Hardening Hypervisors against Vulnerabilities in Instruction Emulators

Kenta Ishiguro
Keio University
kentaishiguro@sslab.ics.keio.ac.jp

Kenji Kono
Keio University
kono@sslab.ics.keio.ac.jp

ABSTRACT

Vulnerabilities in hypervisors are crucial in multi-tenant clouds and attractive for attackers because a vulnerability in the hypervisor can undermine all the virtual machine (VM) security. This paper focuses on vulnerabilities in instruction emulators inside hypervisors. Vulnerabilities in instruction emulators are not rare; CVE-2017-2583, CVE-2016-9756, CVE-2015-0239, CVE-2014-3647, to name a few. For backward compatibility with legacy x86 CPUs, conventional hypervisors emulate arbitrary instructions at any time if requested. This design leads to a large attack surface, making it hard to get rid of vulnerabilities in the emulator.

This paper proposes *FWinst* that narrows the attack surface against vulnerabilities in the emulator. The key insight behind *FWinst* is that the emulator should emulate only a small subset of instructions, depending on the underlying CPU micro-architecture and the hypervisor configuration. *FWinst* recognizes emulation contexts in which the instruction emulator is invoked, and identifies a legitimate subset of instructions that are allowed to be emulated in the current context. By filtering out illegitimate instructions, *FWinst* narrows the attack surface. In particular, *FWinst* is effective on recent x86 micro-architectures because the legitimate subset becomes very small. Our experimental results demonstrate *FWinst* prevents existing vulnerabilities in the emulator from being exploited on Westmere micro-architecture, and the runtime overhead is negligible.

CCS CONCEPTS

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

KEYWORDS

Virtualization, Hypervisor, Instruction Emulator, Security

ACM Reference Format:
Kenta Ishiguro and Kenji Kono. 2018. Hardening Hypervisors against Vulnerabilities in Instruction Emulators. In *EuroSec'18: 11th European Workshop on Systems Security , April 23–26, 2018, Porto, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3193111.3193118

1 INTRODUCTION

Vulnerabilities in hypervisors are crucial in multi-tenant clouds because they can undermine all the virtual machine (VM) security. If a vulnerability results in *VM Escape*, a malicious VM breaks out of itself, gets the full control over the hypervisor, and attacks other co-located VMs using the privilege of the hypervisor. Since the hypervisor is the most privileged, the malicious VM can do whatever it wants. Unfortunately, there are many reported vulnerabilities in the hypervisor. As of November 2017, 110 CVEs are reported for KVM [8] and 240 vulnerabilities are in Xen Security Advisories (XSA) [15].

This paper focuses on vulnerabilities in instruction emulation in the hypervisor. Ideally, the hypervisor would only need to emulate a small subset of the instruction set. However, on x86 architecture, the hypervisor may be required to emulate most instructions [4, 5]. Instructions other than sensitive ones are emulated in 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 [11].
- **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.

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. [4] demonstrate any instructions can be forced to be emulated. This new attack vector allows an attacker to exploit a vulnerability in any instructions.

This paper presents *FWinst*, which raises the bar for attacks on instruction emulation by narrowing the attack surface against it.

EuroSec'18, April 23–26, 2018, Porto, Portugal

ACM ISBN 978-1-4503-5652-7/18/04.
https://doi.org/10.1145/3193111.3193118

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 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 instructions not legitimate in the current context. *FWinst* identifies five contexts: 1) PIO context, 2) MMIO context, 3) shadow page table context, 4) real-mode context, and 5) migration 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 a hypervisor is invoked 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), which runs on Intel Westmere micro-architecture 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). 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.

The paper is organized as follows. Section 2 describes the threat model and analyzes the vulnerabilities in instruction emulation. Section 3 shows the design and implementation of *FWinst*, and Section 4 reports the experimental results. Section 5 relates our work with others and Section 6 concludes the paper.

2 THREAT MODEL AND VULNERABILITY ANALYSIS

2.1 Treat Model

Before describing the threat model, this section explains how an instruction emulator is invoked inside the hypervisor, targeting on Intel CPU with virtualization support (VT-x). Figure 1 a) illustrates the internal architecture of typical hypervisors. Whenever some support is necessary from the hypervisor, CPU causes a "VMExit" and the control is transferred from a guest VM to the hypervisor. To indicate the reason the VMExit has occurred, a small integer called "VMExit reason" is set by CPU in a special memory area (VMCS; VM control structure). On the VMExit, a handler dedicated to each VMExit reason is invoked. For example, when a guest VM executes `cpuid` instruction, the handler for `cpuid` is automatically invoked without decoding `cpuid` instruction.

