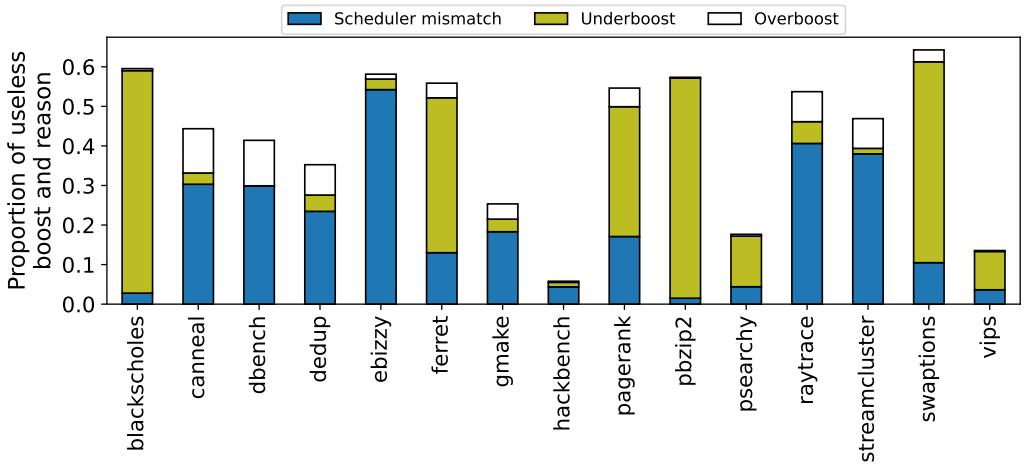


Figure 2.8. Proportion of scheduler mismatch, underboost, and overboost on 2-VM/8-pCPU setting.

vCPU on more vCPU VMs. The higher reduction rate on the 4-VM/28-pCPU setting suggests that our mitigations successfully select root-cause vCPUs in many situations.

Figure 2.8 shows the rate of scheduler mismatch, underboost, and overboost in the baseline KVM with the 2-VM/8-pCPU setting. The reduced rate of PLE events with our mitigations is higher than the total rate of the scheduler mismatch, underboost, and overboost in most benchmarks. For example, PLE events are reduced by 96%, but the total rate of the scheduler mismatch, underboost, and overboost is around 41% in `dbench`. Even if an inappropriate vCPU is boosted, the KVM vCPU scheduler believes the root cause has been resolved and schedules the vCPU that triggered a PLE event. In case of the scheduler mismatch and underboost, the vCPU scheduler schedules the vCPU again and again, triggering the PLE event. This amplifies the number of PLE events, and as a result, the reduction rate of the PLE events is higher than the total rate of the scheduler mismatch, underboost and overboost.

The reduction in PLE events for `psearchy` and `vips` is over 70%, but the total rate of scheduler mismatch, underboost, and overboost is less than 30%. Underboost is frequent in these two benchmarks because they are TLB shootdown intensive and spend most of their execution time in user mode. In addition, they trigger more than 10,000 PLE events per second as shown in Figure 2.2.

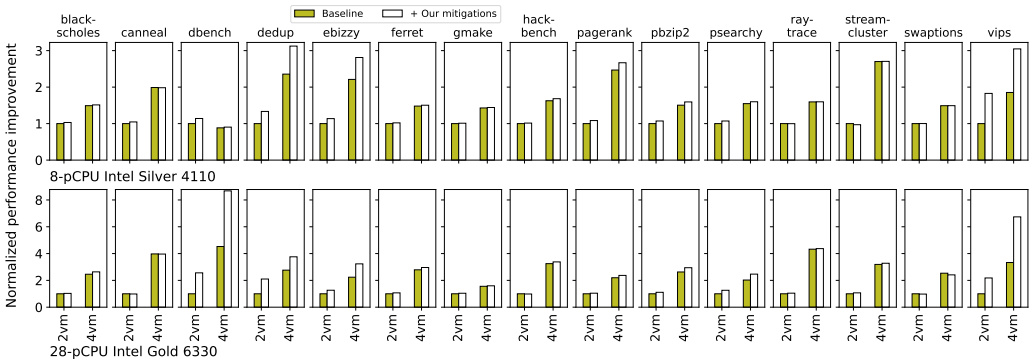


Figure 2.9. Performance improvement normalized with baseline KVM running 2VM.

Consequently, our mitigations effectively reduce PLE events, which have been amplified by frequent underboosts.

The reduction in PLE events for `gmake` and `hackbench` is less than 30%, where the total rate of scheduler mismatch, underboost, and deboost is also less than 30%. These benchmarks are spinlock intensive as shown in Figure 2.4. There is less room for improvement in these benchmarks with our mitigations. The number of PLE events slightly increases in `hackbench` with the 2-VM/8-pCPU setting. In all other settings, our mitigations reduce PLE events in `hackbench` because the issues in the baseline KVM become more severe in the settings with more vCPUs or/and pCPUs.

2.5.3 Benchmark Performance Improvement

Overall performance improvement

Figure 2.9 shows the performance normalized with the baseline KVM running 2VM/8-pCPU or 2VM/28-pCPU. The performance improvement is 12%/10%, 31%/22% on average in 8-pCPU (2VM/4VM) and 28-pCPU (2VM/4VM), respectively. The performance improvement is significant (33–118%) for `dedup` and `vips`. These benchmarks trigger a large number of PLE events (>10,000 PLEs/sec) as shown in Figure 2.2. Reducing PLE events largely contributes to the performance improvement. Other benchmarks (`pbzip2`, and `psearchy`), which trigger >1,000 PLEs/sec, show moderate performance improvement

(3%−21%).

Interestingly, the performance of `pagerank` improves by 8% even though it shows the lowest number of PLEs/sec among the 15 benchmarks. Although the rate of PLE events is low in `pagerank`, the total number of PLE events is high because of the long execution time. Thus, reducing PLE events contributes to performance improvement.

Comparing the performance of 8-pCPU and 28-pCPU, the performance improvement is greater on 28-pCPU (12 to 31% on 2VM, 10 to 22% on 4VM). The increased number of vCPUs increases PLE events; thus, our mitigations can reduce more PLE events in the 28-pCPU setting. For example, the number of PLE events increases 140× in `dbench`, and the performance improvement increases from 14% to 156% (2.6×).

Altough the performance improvement is lower in the 4VM setting than in the 2VM setting, this does not mean our mitigations are not effective for the 4VM settings. As shown in Figure 2.12, the average co-runner performance improvement is 3.6% with 4VM while it is 2.6% with 2VM.

Time Spent on Spinlock and TLB Shootdown

We evaluate the execution time of spinlock and TLB shootdown inside the guests to see the impact of PLE event reduction. We monitor two functions, `native_spin_lock` (spinlock) and `smp_call_function_many` (TLB shootdown), because these two functions are the major producers of PLE events.

Figure 2.10 shows the normalized execution time of spinlock and TLB shootdown inside the guests in each benchmark on the 8-pCPU server. Our mitigations reduce the total execution time from the baseline in all benchmarks by 48% on average. Since our mitigations suppress a large number of continuous PLE events, the worst-case latency is reduced. Consequently, the total execution time is reduced from the baseline.

The various performance improvement shown in the section comes from the proportion of the execution time of spinlock and TLB shootdown in the total benchmark execution time. Although the execution time reduction of spinlock and TLB shootdown is similar in `vips` and `ferret` by 59% and 55%, the perfor-

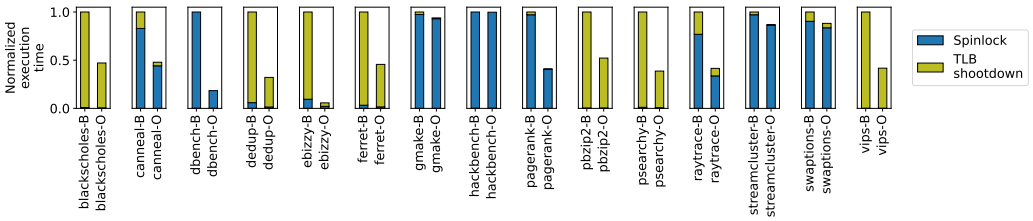


Figure 2.10. Normalized total time spent on `native_spin_lock` (spinlock) and `smp_call_function_many` (TLB shootdown). In the figure, "B" stands for "Baseline" and "O" stands for "+ Our mitigations"

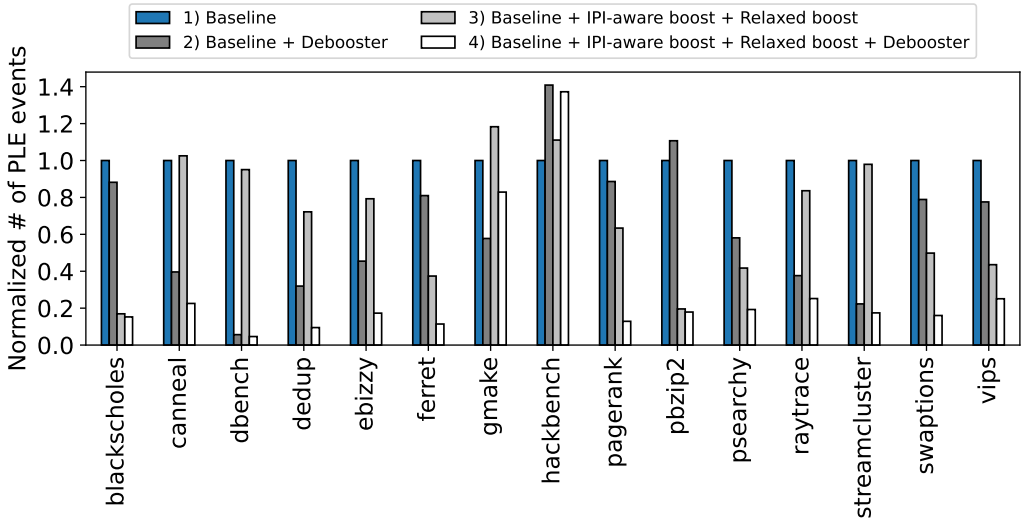


Figure 2.11. PLE reduction with/without debooster on 2VM/8-pCPU server, normalized with baseline KVM.

mance improvement of `vips` is more significant than `ferret`. The execution time of TLB shootdown is dominant in `vips`, whereas it is less dominant in `ferret`.

2.5.4 vCPU Hierarchical Debooster Effectiveness

PLE reduction by Debooster

To evaluate the effectiveness of the hierarchical debooster, we compare the number of PLE events with four configurations: 1) the baseline KVM, 2) 1) + de-

booster, 3) 1) + IPI-aware & relaxed boost, and 4) 3) + debooster. Figure 2.11 shows the normalized number of PLE events on the 2VM/8-pCPU setting. First, we compare 1) and 2) to determine the debooster effectiveness. The debooster reduces PLE events in 13 benchmarks out of 15, with an average reduction rate of 38%. Meanwhile, PLE events increase in `pbzip2` and `hackbench`. CPU utilization is high in these benchmarks, and the host Linux scheduler distributes vCPUs across different run queues as a result. Since the debooster works only when the *skip* and the *next* are in the same run queue, the debooster is not effective in these benchmarks.

Next, we compare 2) and 4) to verify that the IPI-aware & relaxed boost do not interfere with the debooster. The number of PLE events is lowered in all benchmarks except for `gmake`. By comparing 1) and 3) in `gmake`, we can see that the IPI-aware & relaxed boost increase PLE events. As `gmake` is a spinlock-intensive workload, it does not benefit from the IPI-aware boost which selects user-mode IPI-receivers for boosting. The comparison of 3) and 4) indicates that the debooster successfully reduces the PLE events inflated by the IPI-aware boost.

System fairness with debooster

To verify that the debooster does not violate scheduling fairness, we evaluate the performance of the co-runner VM. If the debooster severely degrades the performance of the co-runner VM, it interferes with the host OS scheduler; the vCPU priority lowered by the debooster has a negative impact on the co-runner VM. Figure 2.12 shows the normalized performance of `swaptions` running in the co-runner VM. Even with the debooster, the performance is almost the same as on the baseline KVM. It is improved 2.7% over the baseline on average. Interestingly, the performance co-located with `vips` improves by 29% because the reduction in PLE events gives more CPU time to the co-runner VM.

2.5.5 IPI-aware Boost Effectiveness

IPI-aware boost consists of two sub-mitigations: boosting user-mode IPI-receivers and HLT-ing IPI-receivers, if and only if a vCPU that triggered a PLE event has sent an IPI to them. To evaluate the effectiveness of IPI-aware boost, we compare the number of PLE events with four configurations: I-1) the baseline

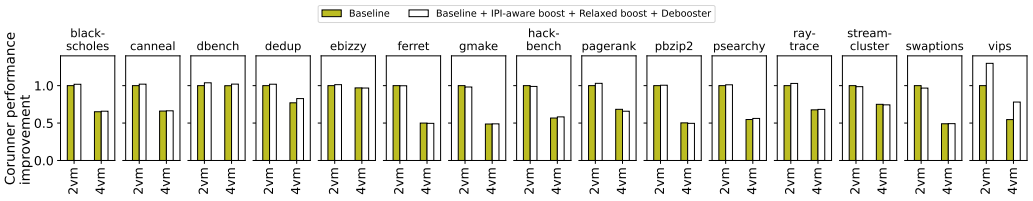


Figure 2.12. Performance of co-runner VM on 2 or 4VM/8-pCPU setting.