Decoding VMExiting instructions is required in some handlers to get the exact operations of those instructions. For example, if an MMIO region is accessed, a VMExit is caused with EPT violation (illegal memory access) as the VMExit reason. In this case, the instruction emulator is invoked to decode the VMExiting instruction to get what operation is done on the MMIO region. As described in Section 1, hypervisors on Intel x86 emulate instructions in the following contexts: 1) Port I/O, 2) MMIO, 3) Shadow Page Table, 4) Real Mode, and 5) Migration.

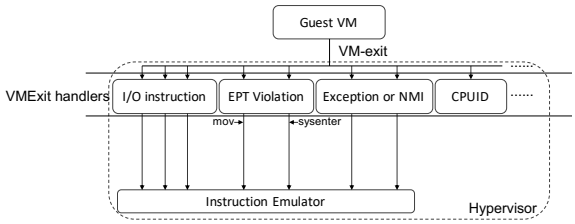
At first glance, an attacker appears unable to exploit a vulnerable instruction if it does not cause any VMExit because the emulator is not invoked. Suppose that an attacker is trying to exploit a vulnerability in the emulation of `sysenter` instruction (CVE-2015-0239). When `sysenter` is executed on Intel x86, it does not cause any VMExits and thus the emulator is not invoked. Interestingly, Amit et al. [4] have proposed a new attack vector to force the emulator to decode whichever instruction the attacker wants to exploit. This attack vector is a timing attack and exploits a short time interval between the VMExit and the emulator invocation. In Figure 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 proposed 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.

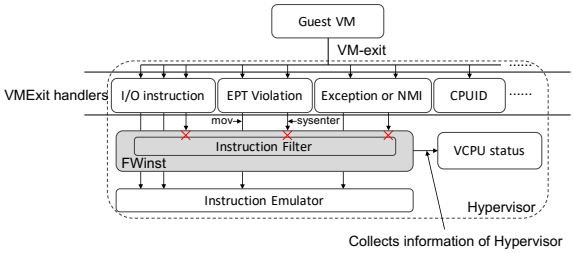
2.2 Vulnerability Analysis

As mentioned in Section 1, 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 lead 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



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 1: Instruction Emulator in Ordinary Hypervisors and in Hypervisors with *FWinst*.

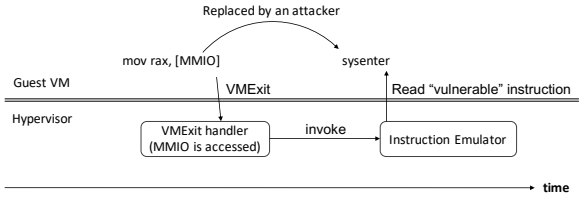


Figure 2: Timing Attack on Instruction Emulation.

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 5 emulation contexts, the hypevisor on recent micro-architectures has to support only 3 contexts: 1) Port I/O, 2) MMIO, and 3) Migration. Emulation in Real-Mode and Shadow Page Table is not necessary in recent micro-architectures because real-mode in guest-mode is allowed and EPT (extended page table) is supported for second level address translation.

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.

Table 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`

3 DESIGN AND IMPLEMENTATION

The vulnerability analysis in Section 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.

3.1 Overall Architecture

Figure 1 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 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 other contexts, some engineering efforts are needed to determine the legitimate set. Section 3.3 describes the approach *FWinst* has taken to determine the legitimate set.

3.2 Identifying Emulation Contexts

Table 1 shows the summary of the emulation contexts identified in *FWinst*. *FWinst* identifies five contexts: 1) Port I/O, 2) MMIO, 3) shadow page table, 4) real mode, and 5) migration.

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.

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. 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, the hypervisor does not invoke *FWinst* because it requires only the faulting address for processing the VMExits.

Shadow page table context. If the EPT feature is not available, the shadow page table context is identified with the cooperation of the hypervisor. Since the hypervisor knows the memory locations of guest page tables, *FWinst* concludes that the current context is the shadow page table context if the faulting address fits in the guest page tables.

Real mode context. If the unrestricted guest mode is not available, the real-mode code is executed either in virtual 8086 mode or on the emulator. If this is the case, the hypervisor maintains a global state that tells the emulation for real-mode is required or not. *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 any vendor-specific instructions are emulated without further inspection. Since the number of legitimate instructions in the migration context is very limited, *FWinst* carefully inspects the instruction modes and rejects the emulation later in the verification phase.