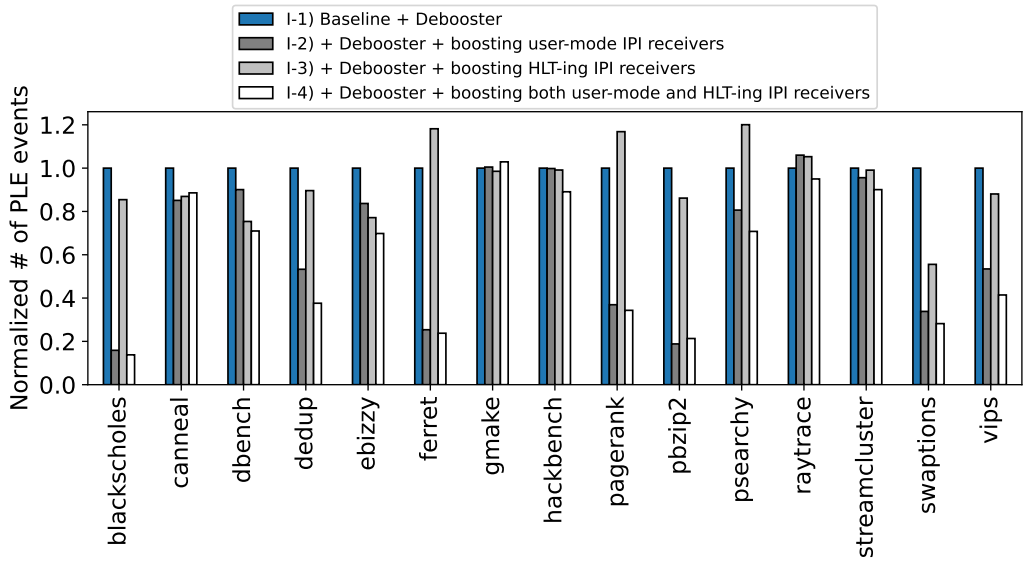


Figure 2.13. PLE reduction with/without IPI-aware boost on 2VM/8-pCPU, normalized with baseline KVM.

KVM + debooster, I-2) I-1) + boosting user-mode IPI receivers, I-3) I-1) + boosting HLT-ing IPI receivers, and I-4) I-2) & I-3). The debooster is incorporated in all configurations to mitigate scheduler mismatch. Figure 2.13 shows the number of PLE events normalized with I-1) on the 2VM/8-pCPU setting.

The comparison of I-1) and I-2), indicates that boosting user-mode IPI-receivers reduces PLE events by 35% on average. Since I-2) mitigates under-boost, it is effective for benchmarks that involve many TLB shootdowns such as `blackscholes`, `dedup`, `ferret`, `pbzip2`, `psearchy`, and `vips`. On the other hand, the benchmarks that are spinlock-intensive such as `gmake`, `hackbench`, and `raytrace` do not benefit from I-2). Although I-2) boosts non-lock-holder vCPUs, it is still effective for spinlock-intensive benchmarks

such as canneal, dbench, ebizzy, and pagerank. This indicates that a single occurrence of underboost incurs many PLE events because the vCPU scheduler never boosts the root-cause vCPU if it is in user mode.

The comparison of I-1) and I-3) shows that boosting HLT-ing IPI-receivers sometimes reduces PLE events but not always. Since it mitigates the overboost, the benchmarks that involve many TLB shootdowns such as blackscholes, dedup, pbzip2, and vips benefit from it. On the other hand, the benchmarks that are spinlock-intensive such as gmake, hackbench, and raytrace do not. It should be noted that I-3) increases PLE events for ferret, pagerank, and psearchy, although TLB shootdown is dominant in these benchmarks. As described in Section 2.2.3, the overboost is problematic when two vCPUs trigger PLE events at the same time, so introducing only I-3) is not effective in these benchmarks.

The comparison of I-1) and I-4) indicates that combining I-2) and I-3) reduces PLE events by 41% on average. Interestingly, I-4) is effective for the three benchmarks mentioned above (ferret, pagerank, and psearchy) whose PLE events are increased by I-3).

2.5.6 Relaxed Boost Effectiveness

To show the effectiveness of relaxed boost, we compare the number of PLE events on 2VM/8-pCPU with three configurations: R-1) the baseline KVM + debooster, R-2) R-1) + IPI-aware boost, and R-3) R-2) + relaxed boost. Note that the debooster is always turned on to mitigate the scheduler mismatch. Figure 2.14 shows the number of PLE events normalize with R-1). Relaxed boost reduces PLE events by 47% on average, which is 7% greater than R-2). The benchmark that benefits the most is psearchy, with a 66% reduction in PLE events.

Relaxed boost reduces PLE events from 17% to 54% in the benchmarks that involve many TLB shootdowns such as dedup, ebizzy, ferret, pbzip2, vips, canneal, pagerank, and streamcluster. Although IPI-aware boost mitigates PLE events caused by TLB shootdown, the KVM vCPU scheduler without the relaxed boost sometimes overlooks root-cause vCPUs when PLE events are triggered by TLB shootdown. For example, overlooking happens when the root-cause IPI receiver is scheduled but preempted before completing

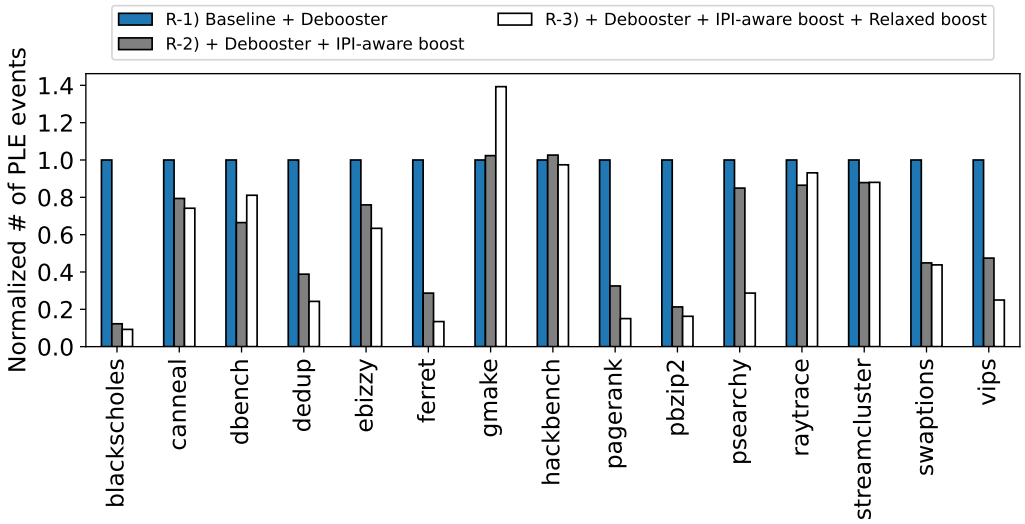


Figure 2.14. PLE reduction with/without relaxed boost on 2VM/8-pCPU server, normalized with baseline KVM + debooster.

the IPI response. IPI-aware boost clears the IPI communication history of the IPI receiver when the IPI receiver is scheduled. In this case, IPI-aware boost without the relaxed boost does not boost the root-cause vCPU in the rest of rounds because it considers the root-cause vCPU has already responded to IPI. Therefore, the relaxed boost is effective for the benchmarks that involve many TLB shootdowns.

In contrast, relaxed boost increases PLE events by up to 39% in spinlock-intensive benchmarks such as `dbench`, `gmake`, and `raytrace`. The baseline KVM rarely overlooks root-cause vCPUs if PLE events are triggered by spinlocks. The relaxed boost is a safety net to avoid overlooking root-cause vCPUs when PLE events are triggered by TLB shootdowns (more precisely, IPIs). If PLE events are seldom triggered by TLB shootdowns, the relaxed boost is likely to boost vCPUs not related to the root cause. Therefore, the relaxed boost tends to increase PLE events in spinlock-intensive benchmarks. Note that this does not mean the relaxed boost should not be applied. Compared with R-1), R-3) reduces PLE events in all the benchmarks other than `gmake`. Even in `gmake`, compared with the baseline KVM, R-3) reduces PLE events as seen in Figure 2.7.

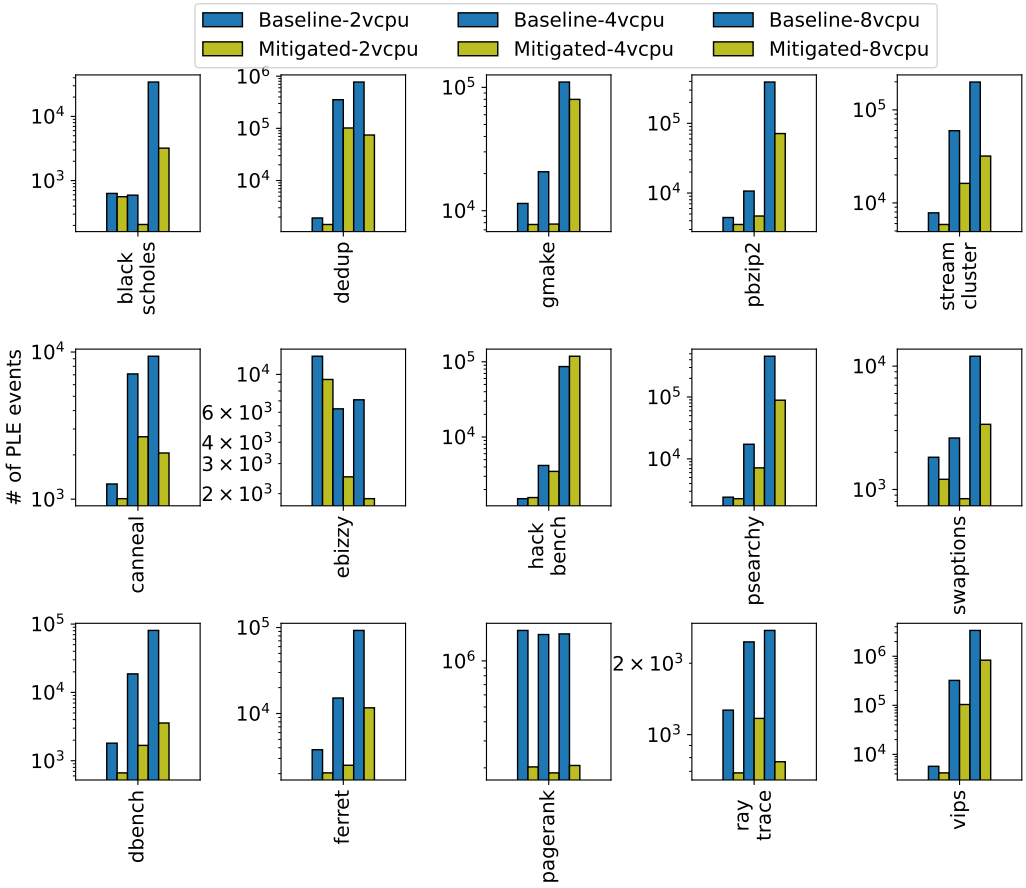


Figure 2.15. Reduction in the number of PLE events (in log-scale) in VMs of diffrent number of vCPUs, normalized with baseline KVM running 2-vCPU VM.

2.5.7 Effectiveness for VMs of different numbers of vCPUs

To show the effectiveness of our mitigations for VMs of different numbers of vCPUs, we compare the number of PLE events with 2-, 4-, and 8-vCPU VMs on the 8-pCPU server. In this setup, each benchmark shown in Table 2.1 runs in 2-, 4-, or 8-vCPU VM along with the CPU-intensive `swaptions` in the 8-vCPU VM on the 8-pCPU server.

As shown in Figure. 2.15, our mitigations reduce PLE events from the baseline for all the VMs of different numbers of vCPUs. The PLE event reduction is usually more significant for 8-vCPU VM than for 2- or 4-vCPU VMs. This is because PLE events more frequently occur with more vCPUs for two reasons. First, resolving

spinlocks becomes challenging as the number of vCPUs increases because the boosting candidates increase at every PLE event. Second, the waits due to TLB shootdown take longer and frequently happen as the number of vCPUs increases because the mapping can be shared with more vCPUs.

In some cases, PLE events do not increase even if the number of vCPUs increases. For example, in `ebizzy`, the experiment with 2-vCPU shows more PLE events than with 4-vCPU. The high spinlock rate for the in-kernel read/write semaphore increases PLE events from 2-vCPU to 4-vCPU in `ebizzy`. To acquire the semaphore, threads optimistically spin until exceeding their time limit. In VMs with more vCPUs, using blocking instead of spinning for synchronization reduces PLE events. The same phenomenon happens with 4-vCPU and 8-vCPU in `dedup` when our mitigations are enabled. Another PLE event reduction with more vCPUs can be seen in `blackscholes`, `canneal`, and `raytrace`. These reductions are due to shorter execution time with more vCPUs.

2.6 Related work

Alleviating virtualization overhead has been tempting for academics and industries for over a decade to improve cloud computing efficiency. Excessive vCPU spinning remains one of the major causes of the virtualization overhead of multithreading [80]. Most previous works tried to reduce virtualization overhead due to excessive vCPU spinning by leveraging para-virtualization or requiring heavy modification of hypervisor schedulers. In contrast, our work revisits the approach taken by VM-agnostic KVM and Linux CFS to uncover the issues that enlarge excessive vCPU spinning and address them with modest modification.

Initially, problems related to spinlocks were explored to reduce the virtualized overhead that stems from excessive vCPU spinning. Uhlig et al. [97] pointed out that the lock holder preemption (LHP) problem occurs when a vCPU holding a spinlock gets preempted, and all waiters waste CPU cycles for the lock. Ticket spinlocks [66], which are a form of spinlock that enforces ordering among lock acquisitions, incur the lock waiter preemption (LWP) problem when a particular waiter is preempted before acquiring the lock. To deal with LHP and LWP, a software-based approach with para-virtualization [30] and hardware-based approaches [77, 6, 104] were proposed to notify unusual long waits in

guest VMs to hypervisors. Several vCPU schedulers, including the KVM vCPU scheduler, leverage these hardware-based detection systems to mitigate excessive vCPU spinning. As mentioned earlier, when a PLE event occurs, candidate vCPU selection to boost directly affects the time it takes to resolve excessive vCPU spinning. Raghavendra proposed several optimizations for candidate selection of the KVM vCPU scheduler [42]. However, it incurs additional overheads on TLB shootdown heavy workloads because it focuses on resolving excessive vCPU spinning caused by spinlocks. APPLES [86] prioritizes resource waiter vCPUs to prevent excessive vCPU spinning that stems from spinlocks. These enhancements work in tandem with the KVM vCPU scheduler and our mitigations to resolve excessive vCPU spinning quickly. Several works proposed paravirtualized spinlock mechanisms to reduce excessive vCPU spinning. To avoid the LWP, a queue-based spinlock is used instead of the ticket-based spinlock [64]. Preemptable ticket spinlocks [73] allow vCPUs to acquire ticket spinlocks in relaxed order. Opportunistic ticket spinlock [45] was proposed to alleviate sleepy spinlock anomaly caused by the para-virtualized spinlock. All para-virtualized spinlocks need to modify the spinlock implementation of guest OSes.