3.3 Legitimate Instructions

For each emulation context, a set of legitimate instructions are defined. Table 1 shows the summary of the legitimate set of instructions for each context. For PIO context, it is straightforward to define the set; the family of `in` and `out` instructions because I/O ports are accessed only through them.

For MMIO context and shadow page table context, the legitimate set of instructions is memory-accessing instructions; i.e, instructions having memory-access operands. If the operating systems hosted on the hypervisor are known in 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.

The coding conventions can be utilized as follows. For MMIO context, since an MMIO region is accessed from device drivers, the legitimate set can be derived from in-kernel functions or macros exported for device drivers. Memory-accessing instructions that do not appear in the compiled macros or functions can be removed from the legitimate set. For example, Linux provides `readl` and `writel` macros for MMIO accesses. Windows provides `READ_REGISTER_UCHAR` and `WRITE_REGISTER_UCHAR` functions for MMIO accesses. For the shadow page table context, all the functions that update page tables must be investigated to restrict the legitimate set.

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 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 the 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`. The legitimate set changes depending on the vendors and micro-architectures of physical CPUs. Since it is nonsense to emulate natively supported instructions, the legitimate set includes the instructions that are not supported natively on the physical CPUs.

3.4 Implementation

A prototype of *FWinst* has been implemented on Linux KVM (Linux Kernel 4.8) for Intel x86-64 architecture. 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-architecture or later.

The current prototype identifies PIO, MMIO, and migration contexts. Shadow page table or real mode contexts are not recognized because the EPT feature and the unrestricted guest mode are enabled. For MMIO context, the legitimate set is restricted, assuming that the guest operating systems are Linux or Windows. Since BIOS makes an access to an MMIO region, we took execution logs of BIOS to elaborate the set. As a result, the legitimate set for MMIO includes the family of `MOV`, `MOVSx`, `STOSx`, and `OR`. For Migration context, our current implementation includes the instructions specific to AMD (`vmmcall`) and supported on later Intel micro-architectures (`movbe`). Currently, `rsm` is also included in the set.

In contexts other than migration, *FWinst* needs only the opcode of an emulated instruction. To avoid duplicated implementation of instruction decoders, *FWinst* lets the instruction emulator decode each instruction. This design allows us to reuse the instruction decoder, and releases us from maintaining two decoders (the one in

Table 2: Experimental Environment

Host OS	Linux Kernel 4.8.1
Host QEMU	Version 2.9.50
Host CPU	Intel Westmere Xeon X5650 2.67 GHz
Host memory	4 GB
Guest OS	Ubuntu 16.10 x86_64
# of VCPUs	2
Guest memory	1 GB

Table 3: Summary of vulnerabilties

CVE #	vul. inst.	Intel	AMD
2014-3647	`far jump` or `far ret`	√	√
2014-8480	`clflush`, `hint-nop`, `prefetch`	√	√
2014-8481	`movbe`	d	d
2015-0239	`sysenter`	√	d
2016-8630	illegal instruction	√	√
2016-9756	`far jump` or `far ret`	√	√
2017-2583	`mov SS`	√	√
2017-2584	`fxrstor`, `fxsave`, `sgdt`, `sidt`	√	√

d: depends on migration contexts

the instruction emulator and the other in *FWinst*). After the instruction emulator finishes decoding the opcode, the opcode is notified to *FWinst* and *FWinst* filters it out if the instruction is not in the legitimate set. Note that *FWinst* does *not* rely on the operand decoder, which is more complicated and more vulnerable than the opcode decoder; even if there is a vulnerability in the operand decoder, *FWinst* works properly.

4 EXPERIMENTS

To demonstrate the effectiveness of *FWinst*, we have implemented an early prototype of *FWinst* on Intel x86 Westmere micro-architecture. 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 2 shows the experimental environment.

4.1 Security Analysis

To demonstrate the effectiveness of *FWinst*, we have investigated 20 vulnerability reports from 2009 to 2017 that are related to emulation of instructions, and confirmed *FWinst* would prevent the emulation by the instruction emulator of all the vulnerable instructions on Haswell (later than Westmere) micro-architecture because those instructions are not included in any legitimate set of instructions of any emulation contexts. This result does not imply *FWinst* can defend against all vulnerabilities in the emulator. Since *FWinst* simply narrows the attack surface, it cannot defend against vulnerabilities in the instructions in the legitimate set.

For detailed discussion, we have chosen eight vulnerabilities which are emulated by the instruction emulator listed in Table 3. For these vulnerabilities we have collected or implemented PoC (proof-of-concept) code and tested it on *FWinst*. As you can see from Table 3, *FWinst* can defend against 7 vulnerabilities out of 8 on Intel Westmere micro-architecture (indicated by √ in column 'Intel'), and 6 out of 8 on AMD (indicated by √ in column 'AMD').

CVE-2014-8481, which is marked as 'depends' in both Intel and AMD, is about the emulation of `movbe` instruction, which has been introduced in Haswell (later than Westmere). 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.

Since `movbe` is an Intel-specific instruction, *FWinst* running on AMD rejects the emulation of `movbe` if it recognizes the guest is running binary for AMD. But if the guest is migrated from another machine and runs binary for Intel, *FWinst* on AMD emulates `movbe` because `movbe` is included in the legitimate set of instructions for Migration context. Vulnerable `movbe` can be exploited in this case.

Column 'AMD' is marked as 'depends' in Table 3 in CVE-2015-0239. Since `sysenter` is an Intel-specific instruction, *FWinst* running on Intel rejects the emulation of this instruction because there is no need to natively supported instructions. But the situation is subtle if this instruction is executed in guest on AMD. If *FWinst* recognizes the guest is running binary for AMD, *FWinst* rejects the emulation because it is strange that AMD tries to execute unsupported instruction. But if the guest is migrated from another machine and runs binary for Intel, *FWinst* emulates `sysenter` on AMD and the vulnerability can not be avoided.

`Mov SS` in CVE-2017-2583 is an instruction that loads or stores the stack segment register. Although this instruction looks like a memory-accessing instruction, it is excluded from the legitimate set for MMIO context because the segment registers are not loaded from or stored to MMIO regions. *FWinst* running on Intel or AMD CPUs reject the emulation and prevents this vulnerability from being exploited.

CVE-2016-8630 is about the emulation of illegal instructions. If a guest executes an illegal instruction, the VMExit handler for illegal instructions is invoked and it is determined in the handler if the emulation is necessary or not. If an adversary uses the timing attack introduced in Figure 2, the VMExit handler for illegal instructions is bypassed. KVM emulator tries to emulate the illegal instruction, which results in the exploitation of the vulnerability. Before invoking the emulator, *FWinst* checks the legitimate set of instructions for the illegal instruction, and rejects the emulation because it is not in the current legitimate set.

4.2 Runtime Overhead

To estimate runtime overhead introduced by *FWinst*, we measure the runtime of several standard benchmarks: UnixBench [3], Apache Bench [1] and sysbench [2]. Figure 3 shows the relative performance. Overall, *FWinst* introduces the overhead less than about 2.5%.

5 RELATED WORK

Testing The Hypervisor. Virtual CPU Validation [4] takes advantage of Intel's testing facilities to look for security vulnerabilities in KVM. Over half of the 117 bugs they discovered are instruction emulator bugs, five of which are security vulnerabilities. To

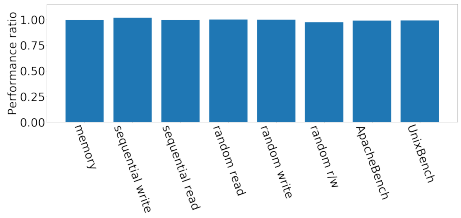


Figure 3: Normalized performance of UnixBench, Apache Bench and sysbench with the original KVM as the baseline

exploit vulnerabilities in instruction emulators, a new attack vector is shown to force the emulator to emulate arbitrary instructions at any time.

Hardening Hypervisors. Conceptually, *FWinst* is similar to network firewalls. A network firewall narrows the attack surface against vulnerable servers and clients inside the firewall. While many techniques are developed and deployed to harden servers and clients in general, network firewalls are still useful to reduce the risk of exposing vulnerable servers and clients. *FWinst* reduces the risk of exposing vulnerable emulation of instructions, and allows us to get rid of the emulation of legacy and intricate instructions. Nioh [10] is conceptually similar to *FWinst*. It is a firewall for virtual devices that filters out suspicious requests to virtual devices.

Aside from the approach like *FWinst*, there are many research efforts to harden the hypervisors against general attacks. These approaches can be used together with *FWinst*. Since none of these approaches can eliminate all the security threats, *FWinst* reduces the risk of exposing security holes lurking in the emulator that spill out of the state-of-the-art protection.

Monitoring hypervisors at runtime is a promising approach to improve the security of hypervisors. Dancing with Wolves [6] monitors untrusted hypervisors from a secure event-driven monitor. HyperSafe [16] and HyperVerify [7] provide runtime protection for hypervisors.