Subsequent studies tackled the latency of IPI synchronization in guest VMs to address excessive vCPU spinning. To mitigate the problem, demand-based coordinated scheduling [50] prioritizes IPI involving vCPUs for fast responses to IPI by monitoring IPI signals. Contrary to our work, demand-based coordinated scheduling requires drastic scheduler changes to support urgent and load-conscious balance scheduling. To reduce the latency of IPI synchronization, paravirtualized TLB shootdown schemes were proposed [59, 74, 49, 9]. Performing remote TLB flush through a hypercall can reduce not only excessive vCPU spinning but also VMExit because of sending IPIs, but these works require guest OS modification as well as hypervisor modification.

Serving multi-threaded applications with micro-timeslice vCPUs can complete critical sections in guest VMs quickly, resulting in effective mitigation of excessive vCPU spinning. Since frequent context switches with micro-timeslice vCPUs degrade the performance of cache-sensitive applications, the negative effect due to frequent context switches needs to be minimized. Ahn et al. [2] proposed a scheme to serve only critical OS services with micro-timeslice by leveraging kernel symbols of guest OSes and the instruction pointer of vCPUs to

detect vCPUs preempted while executing critical OS services. Reducing the cost of context switches by manipulating caches with new architectural support [3] is also effective in minimizing the negative effect of micro-timeslice vCPUs. Several works [108, 51, 96, 116] leverage vCPU behavior (e.g., spinlock latency) as an indicator to identify applications' characteristics and change vCPU timeslice based on the applications. However, these approaches require heavy modifications to their host scheduler.

Sharing the scheduling information of host OS and guest VMs is another possible approach to address virtualization overhead. I-Spinlock [95] and eCS [46] share the guest OS execution information with hypervisors to change the scheduling time slice if vCPUs are in critical sections. Dynamic vCPU scaling [89, 20] allows the guest scheduler to schedule threads in guest VMs on only active vCPUs. While dynamic vCPU scaling is effective in avoiding excessive vCPU spinning because it increases the chance that each vCPU occupies a dedicated pCPU, it requires modification of the guest OSes and the hypervisor scheduler for fast vCPU reconfiguration. IRS [117] applies the classical scheduler activation approach in hybrid threading to vCPU scheduling for excessive vCPU spinning mitigation. IRS performs load-balancing by considering whether vCPUs are preempted. Thus the guest OS can migrate the thread running on the preempted vCPU to another running vCPU to avoid excessive vCPU spinning. To bridge the semantic gap, IRS requires changes to the guest OSes and the hypervisors. Contrary to these works, our approach tackles excessive vCPU spinning in a VM-agnostic way.

For hypervisor scheduler-based approaches, co-scheduling can be utilized to mitigate the negative effects of excessive vCPU spinning. It simultaneously schedules all the sibling vCPUs in the same VM [100]. Co-scheduling has drawbacks, such as CPU fragmentation and priority inversion. To address these drawbacks, some works minimize the period of co-scheduling by leveraging additional information with para-virtualization [105] or modifying drastically the hypervisor scheduler [114]. Although balance scheduling [91] can alleviate such drawbacks, it prevents migrating vCPUs to maintain a fair load balance. In contrast, because our mitigations do not require modifying the host OS scheduler core, a fair load balance is maintained.

2.7 Summary

Excessive vCPU spinning is a widely known problem caused by a semantic gap between the hypervisor and guest operating systems. Unfortunately, as we demonstrated, KVM still suffers from non-negligible overheads caused by excessive vCPU spinning.

We performed an in-depth analysis of excessive vCPU spinning in the VM-agnostic KVM hypervisor and analyzed the root causes of this problem. We then presented slight modifications (89 LOC) on the KVM vCPU scheduler that efficiently solve these issues and improve the performance by up to 2.6× without sacrificing scheduling fairness.

Chapter 3

Mitigating vulnerabilities in instruction emulation

The objective of this chapter is to demonstrate *FWinst*, which is a context-aware instruction filter to reduce the attack surface of the instruction emulator in commodity hypervisors. First, Section 3.1 describes the necessity of instruction emulation in KVM and how the evolution of hardware virtualization extensions eliminates the need for emulation. Second, Section 3.2 provides the threat model of *FWinst* and an analysis of the vulnerabilities in instruction emulation. Then, based on the analysis, Section 3.3 introduces *FWinst*. Section 3.4 shows that *FWinst* effectively protects hypervisors against real-world vulnerabilities and works with negligible overhead.

3.1 Background

3.1.1 Intel VT-x Extension

Hardware virtualization extensions, Intel VT-x and AMD-V for instance, enable almost all instructions of a guest VM to run natively on host CPUs. Many current hypervisors are implemented with hardware virtualization extensions. For example, KVM and Xen Hardware-assisted Virtual Machine (HVM) make use of the virtualization extensions. The rest of this section explains how an instruction emulator is invoked inside the hypervisor, targeting on Intel CPU with virtual-

ization support (VT-x).

In Intel VT-x, two execution modes, the root and non-root modes are added. Hypervisor code runs in the root mode, whereas a guest VM code runs in the non-root mode. Both the root and non-root modes have traditional execution modes (i.e. real mode and protected mode) and privilege levels (i.e. ring protections). Therefore, guest VMs in the non-root mode can use any of the execution modes and the privilege levels without any support from the hypervisor. Whenever some support is necessary from the hypervisor, the control is transferred from a guest VM to the hypervisor, called "VMExit", changing the CPU mode from the non-root mode to the root mode.

Once a VMExit occurs, the reason of VMExit is written in a Virtual Machine Control Structure (VMCS) by the hardware. The VMCS is a key virtualization structure in memory that consists of several fields, for example, the guest or host state fields, control fields, and VMExit information fields. The hypervisor can control VM state and settings through writing to the VMCS and get information about VM state from the VMCS. On the VMExit, a handler dedicated to each VMExit reason is invoked to emulate virtualized hardware.

A VMExit occurs, for instance, when a guest VM attempts to execute the `cpuid` instruction. A host CPU cannot execute the `cpuid` instruction in a guest VM because it should return the VCPU ID instead of the physical CPU's. Other system instructions, for example, accesses to the CRx, GDTR, LDTR, and MSR registers cause VMExits.

3.1.2 Instruction Emulation in Hypervisors

Some instructions executed in a guest VM must be emulated in the hypervisor [7, 11] although most instructions execute natively on host CPUs. For example, if an MMIO region is accessed, the hypervisor must intercept the I/O operation and emulate it. Because VT-x does not virtualize devices and thus the issued I/O operation cannot be executed natively on the physical device.

To trap access to an MMIO region from a guest VM, the hypervisor sets all MMIO regions inaccessible from every guest VM. A VMExit is caused with EPT violation (illegal memory access) as the VMExit reason when a guest VM accesses an MMIO region. The hypervisor analyzes the faulting address to determine

whether the access is caused by the access to an MMIO region.

Then, the hypervisor fetches the instruction that accessed to the MMIO region. The instruction emulator decodes and partially emulates the fetched instruction to recognize its operand. According to the operand, a device emulator such as QEMU is invoked.

Instruction emulation is not limited to the access to MMIO regions. The contexts that must be emulated are not only in the case of accessing an MMIO region. The hypervisor emulates instructions in the following six contexts.

- **Port I/O (PIO) context:** The hypervisor emulates an instruction that performs PIO. PIO is as an interface to interact with devices and accessed through `in` or `out` instructions are used to perform PIO. Executing `in` or `out` instructions in guest VMs incurs VMExit and these instructions are emulated by the hypervisor.

- **Memory Mapped I/O (MMIO) context:** The access to an MMIO region must be emulated for device emulation. MMIO is an interface to interact with devices through system memory. The memory and registers of devices are mapped to system memory so that CPUs can access devices by the same instructions that are used to access system memory. The hypervisor traps and emulates MMIO operations by making MMIO regions inaccessible.

- **Shadow Page Tables context:** The hypervisor needs to emulate an instruction that updates a guest page table to keep the consistency between guest and host (shadow) page tables. Prior to Nehalem micro-architecture, Intel CPUs did not support the second-level address translation. The hypervisor uses shadow page tables to translate guest virtual addresses into host physical addresses. The hypervisor traps and emulates an instruction that writes to a guest page table, and updates shadow page tables to keep the consistency.

- **Real Mode context:** Prior to Westmere micro-architecture, all guest instructions in real-mode must be emulated by the hypervisor. Intel CPUs prevent real-mode code from running in guest-mode. CPUs boot in real-mode and thus the hypervisor emulates all the instructions until they enter

protected-mode.

- **Migration context:** To allow VM migration between different vendor CPUs (Intel and AMD), vendor-specific instructions must be emulated if the VCPU's vendor differs from the physical CPU's. For example, `vmcall` and `vmmcall` are specific to Intel and AMD respectively. Both of them invoke hypercall that hypervisors prepared for paravirtualization. Fast control transfer instructions such as `sysenter`, `sysexit`, `syscall` and `sysret` are vendor-specific. Intel CPUs do not support `syscall`/`sysret` instructions for 32-bit kernels and also AMD CPUs do not support `sysenter`/`sysexit` instructions for 64-bit kernels. The hypervisor reports that the VCPU supports vendor-specific instructions to use them even if they are not supported by the physical CPU. If the migrated VM execute vendor-specific instructions not supported by the physical CPU, the physical CPU throws an illegal instruction exception. Then, the hypervisor traps illegal instruction exceptions and emulates vendor-specific instructions.

- **User-Mode Instruction Prevention (UMIP) context:** UMIP is a security feature of Intel processors to prevent unprivileged code from reading system-wide settings such as the physical address to an interrupt vector table (interrupt descriptor table in Intel). More concretely, UMIP prevents execution of `sgdt`, `sidt`, `sldt`, `smsw`, and `str` instructions at unprivileged level [36]. To emulate UMIP on legacy CPUs not supporting it, the hypervisor traps and emulates them.

The new hardware virtualization extensions obviate the need for instruction emulation in some contexts. We describe the detail of eliminated contexts in Section 3.1.3.

3.1.3 Evolution of Intel VT-x

Intel VT-x has evolved since the first introduction on Pentium 4 in 2005. While six contexts require instruction emulation in Intel Pentium 4, the hypervisor on the most recent micro-architecture has to support three contexts. Figure 3.1 shows

the evolution of Intel VT-x and the relationship between CPU features and emulation contexts over time. Since the new features of VT-x allow guest VMs in Real-Mode and Shadow Page Table contexts to execute instructions natively, these contexts no longer require the instruction emulation. These features have been enabled by default in popular hypervisors such as KVM and Xen for ten years [43, 44]

In Nehalem micro-architecture, the extended page table (EPT) was introduced as second-level address translation. The hypervisor does not need to perform instruction emulation in the shadow page table context if the EPT is enabled. The EPT holds translations of guest physical address to host physical address and the hypervisor maintains EPTs instead of shadow page tables. Therefore, the hypervisor does not need to trap and emulate instructions that modify guest page tables because the hypervisor does not need to monitor the updates to guest page tables.

In Westmere micro-architecture, "Unrestricted Guest" feature was introduced. This feature enables guest VMs to run real-mode code in guest-mode. The emulation in real-mode context has not been necessary anymore.

The instruction emulator still supports many instructions for backward compatibility while new features of VT-x obviate the need for instruction emulation in Real-Mode and Shadow page Table contexts. The current hypervisors invoke the instruction emulator in those contexts if the host uses legacy CPUs; they do not support EPT or unrestricted guest. A cross-modifying code attack that enables attackers to force the emulation of arbitrary instructions to exploit its emulation exists [7]. As a result, in spite of the new features of VT-x, the attack surface in the instruction emulator is still large.

3.2 Threat Model and Vulnerability Analysis

3.2.1 Threat Model

Before describing the threat model, this section explains the detail of a cross-modifying code attack. This attack is necessary to exploit a wide range of vulnerable instructions. To exploit the instruction emulator, an attacker has to force the hypervisor to perform emulation of a vulnerable instruction. However, as

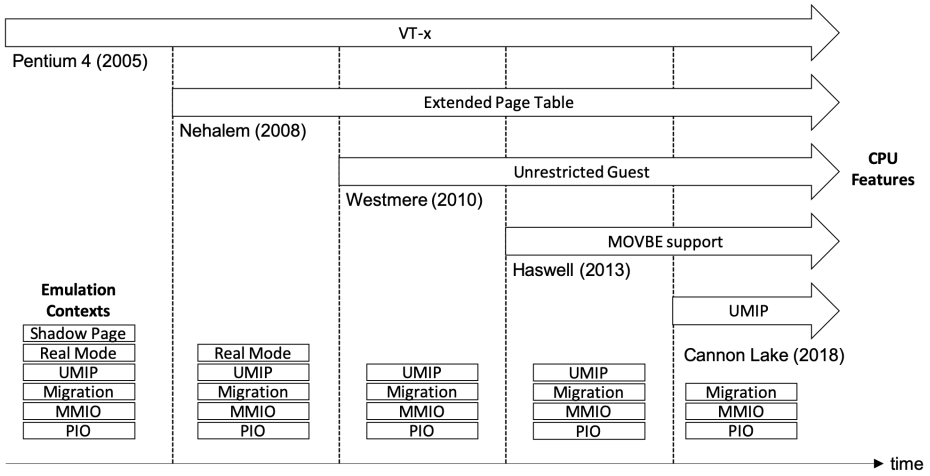


Figure 3.1. Evolution of Intel VT-x and corresponding emulation contexts over time.

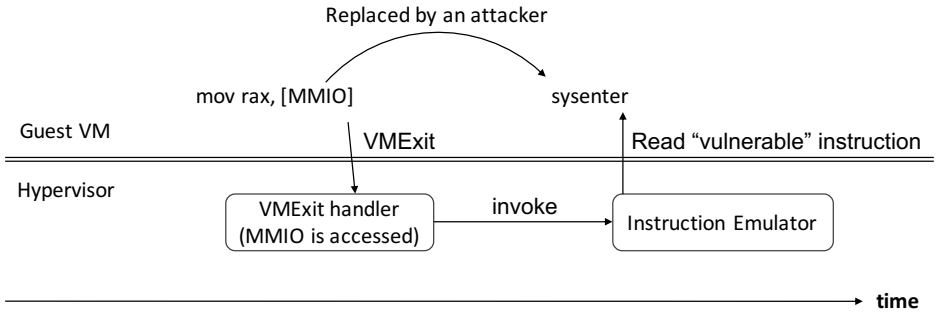


Figure 3.2. Timing Attack on Instruction Emulation.

described in Section 3.1.2, the contexts of the instruction emulator invoked are limited.

At first glance, an attacker appears unable to exploit a vulnerable instruction if it does not cause any VMExit because the emulator is not invoked. Suppose that an attacker is trying to exploit a vulnerability in the emulation of `sysenter` instruction (CVE-2015-0239). When `sysenter` is executed on Intel x86, it does not cause any VMExits and thus the emulator is not invoked. Interestingly, Amit et al. [7] have shown the cross-modifying code attack to force the emulator to decode whichever instruction the attacker wants to exploit. This attack is a timing attack and exploits a short time interval between the VMExit and the emu-

lator invocation. In Figure 3.2, an attacker accesses an MMIO region to cause a VMExit, and quickly replaces the accessing instruction with a vulnerable instruction (`sysenter`). If the replacement finishes before the VMExiting instruction (`mov`) is fetched, the emulator fetches and decodes the vulnerable instruction.

Our threat model is as follows. We assume that a guest operating system is not trustworthy; it may have security holes and be subverted by an attacker. Together with the attack vector shown by Amit et al., this assumption implies that an attacker can force any instruction to be emulated through an MMIO region. Note that an attack on the instruction emulator is sometimes possible from the user-space. Recent Linux allows a small portion of the MMIO region to be exposed to user-space; HPET (High Precision Event Timer) can be configured to be exposed to user-space in Linux.

3.2.2 Vulnerability Analysis

As described in Section 3.1.3, the emulator in the hypervisor supports many instructions for backward compatibility. The complexity of x86 instruction set leads to vulnerabilities in the emulator. In particular, instructions rarely used in modern environments are not tested and maintained well and are likely to be vulnerable. CVE-2015-0239 reports a vulnerability in the emulation of `sysenter` in 16-bit mode, which results in the privilege escalation. CVE-2016-9756 reports vulnerabilities in the emulation of `far jump` and `far ret` in 32-bit mode, which leads to the leak of the host kernel stack. More vulnerabilities are reported; CVE-2017-2584, CVE-2017-2583, CVE-2014-8480, CVE-2014-3647, CVE-2016-8630, and CVE-2014-8481 are all related to vulnerabilities in the emulator.

The goal of *FWinst* is to narrow an attack surface against vulnerabilities in instruction emulation. Our insight behind *FWinst* is twofold. First, emulation of most instructions is required for backward compatibility. If the hypervisor runs on CPUs with full-fledged support for virtualization, the number of *emulation contexts* that require instruction emulation becomes much smaller. While the hypervisor on legacy x86 micro-architectures must support 6 emulation contexts, the hypervisor on recent micro-architectures has to support only 3 contexts: 1) Port I/O, 2) MMIO, and 3) Migration. Emulation in Real-Mode, Shadow Page Table, and UMIP is not necessary in recent micro-architectures because real-mode

in guest-mode is allowed, EPT (extended page table) is supported for second level address translation, and guest VMs can leverage UMIP without VMExiting.

Second, a *legitimate* subset of instructions is very limited that is allowed to be emulated in each emulation context; arbitrary instructions should be emulated in every emulation context. For example, an MMIO region is accessed only by memory-accessing instructions; it is not legitimate to jump into an MMIO region or to invoke `sysenter` on an MMIO region. If the instructions not legitimate in the current emulation context are filtered out, the attack surface is narrowed; an attacker can exploit a vulnerability in the instructions that are legitimate in the current context.

By narrowing the attack surface, *FWinst* is expected to prevent an attacker from exploiting vulnerabilities in instruction emulation. Since only the memory-accessing instructions are legitimate in MMIO context, it is impossible to force the emulation of vulnerable `sysenter`, `far jump`, and `far ret` through the MMIO region. On recent micro-architectures, a legitimate set of instructions does not include legacy, rarely-used instructions. In addition, it would be easier to maintain the emulation code and verify its correctness because the number of legitimate instructions is much smaller than that of the entire instructions. This would enhance the overall safety of the instruction emulator.

3.3 Design and Implementation

The vulnerability analysis in Section 3.2 suggests the attack surface against the instruction emulator can be narrowed if the emulation context is taken into account. This section describes the design and implementation of *FWinst*, which filters out instructions that should not be emulated in the current emulation context.

3.3.1 Overall Architecture

Figure 3.3 b) illustrates the overall architecture of *FWinst*. *FWinst* resides in the hypervisor between VMExit handlers and the instruction emulator. When a VMExit handler is invoked and needs the instruction emulation, it invokes *FWinst* and passes it the VMExit reason. It tells the hypervisor what event has

Table 3.1. Summary of Emulation Contexts and Legitimate Set of Instructions.

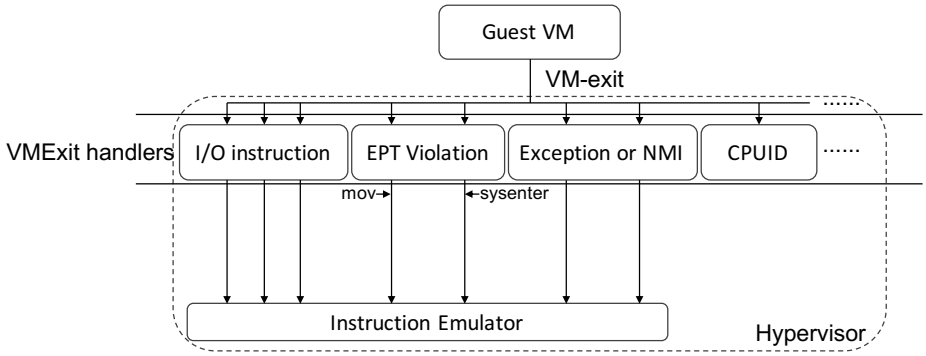
Emulation Context	Context Identification	Legitimate Instructions
PIO	I/O instruction	`in`, `out`
MMIO	EPT violation or EPT misconfig	`mov`, `movsx` `stosx`, `or`
Shadow page table	Exception or NMI (#PF)	memory access instructions
Real mode	VCPU status (No VMExit)	all real-mode instructions
Migration	Exception or NMI (#UD)	`vmcall`, `vmmcall` `syscall`, `sysenter` `sysexit`, `rsm`, `movbe`
UMIP	Access to GDTR or IDTR or Access to LDTR or TR	`sgdt`, `sidt`, `sldt` `smsw`, `str`

happened in the guest VM and provides a good clue to estimate the emulation context. If *FWinst* cannot determine the emulation context only from the VMExit reason, it collects more pieces of information from the internal states managed by the hypervisor.

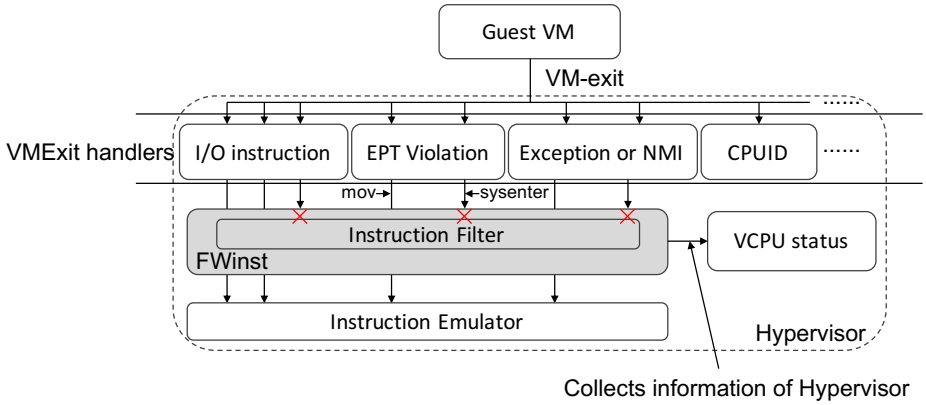
To determine which instruction should be emulated in each emulation context, *FWinst* maintains a list of legitimate instructions for each context. This list is constructed in advance. For some contexts, it is straightforward to define the legitimate set of instructions. For example, the legitimate instructions for Port I/O context are those in the family of `in` and `out` instructions, because I/O ports are accessed only through them. For other contexts, such as MMIO context, some engineering efforts are needed to determine the legitimate set. Section 3.3.3 describes the approach *FWinst* has taken to determine the legitimate set.

3.3.2 Identifying Emulation Contexts

Table 3.1 shows the summary of the emulation contexts identified in *FWinst*. *FWinst* identifies six contexts: 1) Port I/O, 2) MMIO, 3) shadow page table, 4) real



a) Ordinary Hypervisor. A VMExit handler invokes the instruction emulator regardless of the emulation context.



b) Hypervisor with *FWinst*. *FWinst* filters out instructions that should not be emulated in the current context.

Figure 3.3. Instruction Emulator in Ordinary Hypervisors and in Hypervisors with *FWinst*.

mode, 5) migration, and 6) UMIP.

Port I/O context. It can be identified directly from the VMExit reason. When a guest OS makes an access to an I/O port, it incurs a VMExit with the reason set to 'I/O instruction'. *FWinst* determines the current context is Port I/O from the VMExit Reason.

MMIO context. It is identified by confirming a VMExit occurs due to an access to an MMIO region. When a guest OS makes an access to an MMIO region, the faulting address is notified. *FWinst* confirms the faulting address fits in the MMIO region. The detailed behavior differs depending on the configuration of

the hypervisor. If the EPT feature is turned on, the VMExit reason is set to 'EPT Violation/Misconfiguration'. If the EPT feature is unavailable or turned off, the VMExit reason is set to 'Exception or Non-maskable interrupt (#PF)'. In both cases, if the faulting address resides in an MMIO region, *FWinst* concludes the context is MMIO, because there is no overlap between an MMIO region and guest page tables.

There are two things to be noted. First, when the memory-mapped APIC (Advanced Programmable Interrupt Controller) is accessed, an VMExit with 'APIC Access' occurs. In this case, *FWinst* concludes the current context is MMIO because this is the access to the APIC control registers using an MMIO interface. Second, the hypervisor sometimes — e.g., for host swapping — intentionally configures EPT entries or shadow page tables to cause VMExits on the access to a certain page. In this case, *FWinst* is not invoked because the hypervisor does not emulate any instructions. The hypervisor resolves the VMExits by loading memory pages and/or setting page tables properly.

Shadow page table context. If the EPT feature is not available, the shadow page table context is identified with the cooperation of the hypervisor. This context is identified by confirming a VMExit occurs due to an access to a guest page table. When a guest page table is accessed in the guest, a VMExit with the VMExit reason set to 'Exception or Non-maskable interrupt(#PF)' is incurred and the faulting address is notified to the hypervisor. The hypervisor keeps track of the addresses to guest page tables (stored in CR3) and thus can determine if the access is to a page table or not.

Real mode context. If the unrestricted guest mode is not available, the real-mode code is executed either in the virtual 8086 mode or on the emulator [43]. The hypervisor maintains a global state that tells whether the emulation for real-mode is required or not. By checking the status register (CR0 in this case), the hypervisor can know whether the VCPU is in real mode or not. Note that the instructions are not always emulated in real mode. KVM checks the VCPU status and lets the guest run in the virtual 8086 mode if possible. *FWinst* inquires of the hypervisor whether the emulation is necessary. *FWinst* checks the global state to determine the current emulation context.

Migration context. If an unsupported instruction is executed in a guest, a VMExit occurs with the reason set to 'Exception or Non-maskable interrupt

(#UD)'. Encountering this VMExit reason, *FWinst* concludes the current context is migration. At first glance, this strategy looks dangerous because all vendor-specific instructions are emulated without further inspection. Since the number of legitimate instructions is small in the migration context, *FWinst* checks which vendor-specific instruction is supported and rejects the emulation of the instructions natively supported because it is nonsense to emulate natively supported instructions. Note that *FWinst* does not confirm a virtual machine in question is actually migrated from another machine because a virtual machine image built for AMD, for instance, can be executed on Intel x86 without migration.

UMIP context. It can be identified directly from the VMExit reason. Executing the instructions covered by UMIP incurs a VMExit with the reason set to 'Access to GDTR or IDRT' or 'Access to LDTR or TR'. *FWinst* determines the current context is UMIP if these VMExit reasons are set in the VMCS.

Note that determining the emulation context is quite simple. Since the mapping between the emulation contexts and the reason the hypervisor is invoked is straightforward, we believe the possibility of making a mistake in determining a valid context is quite low. If there is a mistake in determining a valid context, it can lead to false-positive or -negative. A false-positive occurs if an incorrect context prevents the emulation of a legitimate instruction. A false-negative occurs if an incorrect context allows the emulation of an illegitimate instruction.

3.3.3 Legitimate Instructions

For each emulation context, a set of legitimate instructions are defined. Table 3.1 shows the summary of the legitimate set of instructions for each context.

For PIO and UMIP context, it is straightforward to define the sets. For PIO context, the family of `in` and `out` instructions are legitimate because I/O ports are accessed only through them. For UMIP context, `sgdt`, `sidt`, `sldt`, `smsw`, and `str` instructions are legitimate because these instructions are covered by UMIP [36].

For MMIO context and shadow page table context, the legitimate set of instructions is memory-accessing instructions; i.e. instructions having memory-access operands. By default, *FWinst* allows these instructions to be emulated in these contexts. If the operating systems hosted on the hypervisor are known in

Listing 3.1. Example of MMIO accessor in Linux kernel 4.8.1 arch/x86/include/asm/io.h line 46

```
#define build_mmio_read(name, size, type, \
                        reg, barrier) \
static inline type name( \
  const volatile void __iomem *addr) \
{ \
  type ret; \
  asm volatile("mov" size " %1,%0":reg (ret) \
               :"m" (*(volatile type __force *)addr) \
               barrier); \
  return ret; \
}
```

advance, their coding conventions can be leveraged to further restrict the legitimate set. For example, the hosted operating systems are known in advance in the PaaS (Platform-as-a-Service) environments.

For MMIO context, the legitimate set can be further restricted. An MMIO region is accessed solely by device drivers, and the operating systems provide accessor functions/macros to MMIO regions to encapsulate the coding difficulties in memory coherence such as barriers. Listing 3.1 and 3.2 exemplify the accessors in Linux and Windows, respectively. The legitimate set can be derived from memory-accessing instructions in the accessors. Our current prototype restricts the legitimate set in this way for Linux and Windows. This approach works well for drivers that use the accessors to access MMIO regions. In practice, driver developers use the accessors to avoid writing the code for complicated memory synchronization.

For the shadow page table context, all the functions that update page tables in the guest OS must be investigated. The legitimate set is memory-accessing instructions in those functions. Fortunately, the number of those functions is small. Linux has five functions that update page tables.

For the real mode context, it is almost impossible to define a small set of legitimate instructions because real-mode code can execute a bunch of instructions

Listing 3.2. Example of MMIO accessor in wdm.h line 17433 that is included in the Windows Driver Kit, build version 0162

```
__forceinline
UCHAR
READ_REGISTER_UCHAR (
    _In_ _Notliteral_ volatile UCHAR *Register
    )
{
    _ReadWriteBarrier();
    return *Register;
}
```

during the boot sequence. Currently, *FWinst* includes all the instructions valid in real mode in the legitimate set. To avoid attacks during the boot sequence, it is better to load a virtual machine image after the boot sequence (i.e., CPU in protected mode), which has been built in an isolated and secure environment.

For migration context, vendor-specific instructions must be emulated. KVM/QEMU lists up all the vendor-specific instructions: `vmcall`, `vmmcall`, `syscall`, `sysenter`, `sysexit`, `rsm`, and `movbe`. These instructions are included in the legitimate set for the migration context. Since it is nonsense to emulate natively supported instructions, the legitimate set excludes the instructions that are supported natively on the physical CPUs.

3.3.4 Implementation

A prototype of *FWinst* [1] has been implemented on Linux KVM (Linux kernel 4.8.1) for Intel x86-64 architecture. *FWinst* is implemented with 279 LoC and can be ported from Linux kernel 4.8.1 to 6.6.8 with one day of work by one person. We assume the micro-architectures posterior to Westmere, and the full-fledged features (EPT and unrestricted guest mode) for virtualization are enabled. Westmere was released around 2010 and thus, it is natural to assume Westmere micro-

[1]The *FWinst* prototype for Linux kernel 6.6.8 is available at `https://github.com/sslab-keio/FWinst`.

architecture or later. *FWinst* assumes Intel x86. Our description targets on KVM but *FWinst* can be applied to Xen or other hypervisors in principle.

The current prototype identifies PIO, MMIO, migration, and UMIP contexts. Shadow page table or real mode contexts are not recognized because the EPT feature and the unrestricted guest mode are enabled.

Building the Legitimate Instruction Set

For PIO and UMIP contexts, the legitimate set of instructions is straightforward, as described in Section 3.3.3.

For the migration context, our current prototype includes the instructions specific to AMD (`vmcall`) and those supported on later Intel micro-architectures(`movbe`). *FWinst* can recognize which vendor-specific instructions are supported because the hypervisor in which *FWinst* is running has an access to the model-specific register (MSR) that tells the CPU micro-architecture. As described in Section 3.3.2, the current implementation of *FWinst* does not confirm a virtual machine is actually migrated from another machine but this does not mean migration context is not handled; it rejects the emulation of vendor-specific instructions natively supported on the current micro-architecture.

For MMIO context, the legitimate set is restricted for Linux and Windows Vista, 7, 8, and 10. In the case of Linux, the MMIO region is accessed only through some macros and inline functions used in device drivers. From the compiled binary of the device drivers, we have extracted memory-accessing instructions and included them in the set. The resulting set of legitimate instructions includes only the instructions of the `mov` family. In the case of Windows, another approach has been taken because of the unavailability of the source code. The legitimate set is extracted from the log of instructions executed during the kernel launch and shutdown times. Since device drivers are loaded at the launch time and unloaded at the shutdown time, this log covers the memory-accessing instructions used to access to MMIO regions. The resulting set of legitimate instructions includes the `mov` family of instructions.

There is a subtle problem in MMIO context. During the boot sequence, BIOS makes an access to an MMIO region. Therefore, the legitimate set for MMIO context has to include instructions used by BIOS to access to an MMIO region. To

extract those instructions we again relied on the execution log. BIOS uses `movs` and `stos` instructions to access to the MMIO region and thus, those instructions have been added to the legitimate set. The MMIO-accessing instructions solely used by BIOS can be excluded from the legitimate set after the boot sequence. Since the BIOS can be accessed only in some CPU modes, *FWinst* can remove those instructions from the legitimate set when the CPU is not in those CPU modes. Or they can be entirely removed if we can assume the guest VM always loads a booted VM image.

Implementation of the Instruction Filter

The control and data flow between the components of the instruction emulator are depicted in Figure 3.4. Solid lines show the control flow and dotted lines show the data flow. KVM instruction emulator consists of three major components: 1) opcode decoder, 2) operand decoder, and 3) emulator. When KVM emulates an instruction, these three components are invoked in this order. We add *FWinst* (depicted as a gray box) as a new component to the instruction emulator. *FWinst* filters improper instructions according to the emulation context and the legitimate set.

When a VMExit occurs, the VMExit handler determines the emulation context according to the VMExit reason with the support from the hypervisor. If the emulation is necessary, it invokes the opcode decoder. The opcode decoder fetches the instruction to be emulated from the guest memory and stores it in a memory area for the emulation that is accessible only from the inside of KVM. After decoding the opcode, it invokes *FWinst* with the emulation context passed to *FWinst*. *FWinst* gets the instruction to be emulated. If it is not included in the legitimate set, *FWinst* filters out the instruction. If it is included in the set, *FWinst* invokes the operand decoder.

To filter the instruction before emulating, *FWinst* needs only the opcode of an emulated instruction in the contexts. To avoid duplicated implementation of opcode decoders, *FWinst* lets the instruction emulator decode each instruction. This design allows us to reuse the opcode decoder and releases us from maintaining two decoders (the one in the instruction emulator and the other in *FWinst*). Note that *FWinst* does *not* rely on the operand decoder, which is more com-

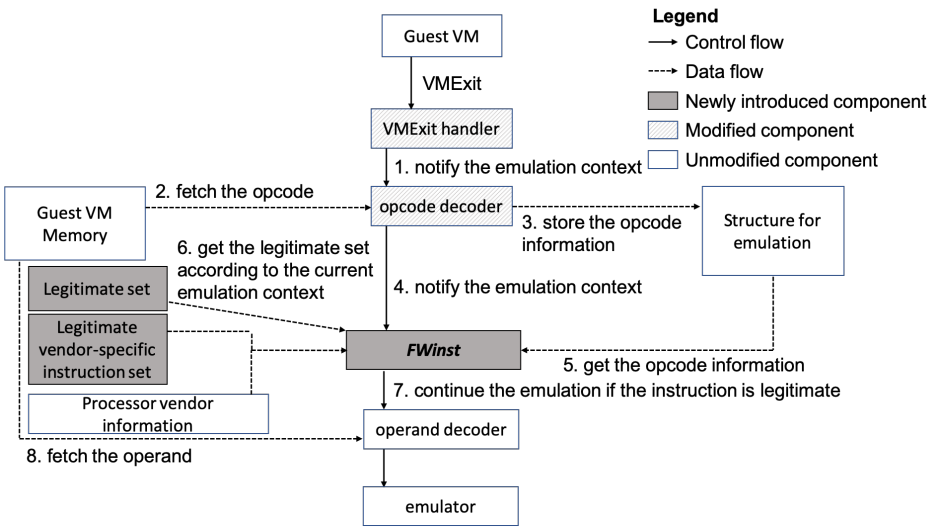


Figure 3.4. The control and data flow between the components of the instruction emulator in KVM with *FWinst*.

plicated and more vulnerable than the opcode decoder. The operand decoder has 665 LOC, whereas the opcode decoder has 279 LOC. Three vulnerabilities (CVE-2016-8630, CVE-2014-8481, and CVE-2014-8480) in the operand decoder have been reported whereas one vulnerability (CVE-2009-4031) in the opcode decoder has been reported. Even if there is a vulnerability in the operand decoder, *FWinst* works properly.

3.4 Experiments

To demonstrate the effectiveness of *FWinst*, we have implemented a prototype of *FWinst* on Intel x86 Skylake and Westmere micro-architectures. In the following analysis and experiments, all the CPU support for virtualization is turned on; i.e. EPT and the unrestricted guest mode are both turned on. Table 3.2 shows the experimental environment.

3.4.1 Security Analysis

To demonstrate the effectiveness of *FWinst*, we have investigated 110 vulnerability reports from 2009 to 2018 that are related to KVM and found that 17 vul-

Table 3.2. Experimental Environment for *FWinst*

Hardware for Skylake	
Host CPU	Intel Xeon Silver 4110 2.10GHz (Skylake)
Host memory	32 GB
Hardware for Westmere	
Host CPU	Intel Xeon X5650 2.67GHz (Westmere)
Host memory	4 GB
Software for both	
Host OS	Linux kernel 4.8.1
Host QEMU	Version 2.9.50
Guest OS	Ubuntu 18.04 x86_64
# of VCPUs	2
Guest memory	1 GB
Virtual drive	IDE
Virtual graphics card	VGA standard
Virtual network interface card	e1000

nerabilities reside in the instruction emulator, which are listed in Table 3.3. For these vulnerabilities we have collected or implemented PoC (proof-of-concept) code and tested it on *FWinst*. As shown in Table 3.3, *FWinst* can defend against 14 vulnerabilities out of 17 on Haswell, (indicated by $\surd$ in column 'Haswell'), 13 vulnerabilities out of 17 on Westmere (indicated by $\surd$ in column 'Westmere'), and 13 out of 17 on AMD Jaguar (indicated by $\surd$ in column 'AMD' and 'AMD' refers to AMD Jaguar in the rest of this section). Since *FWinst* can filter out more instructions in more recent micro-architectures, Haswell prevents more attacks than Westmere.

The column 'emu. context' denotes the emulation contexts whose legitimate sets of instructions include vulnerable instructions listed in 'vul. inst'. The vulnerable instructions cannot be exploited on micro-architectures in which the contexts in 'emu. context' would not be effective. 'None' in the context col-

Table 3.3. Summary of vulnerabilities.

CVE #	vulnerable instruction	Intel Westmere	Intel Haswell	AMD	vul. comp.	emu. context
2018-10853	`fxrstor`, `fxsave`, `sgdt`, `sidt`	√	√	√	emu.	UMIP, Real Mode
2017-17741	`vmmcall`, `vmcall`	d	d	d	emu.	Migration
2017-7518	`syscall`	√	√	√	emu.	Migration
2017-2584	`fxrstor`, `fxsave`, `sgdt`, `sidt`	×	×	×	emu.	UMIP, Real Mode
2017-2583	`mov SS`	√	√	√	emu.	Real Mode
2016-9756	`far jump` or `far ret`	√	√	√	emu.	Real Mode
2016-8630	illegal instruction	√	√	√	operand	None
2015-0239	`sysenter`	√	√	d	emu.	Migration
2014-8481	`movbe`	d	√	√	operand	Migration, Real Mode
2014-8480	`clflush`, `hint-nop`, `prefetch`	√	√	√	operand	Real Mode
2014-7842	unsupported instructions by the instruction emulator	√	√	√	emu.	None
2014-3647	`far jump` or `far ret`	√	√	√	emu.	Real Mode
2014-0049	`pusha`	√	√	√	emu.	Real Mode
2012-0045	`syscall`	√	√	√	emu.	Migration
2010-5313	unsupported instructions by the instruction emulator	√	√	√	emu.	None
2010-0435	`mov DR`	√	√	√	emu.	Real Mode
2009-4031	instruction that contains too many bytes	×	×	×	opcode	All

√: defended, ×: not defended, d: depends on migration contexts, emu.: emulation, operand: operand decoder

umn in Table 3.3 means the vulnerable instructions should never be emulated in any context. The discussion below follows the contexts listed in 'emu. context'.

Real Mode context only: In CVE-2017-2583, CVE-2016-9756, CVE-2014-8480, CVE-2014-3647, CVE-2014-0049 and CVE-2010-0435, vulnerable instructions are included only in the legitimate set of Real Mode context. The 'unrestricted guest mode' is turned on to natively execute real-mode instructions on all the tested machines. Therefore, Real Mode context would not be effective and these vulnerable instructions are not emulated at all.

In CVE-2017-2583 and CVE-2010-0435, the vulnerable instructions have memory-accessing operands. As described in Section 3.3.3, the instructions that have memory-accessing operands should be included in the legitimate set of MMIO context. `Mov SS` in CVE-2017-2583 loads or stores the stack segment register, and `mov dr` in CVE-2010-0435 loads or stores the debug registers.

These instructions are excluded from the legitimate set of MMIO context because they are not used to access to MMIO regions.

Migration context only: In CVE-2017-17741, CVE-2017-7518, CVE-2015-0239 and CVE-2012-0045, vulnerable instructions are included only in the legitimate set of Migration context. All the vulnerable instructions here are vendor-specific. In CVE-2017-17741, the vulnerable instructions are Intel-specific `vmcall` and AMD-specific `vmmcall`. If these instructions are requested to be emulated on the machines that can natively execute them, *FWinst* rejects the emulation and can prevent the attack. If the requesting guest is migrated from another machine and the running binary is for a different vendor, *FWinst* considers the emulation request is legitimate and cannot prevent the attack. Thus, all the columns for CVE-2017-17741 are marked as 'depend'.

In CVE-2015-0239, the vulnerable instruction is `sysenter`, an Intel-specific instruction, which would not be emulated on Intel machines. On AMD machines, this instruction is emulated only if a guest running the Intel binary is migrated from another machine. Thus, the columns except for AMD are marked as '$\surd$', and the column for AMD is marked as 'd'.

In CVE-2017-7518 and CVE-2012-0045, the vulnerable instruction is `syscall`. This instruction is not implemented only on the 32-bit version of Intel x86; the micro-architectures listed in the table all support `syscall` and thus, this instruction will not be emulated.

Migration and Real Mode contexts: In CVE-2014-8481, the vulnerable instruction is included in both Migration and Real Mode contexts. Since the 'unrestricted guest mode' is effective on all the tested machines, Real Mode context would not be effective and the vulnerable instruction would be emulated only in Migration context.

CVE-2014-8481 is marked as 'depends' in Intel Westmere, and the vulnerable instruction is `movbe` introduced in Intel Haswell or AMD Jaguar. If *FWinst* recognizes a guest is running binary for Westmere, *FWinst* rejects the emulation of `movbe` because it is strange that Westmere binary is executing unsupported `movbe`. But if the guest is migrated from another machine and runs binary for Haswell, *FWinst* emulates `movbe` on Westmere; the vulnerability can be exploited.

UMIP and Real Mode contexts: In CVE-2018-10853 and CVE-2017-2584,

the vulnerable instructions are included in either UMIP or Real Mode contexts. Instructions `fxrstor` and `fxsave` are included only in Real Mode context and thus are not emulated in our testbeds. Vulnerable instructions `sgdt` and `sidt` are included in UMIP context. If the guest is running in UMIP context and the vulnerabilities are exploited, *FWinst* cannot defend against it. Because of this, all the columns of CVE-2017-2584 are marked as '×'.

Interestingly, the vulnerability pointed out in CVE-2018-10853, which launches privilege escalation from non-root/ring3 to non-root/ring0, can be prevented. Privilege escalation to non-root/ring0 is meaningful only if a user-level process launches an attack. Fortunately, the current implementation of the instruction emulator rejects `sgdt` and `sidt` when they are issued at user-level, if UMIP is turned on, because UMIP does not allow the execution of those instructions at user-level. As the result, the privilege escalation is unsuccessful even though *FWinst* does not filter out vulnerable instructions `sgdt` and `sidt`.

All contexts: CVE-2009-4031 is marked as '×' in all columns. This vulnerability lies in the opcode decoder and can be exploited when the opcode length exceeds the maximum length (15 bytes). *FWinst* cannot defend against this vulnerability because *FWinst* reuses the KVM opcode decoder. This vulnerability is exceptional in that it lies in the opcode decoder. As Table 3.3 indicates, most vulnerabilities lie in the operand decoder or the emulator. Checking the opcode length suffices to defend against this vulnerability and thus we have already extended *FWinst* to have this verification phase before the opcode decoding.

Not in any contexts: In CVE-2016-8630, CVE-2014-7842, and CVE-2010-5313, the vulnerable instructions are not included in any contexts and thus, *FWinst* can defend against attacks that attempt to exploit these vulnerabilities.

3.4.2 Runtime Overhead

To estimate runtime overhead introduced by *FWinst*, we measured the runtime overhead of several benchmarks. Our machine environment and its configuration are given in Table 3.2. We prepared a micro-benchmark that accesses to an MMIO region repeatedly to show the overhead of *FWinst*; *FWinst* is invoked every time an MMIO region is accessed by a guest VM. For macro-benchmarks, UnixBench [98], sysbench [53], ApacheBench [10], and Phoronix Test Suite [76]

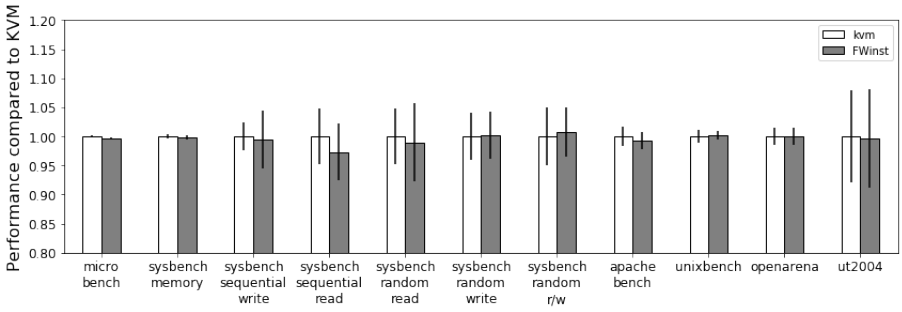


Figure 3.5. Normalized performance of UnixBench, Apache Bench, sysbench, micro benchmark and graphic benchmarks on Skylake with the original KVM as the baseline

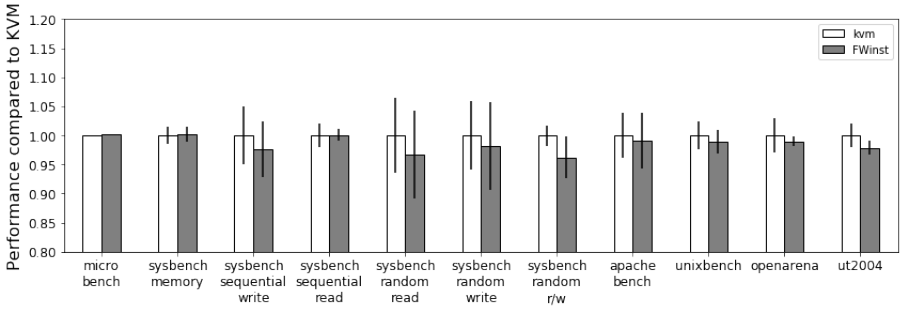


Figure 3.6. Normalized performance of UnixBench, Apache Bench, sysbench, micro benchmark and graphic benchmarks on Westmere with the original KVM as the baseline

are used. UnixBench and sysbench are standard benchmarks to measure the performance of the operating system. ApacheBench is chosen for the server workloads and Phoronix Test Suit is for graphics-intensive workloads. OpenArena and Unreal Tournament 2004 (UT2004), chosen from Phoronix Test Suits, execute OpenGL 3-D games.

Figure 3.5 and 3.6 show the relative performance on Skylake and Westmere, respectively. The overhead of *FWinst* on Skylake is from 0.0 % to 2.7 % and the highest overhead benchmark is the sequential read of sysbench. In the case of Westmere the overhead of *FWinst* is from 0.0 % to 3.8 %. *FWinst* causes relatively low overheads for the following reasons. First, the overhead due to *FWinst* is caused when an instruction emulator is invoked. Recent advance in hardware

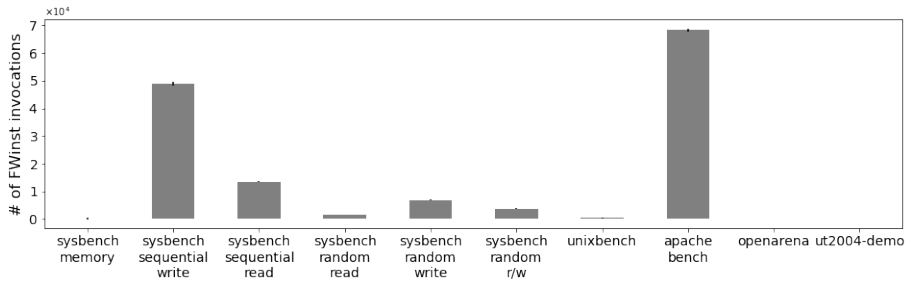


Figure 3.7. # of *FWinst* invocations

virtualization reduces the number of instructions that should be emulated and thus the overhead is getting lower. Second, instruction emulation in our experiments is primarily for the device emulation. KVM emulates device at userspace and thus the relative overhead of *FWinst* becomes very low.

The number of *FWinst* invocations in one second for each benchmark is shown in Figure 3.7. This result shows that the most cause of instruction emulation is I/O operation, because *FWinst* is invoked many times in I/O intensive workloads except graphics benchmarks. Since the guest OS in these workloads performs a lot of I/O operations, the instruction emulator must emulate I/O instructions frequently. Hence, *FWinst* must verify every emulated instruction and *FWinst* is also frequently invoked. Although graphics-intensive workloads are I/O intensive workloads, the number of *FWinst* invocations is not high. There are two reasons as follows. First, the graphics library in this guest environment uses LLVMpipe [63] as a 3D graphics driver and it performs all rendering on the CPU. This eliminates the need for emulating I/O instructions. Another reason is that the emulation of I/O operations is not necessary when the guest OS updates its video ram. Because, in the implementation of virtual VGA in QEMU, KVM and QEMU enable the guest OS to write directly to its video ram for performance reason and the VMExit never occurs writing to its video ram.

3.5 Related work

3.5.1 Protecting Virtual Machines

Initially, hypervisors were believed to be trustworthy because of their small code base and narrow interfaces. By leveraging this characteristic, security systems with virtualization had been extensively studied [84, 85, 78, 102, 88, 39, 40]. However, commodity hypervisors are far from without security concerns due to their complexity and large code base. Attackers shifted their focus from breaking into individual OSes to compromising entire virtual environments [52, 4, 54, 106].

The primary goal of virtual environment security is to ensure security of VMs so that providing security for VMs, even if the underlying hypervisor is compromised or untrusted, is one of the research areas. To protect VMs from untrusted hypervisors, several works eliminate a large part of hypervisors from their trusted computing base (TCB) by leveraging architectural support. For example, H-SVM [41] modifies hardware to protect the memory of VMs from unauthorized access with an untrusted hypervisor. H-SVM intercepts access to nested page tables and manages its ownership. This improves memory isolation among the VMs by blocking the hypervisor's direct modifications of nested page tables. HyperWall [93, 94] leverages the hardware-only accessible memory regions to protect the Confidentiality and Integrity Protection (CIP) tables. The CIP tables hold the information on memory access rights based on the customer's specifications. The memory of the VMs is not accessible by the hypervisor without permission from HyperWall. HyperCoffer [111] only trusts the processor chip to protect VMs from an untrusted hypervisor. The secure processor chip provides memory encryption and integrity checking. However, applying the secure processor alone is insufficient because the semantic gap between the VMs and the hypervisor exists. To bridge the semantic gap, HyperCoffer provides a mechanism called VM-Shim, which helps to exchange data between the VM and the hypervisor while interposing the control transition between them. Besides these approaches, using trusted execution environments deployed in commodity hardware, e.g., Intel SGX [37] or AMD SEV [5], is another approach to protect applications or VMs from untrusted hypervisors. Haven [14], SCONE [12], and Graphene-SGX [18] use SGX to defend applications from software and hardware

attacks, including attacks by untrusted hypervisors. Fidelius [109] is a software-based extension to SEV to address resource encryption issues. Although these previous approaches are attractive because the VMs do not need to trust their underlying hypervisor, using specific hardware is not always feasible for virtual environments because of their various demands. Thus, enhancing virtual environment security through a software-based approach remains necessary to make hypervisors trustworthy.

Deprivileging hypervisors has also been widely studied to reduce TCB of hypervisors. To deprivilege the parts of hypervisors, a couple of works run most hypervisor functionality at a less privilege mode. For example, NOVA [90] designs a new hypervisor in the manner of microkernel from a clean slate. Thanks to the architecture, NOVA only needs kernel mode to run microhypervisor, and the rest of components (e.g., root partition manager and multiple VM monitors) runs at user mode. DuVisor [19] is a user-space hypervisor to reduce TCB. DuVisor directly interacts with VMs at runtime instead of the traditional KVM by leveraging a hardware virtualization extension that securely exposes hardware virtualization interfaces. HypSec [57] splits a monolithic hypervisor into two parts: a trusted an privileged corevisor with full access to VM data, and an untrusted and deprivileged hostvisor delegated with most hypervisor functionality. HypSec leverages hardware virtualization support to isolate and protect the corevisor and execute it at a higher privilege level than the hostvisor. DeHype [107] is a system that applies the least privilege principle to hosted hypervisors. In DeHype, a minimal subset of privileged hypervisor code is responsible for executing instructions in privileged mode when the deprivileged hypervisor demands to issue a privileged instruction. Nested virtualization is also used to deprivilege hypervisors because it allows hypervisors to run in non-root mode. By leveraging nested virtualization, CloudVisor [112] puts Xen and Dom0 in non-root mode so that all privileged operations will trap to Cloudvisor for security checking. CloudVisor-D [67] extends CloudVisor to reduce heavy context switches by leveraging extended page table switching with the VMFUNC instruction. In-kernel isolation is another approach to deprivilege most functionalities of hypervisors. For example, HyperLock [103] provides a secure hypervisor isolation runtime to isolate hypervisors from compromising host OSes by enforcing memory and instruction access control. HyperLock also creates a

shadow hypervisor and pairs it with each VM to limit the negative impact of the compromised hypervisor to only the paired VM. NEXEN [87] decomposes the monolithic Xen into a minimal fully privileged security monitor, a less privileged shared service domain, and fully sandboxed Xen slices. NEXEN uses the Nested Kernel [23, 24] architecture to isolate these components at a single privilege level so that NEXEN can suppress performance overhead due to frequent context switches. NoHype [47, 48, 92] eliminates the virtualization layer rather than deprivileging the hypervisor while it preserves the semantics of virtualization technology. In conclusion, removing a large part of the hypervisor from the TCB improves the virtual environment security. However, in practice, the exposed attack surface of the trusted components increases with vulnerabilities in the deprivileged hypervisor. So, deprivileging and hardening the hypervisor should be done simultaneously.

A control VM such as Xen Dom0 is included in the TCB as well as the rest parts of hypervisors. Hardening the control VM is important to enhance VM security. For example, Xoar [22] focuses on the large code base of the control VM and breaks it into nine classes of service VMs. Each service VM has a single purpose so that Xoar can remove the original monolithic control VM. Several works present the need to differentiate between cloud service providers and cloud system administrators. Butt et al. [16] pointed out cloud system administrators are adversarial, and it resulted in that administrative domain is untrusted. SSC [16] splits Dom0 between a system-wide domain (SDom0) and per-client administrative domains (Udom0s). Disaggregated Xen [68] decouples all the code for building a VM from Dom0. MyCloud [56] removes the control VM from root mode and provides isolation with EPT. These works reduce the TCB of the virtual environment by reconstructing the control VM. These security mechanisms for the control virtual machine are complementary to enhancing CPU virtualization security.

3.5.2 Hardening Hypervisors

In this dissertation, *FWinst* hardens the hypervisor by reducing the attack surface of the CPU emulator of the hypervisor. Like *FWinst*, hardening hypervisors has been studied to enhance virtual environment security. For example, Hyper-

Safe [101] provides hypervisor control-flow integrity by managing page table updates strictly and restricting indirect control transfer. SeKVM [58] is a layered Linux KVM hypervisor that is formally verified. To minimize the effort of verification, the TCB of the hypervisor needs to be small. Thus, SeKVM requires the reconstruction of KVM/Arm and 15K LOC modification. Risotto [34] eliminates errors in CPU emulation involving concurrent memory access through formal verification. Hyper-TP [69] and VM-PHU [79] leverage kernel soft reboot and migration to address security issues of hypervisors at data center scale. These previous works complement *FWinst*; hardening hypervisors with multiple aspects effectively improves the overall hypervisor security.

Continuously monitoring a hypervisor is a way to improve hypervisor security. However, monitoring the hypervisor is challenging because the hypervisor runs with a high privilege to virtualize hardware. HyperCheck [113], HyperSentry [13], and MGuard [62] monitor the hypervisors by leveraging systems with higher privileges than the hypervisor to overcome the challenge. For example, HyperCheck and HyperSentry leverage the CPU system management mode presented in x86. MGuard monitors the hypervisor with their new programmable hardware. Deng et al. [25] place a trusted event-driven monitor at the same privilege level and in the same address space as the untrusted hypervisor. While hypervisor monitoring protects the whole system against compromised hypervisors, *FWinst* reduces the attack surface of the hypervisor to raise the bar of compromising hypervisors.

Several works focus on security concerns associated with device emulation of hypervisors. Nioh [71] and Nioh-PT [83] provide runtime protection for hypervisors from attacks against virtual devices. The key insight of these works is that the attacks against virtual devices are typically performed through illegal I/O requests that are not issued for devices during normal operations. They protect device emulators of hypervisors by filtering out the illegal I/O requests. Min-V [70] focuses on the fact that VMs use limited virtual devices at runtime in cloud environments. To reduce the attack surface of the hypervisor, Min-V analyzes which virtual devices are required only at boot time and eliminates them from VMs at runtime. Firecracker [1] and crosvm [32] re-build the part of KVM/QEMU for their specific purposes. Thanks to their slim designs, these works can reduce the attack surfaces of the hypervisor. These I/O virtualization

protection systems and *FWinst* should be utilized simultaneously because both CPU and I/O virtualization are essential parts of hardware virtualization.

3.5.3 Hypervisor Testing

Testing software like fuzzing or symbolic execution is used to discover bugs and vulnerabilities in a wide range of software. Targets of these approaches have been extended to complicated systems such as hypervisors. Since most fuzzing toolchains are implemented to test applications running in user mode, ring-3 I/O virtualization bugs were the primary targets in the early stages of hypervisor testing, and they are still explored extensively [35, 82, 81, 75, 61, 17]. Despite technical complexity, several works tackle finding CPU virtualization bugs. To enhance vCPU security, Amit et al. [8] perform blackbox testing for vCPUs by taking advantage of Intel's testing facilities. PokeEMU [65] generates test cases for low-fidelity emulators, including CPU emulation of hypervisors, by using symbolic execution on high-fidelity emulators like Bochs. MultiNyx [29] is a symbolic execution framework to test hypervisors' CPU and memory virtualization. MultiNyx leverages the Bochs CPU emulator as an executable specification to model the semantics of complex instructions for virtualization. HyperFuzzer [31] is a hybrid fuzzer for vCPUs. To overcome the complexity and scalability issues of hypervisor testing, HyperFuzzer performs symbolic execution efficiently by leveraging hardware tracing and recovering some semantics from the recorded traces with their new technique, called Nimble Symbolic Execution. These works showed that the complex CPU emulation of the hypervisors contains uncovered bugs. To lower the risk of such bugs, *FWinst* limits the instructions to be emulated based on the observation that the evolution of the hardware virtualization extension has eliminated the need to emulate a large number of instructions.

3.6 Summary

This chapter describes a new approach to narrow the attack surface against vulnerabilities in the KVM instruction emulator if the underlying micro-architecture and the hypervisor configration are taken into account. *FWinst* identifies a legiti-

mate set of instructions by recognizing emulation contexts, and filters out illegitimate instructions, thereby narrowing the attack surface. Our preliminary evaluation shows *FWinst* effectively prevents emulator vulnerabilities from being exploited on posterior to Westmere micro-architectures, and the runtime overhead is from 0.0% to 2.7% on Skylake and from 0.0 % to 3.8 % on Westmere on widely-used benchmarks.

Chapter 4

Conclusion

4.1 Contribution Summary

This dissertation has conducted studies to enhance the performance and security of virtual CPUs in the commodity hypervisor. First, our in-depth analysis of the design and implementation of KVM shows that the commodity hypervisor continues to suffer from inefficiency and security concerns associated with CPU virtualization despite tremendous efforts from previous studies. Then, based on the analysis, this dissertation proposes mitigations against the uncovered real-world issues with minimal host modifications to consider applicability to the commodity hypervisor. Thanks to this minimal modification concept, the part of our proposal has been integrated into the mainline Linux/KVM, which is a widely adopted open-source commodity hypervisor.

As for the performance issues, this dissertation shows three issues in KVM: scheduler mismatch, aggressive candidate limiting, and IPI context misuse. These issues enlarge excessive vCPU spinning in KVM and cause severe performance degradation. Our quantitative analysis of the KVM vCPU scheduler reveals that these issues come from semantic gaps between Linux and KVM or guest VMs and KVM. To mitigate these issues, this dissertation proposes three mitigations: debooster, relaxed boost, and IPI-aware boost. For applicability to the commodity hypervisor, these mitigations require only 89 LoC and do not modify guest OSes and the host scheduler core. Despite the modest modifications, these mitigations reduce excessive vCPU spinning and improve the per-

formance of the benchmarks running in the guest VMs.

As for the security issue, this dissertation shows that the support for backward compatibility of KVM broadens the attack surface of the instruction emulator. This dissertation proposes *FWinst* against the security issue. *FWinst* is a context-aware instruction filter that minimizes the instruction emulator logically by utilizing hardware virtualization extensions. The experiments in this dissertation describe that *FWinst* effectively prevents exploiting the real-world vulnerabilities of the instruction emulator while it introduces negligible performance overhead.

4.2 Future Direction

As shown in Section 2.3.2, limiting candidate vCPUs to boost effectively reduces excessive vCPU spinning unless an unexpected underboost exists. Although the KVM vCPU scheduler already has optimizations to limit the candidates, it still needs to consider a new design of the guest OS. For example, the design performs polling at high privilege for high-performance I/O like DPDK [27] with an unikernel approach. In this case, boosting other vCPUs is meaningless because the polling vCPU does not wait for other vCPUs. It is necessary to propose new heuristic-based candidate restrictions for each possible guest OS type by taking advantage of the safety net provided by relaxed boost.

While debooster effectively reduces PLE events in guest VMs, coordinating inter run queue priority is out of scope in this dissertation. Consequently, the yielded vCPU can cause continuous PLE events because the boosted vCPU is not prioritized over other threads if the boosted vCPU and the yielded vCPU are not in the same run queue, but this is necessary to maintain system fairness. As shown in this dissertation, since the host Linux scheduler does not distinguish between vCPUs and other threads, the KVM vCPU scheduler should have more fine-grained vCPU management to consider the possibility of that the boosted vCPUs are not scheduled.

For further vCPU security enhancement, dividing emulation contexts into the finer ones and pruning a legitimate set of instructions for each fine-grained context is also an interesting approach. In particular, if *FWinst* is installed in PaaS (Platform-as-a-Service) clouds, the hypervisor can make more assumptions

about guest OSes, enabling us to prepare fine-tuned contexts for each guest OS. This would enhance the protection against vulnerable emulators.

Additionally, *FWinst* shows room for evolution in hardware virtualization support to prevent cross-modifying code attacks. If hardware support can uniquely determine the instruction that caused the VMExit, *FWinst* can build a legitimate instruction set containing only one instruction. This is the minimum and ideal case of the attack surface, which means attackers must cause VMExit with instructions that can exploit vulnerabilities in the instruction emulator. Since the instructions used in attacks are typically not emulated with the processor's full-fledged support for virtualization turned on, as shown in Section 3.4.1, the minimum attack surface model improves the security of commodity hypervisors.

Bibliography

[1] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. 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, Feb. 2020. USENIX Association.

[2] J. Ahn, C. H. Park, T. Heo, and J. Huh. Accelerating critical os services in virtualized systems with flexible micro-sliced cores. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[3] J. Ahn, C. H. Park, and J. Huh. Micro-sliced virtual processors to hide the effect of discontinuous cpu availability for consolidated systems. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, page 394–405, USA, 2014. IEEE Computer Society.

[4] J. R. Alexander Tereshkin. Bluepilling the xen hypervisor. Black Hat USA, 2008.

[5] AMD. *AMD Secure Encrypted Virtualization (SEV)*. September 2023.

[6] AMD. AMD64 architecture programmer's manual volume 2: System programming. White paper, AMD, 2006.

[7] N. Amit, D. Tsafrir, A. Schuster, A. Ayoub, and E. Shlomo. Virtual CPU Validation. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 311–327, New York, NY, USA, 2015. ACM.

[8] N. Amit, D. Tsafrir, A. Schuster, A. Ayoub, and E. Shlomo. Virtual cpu validation. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 311–327, New York, NY, USA, 2015. Association for Computing Machinery.

[9] N. Amit and M. Wei. The design and implementation of hyperupcalls. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 97–112, Boston, MA, July 2018. USENIX Association.

[10] Apache. ab - Apache HTTP server benchmarking tool. `https://httpd.apache.org/docs/2.4/programs/ab.html`, 2017.

[11] A. Arcangeli. Using Linux as Hypervisor with KVM. `https://indico.cern.ch/event/39755/attachments/797208/1092716/slides.pdf`, 2008.

[12] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, Savannah, GA, Nov. 2016. USENIX Association.

[13] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. Hypersentry: Enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, page 38–49, New York, NY, USA, 2010. Association for Computing Machinery.

[14] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 267–283, Broomfield, CO, Oct. 2014. USENIX Association.

[15] P. Bonzini. KVM: x86 emulator: emulate MOVAPS and MOVAPD SSE instructions. Linux Kernel Mailing List. `https://lkml.org/lkml/2014/3/17/384`, 2014.

[16] S. Butt, H. A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy. Self-service cloud computing. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 253–264, New York, NY, USA, 2012. Association for Computing Machinery.

[17] C. Cesarano, M. Cinque, D. Cotroneo, L. De Simone, and G. Farina. Iris: a record and replay framework to enable hardware-assisted virtualization fuzzing. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 389–401, 2023.

[18] C. che Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, Santa Clara, CA, July 2017. USENIX Association.

[19] J. Chen, D. Li, Z. Mi, Y. Liu, B. Zang, H. Guan, and H. Chen. Security and performance in the delegated user-level virtualization. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 209–226, Boston, MA, July 2023. USENIX Association.

[20] L. Cheng, J. Rao, and F. C. M. Lau. Vscale: Automatic and efficient processor scaling for smp virtual machines. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[21] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable address spaces using rcu balanced trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, page 199–210, New York, NY, USA, 2012. Association for Computing Machinery.

[22] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 189–202, New York, NY, USA, 2011. Association for Computing Machinery.

[23] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 191–206, New York, NY, USA, 2015. Association for Computing Machinery.

[24] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. *SIGARCH Comput. Archit. News*, 43(1):191–206, mar 2015.

[25] L. Deng, P. Liu, J. Xu, P. Chen, and Q. Zeng. Dancing with wolves: Towards practical event-driven vmm monitoring. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, page 83–96, New York, NY, USA, 2017. Association for Computing Machinery.

[26] Digital Ocean. The modern Droplet: how to choose the "right" VM for business and personal use. `https://www.digitalocean.com/blog/how-to-choose-the-right-droplet-vm`, 2021.

[27] DPDK Project. DPDK: the Data Plane Development Kit. `https://www.dpdk.org`, 2022.

[28] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proc. 17th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, page 37–48, 2012.

[29] P. Fonseca, X. Wang, and A. Krishnamurthy. Multinyx: A multi-level abstraction framework for systematic analysis of hypervisors. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[30] T. Friebel and S. Biemueller. How to Deal with Lock Holder Preemption. Xen Summit, 2008.

[31] X. Ge, B. Niu, R. Brotzman, Y. Chen, H. Han, P. Godefroid, and W. Cui. Hyperfuzzer: An efficient hybrid fuzzer for virtual cpus. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 366–378, New York, NY, USA, 2021. Association for Computing Machinery.

[32] Google. crosvm - The Chrome OS Virtual Machine Monitor. `https://chromium.googlesource.com/chromiumos/platform/crosvm/`, 2022.

[33] Google. Compute Engine Documentation. `https://cloud.google.com/compute/docs/faq`, 2024.

[34] R. Gouicem, D. Sprokholt, J. Ruehl, R. C. O. Rocha, T. Spink, S. Chakraborty, and P. Bhatotia. Risotto: A dynamic binary translator for weak memory model architectures. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, page 107–122, New York, NY, USA, 2022. Association for Computing Machinery.

[35] A. Henderson, H. Yin, G. Jin, H. Han, and H. Deng. VDF: Targeted Evolutionary Fuzz Testing of Virtual Devices. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 3–25. Springer, 2017.

[36] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2022.

[37] Intel. Intel Software Guard Extensions (Intel SGX). `https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html`, 2023.

[38] K. Ishiguro. [RFC PATCH 0/2] Mitigating Excessive Pause-Loop Exiting in VM-Agnostic KVM. `https://lore.kernel.org/all/20210421150831.60133-1-kentaishiguro@sslab.ics.keio.ac.jp/`, 2021.

[39] X. Jiang and X. Wang. "out-of-the-box" monitoring of vm-based high-interaction honeypots. In *Proceedings of the 10th International Conference*

on Recent Advances in Intrusion Detection, RAID'07, page 198–218, Berlin, Heidelberg, 2007. Springer-Verlag.

[40] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, page 128–138, New York, NY, USA, 2007. Association for Computing Machinery.

[41] S. Jin, J. Ahn, S. Cha, and J. Huh. Architectural support for secure virtualization under a vulnerable hypervisor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, page 272–283, New York, NY, USA, 2011. Association for Computing Machinery.

[42] R. K T. Virtual cpu scheduling techniques for kernel based virtual machine (kvm). In *2013 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pages 1–6, 2013.

[43] N. A. Kamble. KVM: VMX: Support Unrestricted Guest feature. Linux Kernel Mailing List. `https://lkml.org/lkml/2009/8/16/41`, 2009.

[44] N. A. Kamble. Unrestricted guest support in VMX. Xen.org mailing list. `http://old-list-archives.xenproject.org/xen-devel/2009-05/msg01196.html`, 2009.

[45] S. Kashyap, C. Min, and T. Kim. Scalability in the clouds! a myth or reality? In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, APSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[46] S. Kashyap, C. Min, and T. Kim. Scaling guest OS critical sections with eCS. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 159–172, Boston, MA, July 2018. USENIX Association.

[47] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. Nohype: Virtualized cloud infrastructure without the virtualization. *SIGARCH Comput. Archit. News*, 38(3):350–361, jun 2010.

[48] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. Nohype: Virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, page 350–361, New York, NY, USA, 2010. Association for Computing Machinery.

[49] O. Kilic, S. Doddamani, A. Bhat, H. Bagdi, and K. Gopalan. Overcoming virtualization overheads for large-vcpu virtual machines. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 369–380, 2018.

[50] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng. Demand-based coordinated scheduling for smp vms. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 369–380, New York, NY, USA, 2013. Association for Computing Machinery.

[51] T. Kim, C. H. Park, J. Huh, and J. Ahn. Reconciling time slice conflicts of virtual machines with dual time slice for clouds. *IEEE Transactions on Parallel and Distributed Systems*, 31(10):2453–2465, 2020.

[52] S. King and P. Chen. Subvirt: implementing malware with virtual machines. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 14 pp.–327, 2006.

[53] A. Kopytov. sysbench. `https://github.com/akopytov/sysbench`, 2017.

[54] K. Kortchinsky. Cloudburst: Hacking 3d (and breaking out of vmware). Black Hat USA, 2009.

[55] KVM. KVM. `http://www.linux-kvm.org/page/Main_Page`, 2016.

[56] M. Li, W. Zang, K. Bai, M. Yu, and P. Liu. Mycloud: Supporting user-configured privacy protection in cloud computing. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, page 59–68, New York, NY, USA, 2013. Association for Computing Machinery.

[57] S.-W. Li, J. S. Koh, and J. Nieh. Protecting cloud virtual machines from hypervisor and host operating system exploits. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1357–1374, Santa Clara, CA, Aug. 2019. USENIX Association.

[58] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui. Formally verified memory protection for a commodity multiprocessor hypervisor. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3953–3970. USENIX Association, Aug. 2021.

[59] W. Li. KVM: X86: Add Paravirt TLB Shootdown. `https://lwn.net/Articles/740363/`, 2017.

[60] W. Li. [PATCH] KVM: Boost vCPU candidiate in user mode which is delivering interrupt. `https://lore.kernel.org/kvm/CANRm+Cy-78UnrkX8nh5WdHut2WW5NU=UL84FRJnUNjsAPK+Uww@mail.gmail.com/T/`, 2021.

[61] Q. Liu, F. Toffalini, Y. Zhou, and M. Payer. Videzzo: Dependency-aware virtual device fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 3228–3245, 2023.

[62] Z. Liu, J. Lee, J. Zeng, Y. Wen, Z. Lin, and W. Shi. Cpu transparent protection of os kernel and hypervisor integrity with programmable dram. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 392–403, New York, NY, USA, 2013. Association for Computing Machinery.

[63] LLVMpipe - The Mesa 3D Graphics Library. `https://www.mesa3d.org/llvmpipe.html`, 2023.

[64] W. Long. qspinlock: a 4-byte queue spinlock with pv support. `https://lwn.net/Articles/597672/`, may 2014.

[65] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. Path-exploration lifting: Hi-fi tests for lo-fi emulators. In *Proceedings of*

the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII, page 337–348, New York, NY, USA, 2012. Association for Computing Machinery.

[66] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, feb 1991.

[67] Z. Mi, D. Li, H. Chen, B. Zang, and H. Guan. (mostly) exitless VM protection from untrusted hypervisor through disaggregated nested virtualization. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1695–1712. USENIX Association, Aug. 2020.

[68] D. G. Murray, G. Milos, and S. Hand. Improving xen security through disaggregation. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, page 151–160, New York, NY, USA, 2008. Association for Computing Machinery.

[69] T. D. Ngoc, B. Teabe, A. Tchana, G. Muller, and D. Hagimont. Mitigating vulnerability windows with hypervisor transplant. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 162–177, New York, NY, USA, 2021. Association for Computing Machinery.

[70] A. Nguyen, H. Raj, S. Rayanchu, S. Saroiu, and A. Wolman. Delusional boot: Securing hypervisors without massive re-engineering. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 141–154, New York, NY, USA, 2012. Association for Computing Machinery.

[71] J. Ogasawara and K. Kono. Nioh: Hardening the hypervisor by filtering illegal i/o requests to virtual devices. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC '17, page 542–552, New York, NY, USA, 2017. Association for Computing Machinery.

[72] OpenMP. OpenMP. `https://www.openmp.org/`, 2012-2023.

[73] J. Ouyang and J. R. Lange. Preemptable ticket spinlocks: Improving consolidated performance in the cloud. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, page 191–200, New York, NY, USA, 2013. Association for Computing Machinery.

[74] J. Ouyang, J. R. Lange, and H. Zheng. Shoot4u: Using vmm assists to optimize tlb operations on preempted vcpus. In *Proceedings of The12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '16, page 17–23, New York, NY, USA, 2016. Association for Computing Machinery.

[75] G. Pan, X. Lin, X. Zhang, Y. Jia, S. Ji, C. Wu, X. Ying, J. Wang, and Y. Wu. V-shuttle: Scalable and semantics-aware hypervisor virtual device fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2197–2213, New York, NY, USA, 2021. Association for Computing Machinery.

[76] Phoronix Media. Phoronix Test Suite - Linux Testing & Benchmarking Platform, Automated Testing, Open-Source Benchmarking. `https://www.phoronix-test-suite.com/`, 2023.

[77] M. Righini. Enabling Intel virtualization technology features and benefits. White paper, Intel, 2010.

[78] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, RAID '08, page 1–20, Berlin, Heidelberg, 2008. Springer-Verlag.

[79] M. Russinovich, N. Govindaraju, M. Raghuraman, D. Hepkin, J. Schwartz, and A. Kishan. Virtual machine preserving host updates for zero day patching in public cloud. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 114–129, New York, NY, USA, 2021. Association for Computing Machinery.

[80] S. Schildermans, J. Shan, K. Aerts, J. Jackrel, and X. Ding. Virtualization Overhead of Multithreading in X86 State-of-the-Art & Remaining Challenges. *IEEE Transactions on Parallel and Distributed Systems*, 32(10):2557–2570, 2021.

[81] S. Schumilo, C. Aschermann, A. Abbasi, S. Wör-ner, and T. Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2597–2614. USENIX Association, Aug. 2021.

[82] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. In *NDSS Symposium 2020*, 2020.

[83] M. Senuki, K. Ishiguro, and K. Kono. Nioh-pt: Virtual i/o filtering for agile protection against vulnerability windows. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, SAC '23, page 1293–1300, New York, NY, USA, 2023. Association for Computing Machinery.

[84] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 335–350, New York, NY, USA, 2007. Association for Computing Machinery.

[85] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. *SIGOPS Oper. Syst. Rev.*, 41(6):335–350, oct 2007.

[86] J. Shan, X. Ding, and N. Gehani. Apples: Efficiently handling spin-lock synchronization on virtualized platforms. *IEEE Transactions on Parallel and Distributed Systems*, 28(7):1811–1824, 2017.

[87] L. Shi, Y. Wu, Y. Xia, N. Dautenhahn, H. Chen, B. Zang, and J. Li. Deconstructing xen. In *NDSS Symposium 2017*, 2017.

[88] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo,

and K. Kato. Bitvisor: A thin hypervisor for enforcing i/o device security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, page 121–130, New York, NY, USA, 2009. Association for Computing Machinery.

[89] X. Song, J. Shi, H. Chen, and B. Zang. Schedule processes, not vcpus. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, APSys '13, New York, NY, USA, 2013. Association for Computing Machinery.

[90] U. Steinberg and B. Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 209–222, New York, NY, USA, 2010. Association for Computing Machinery.

[91] O. Sukwong and H. S. Kim. Is co-scheduling too expensive for smp vms? In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, page 257–272, New York, NY, USA, 2011. Association for Computing Machinery.

[92] J. Szefer, E. Keller, R. B. Lee, and J. Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, page 401–412, New York, NY, USA, 2011. Association for Computing Machinery.

[93] J. Szefer and R. B. Lee. Architectural support for hypervisor-secure virtualization. *SIGPLAN Not.*, 47(4):437–450, mar 2012.

[94] J. Szefer and R. B. Lee. Architectural support for hypervisor-secure virtualization. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, page 437–450, New York, NY, USA, 2012. Association for Computing Machinery.

[95] B. Teabe, V. Nitu, A. Tchana, and D. Hagimont. The lock holder and the lock waiter pre-emption problems: Nip them in the bud using informed spinlocks (i-spinlock). In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 286–297, New York, NY, USA, 2017. Association for Computing Machinery.

[96] B. Teabe, A. Tchana, and D. Hagimont. Application-specific quantum for multi-core platform scheduler. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[97] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3*, VM'04, page 4, USA, 2004. USENIX Association.

[98] Unix Bench. `https://github.com/kdlucas/byte-unixbench`, 2017.

[99] R. van Riel. directed yield for Pause Loop Exiting. `https://lwn.net/Articles/421575/`, 2011.

[100] VMware. *VMware vSphere 4: The CPU Scheduler in VMware ESX 4.1*, 2010.

[101] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, page 380–395, USA, 2010. IEEE Computer Society.

[102] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, page 545–554, New York, NY, USA, 2009. Association for Computing Machinery.

[103] Z. Wang, C. Wu, M. Grace, and X. Jiang. Isolating commodity hosted hypervisors with hyperlock. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 127–140, New York, NY, USA, 2012. Association for Computing Machinery.

[104] P. M. Wells, K. Chakraborty, and G. S. Sohi. Hardware support for spin management in overcommitted virtual machines. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, page 124–133, New York, NY, USA, 2006. Association for Computing Machinery.

[105] C. Weng, Q. Liu, L. Yu, and M. Li. Dynamic adaptive scheduling for virtual machines. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, page 239–250, New York, NY, USA, 2011. Association for Computing Machinery.

[106] R. Wojtczuk. Subverting the xen hypervisor. Black Hat USA, 2008.

[107] C. Wu, Z. Wang, and X. Jiang. Taming hosted hypervisors with (mostly) deprivileged execution. In *NDSS Symposium 2013*, 2013.

[108] S. Wu, Z. Xie, H. Chen, S. Di, X. Zhao, and H. Jin. Dynamic acceleration of parallel applications in cloud platforms by adaptive time-slice control. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 343–352, 2016.

[109] Y. Wu, Y. Liu, R. Liu, H. Chen, B. Zang, and H. Guan. Comprehensive vm protection against untrusted hypervisor through retrofitted amd memory encryption. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 441–453, 2018.

[110] Xenproject.org Security Team. Xen Security Advisory. `https://xenbits.xen.org/xsa/`, 2017.

[111] Y. Xia, Y. Liu, and H. Chen. Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 246–257, 2013.

[112] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 203–216, New York, NY, USA, 2011. Association for Computing Machinery.

[113] F. Zhang, J. Wang, K. Sun, and A. Stavrou. Hypercheck: A hardware-assistedintegrity monitor. *IEEE Transactions on Dependable and Secure Computing*, 11(4):332–344, 2014.

[114] L. Zhang, Y. Chen, Y. Dong, and C. Liu. Lock-visor: An efficient transitory co-scheduling for mp guest. In *2012 41st International Conference on Parallel Processing*, pages 88–97, 2012.

[115] X. Zhang, X. Zheng, Z. Wang, H. Yang, Y. Shen, and X. Long. High-Density Multi-Tenant Bare-Metal Cloud. In *Proc. 25th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 483–495, 2020.

[116] Y. Zhao, J. Rao, and Q. Yi. Characterizing and optimizing the performance of multithreaded programs under interference. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, page 287–297, New York, NY, USA, 2016. Association for Computing Machinery.

[117] Y. Zhao, K. Suo, L. Cheng, and J. Rao. Scheduler activations for interference-resilient smp virtual machine scheduling. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, page 222–234, New York, NY, USA, 2017. Association for Computing Machinery.