Another approach to hardening hypervisors is to reduce TCB (trusted computing base) in the hypervisor. Min-V [9] disables all unnecessary virtual devices when running VMs in the cloud. NoHype [14] eliminates the virtualization layer at runtime, and each VM directly runs on statically assigned resources. NOVA [13] takes a microkernel approach to achieving a smaller TCB. Deconstructing Xen [12] divides Xen's privileged code into per-VM slices, and confines the attacks inside the slices. HyperLock [17] prepares a shadow hypervisor for each VM and provides runtime isolation for the privileged host. DeHype [18] demotes KVM to user mode and runs it as a per-VM deprivileged hypervisor.

6 CONCLUSION

The contribution of this paper is that the attack surface against vulnerabilities in the emulator can be narrowed, if the underlying micro-architecture and the hypervisor configuration are taken into account. *FWinst* identifies a legitimate 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 Westmere micro-architecture, and the runtime overhead is less than 2.5% on widely-used benchmarks.

For future directions, it would be interesting to divide emulation contexts into the finer ones and prune a legitimate set of instructions for each fine-grained context. In particular, if *FWinst* is installed in PaaS (Platform-as-a-Service) clouds, the hypervisor can make more assumptions on guest operating systems, which would enable us to prepare fine-tuned contexts for each guest operating system. This would enhance the protection against vulnerable emulators.

ACKNOWLEDGEMENT

This work is partially supported by Japan Science and Technol- ogy Agency (JST CREST JPMJCR1683).

REFERENCES

[1] 2017. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html. (2017).
[2] 2017. sysbench. https://github.com/akopytov/sysbench. (2017).
[3] 2017. Unix Bench. https://github.com/kdlucas/byte-unixbench. (2017).
[4] Nadav Amit, Dan Tsafrir, Assaf Schuster, Ahmad Ayoub, and Eran Shlomo. 2015. Virtual CPU Validation. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 311–327. https://doi.org/10.1145/2815400.2815420
[5] Andrea Arcangeli. 2008. Using Linux as Hypervisor with KVM. https://indico.cern.ch/event/39755/attachments/797208/1092716/slides.pdf. (2008).
[6] Liang Deng, Peng Liu, Jun Xu, Ping Chen, and Qingkai Zeng. 2017. 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)*. ACM, New York, NY, USA, 83–96. https://doi.org/10.1145/3050748.3050750
[7] Baozeng Ding, Yeping He, Yanjun Wu, and Yuqi Lin. 2013. HyperVerify: A VM-assisted Architecture for Monitoring Hypervisor Non-control Data. In *Proceedings of the 2013 IEEE Seventh International Conference on Software Security and Reliability Companion (SERE-C '13)*. IEEE Computer Society, Washington, DC, USA, 26–34. https://doi.org/10.1109/SERE-C.2013.20
[8] KVM. 2016. KVM. http://www.linux-kvm.org/page/Main_Page. (2016).
[9] Anh Nguyen, Himanshu Raj, Shravan Rayanchu, Stefan Saroiu, and Alec Wolman. 2012. Delusional Boot: Securing Hypervisors Without Massive Re-engineering. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, New York, NY, USA, 141–154. https://doi.org/10.1145/2168836.2168851
[10] Junya Ogasawara and Kenji Kono. 2017. Nioh: Hardening The Hypervisor by Filtering Illegal I/O Requests to Virtual Devices. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC 2017)*. ACM, New York, NY, USA, 542–552. https://doi.org/10.1145/3134600.3134648
[11] Paolo Bonzini. 2014. KVM: x86 emulator: emulate MOVAPS and MOVAPD SSE instructions. Linux Kernel Mailing List. https://lkml.org/lkml/2014/3/17/384. (2014).
[12] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, Haibing Guan, and Jinming Li. 2017. Deconstructing Xen. In *The Network and Distributed System Security Symposium 2017 (NDSS '17)*.
[13] Udo Steinberg and Bernhard Kauer. 2010. NOVA: A Microhypervisor-based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. ACM, New York, NY, USA, 209–222. https://doi.org/10.1145/1755913.1755935
[14] Jakub Szefer, Eric Keller, Ruby B. Lee, and Jennifer Rexford. 2011. Eliminating the Hypervisor Attack Surface for a More Secure Cloud. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. ACM, New York, NY, USA, 401–412. https://doi.org/10.1145/2046707.2046754
[15] Xenproject.org Security Team. 2017. Xen Security Advisory. https://xenbits.xen.org/xsa/. (2017).
[16] Zhi Wang and Xuxian Jiang. 2010. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *IEEE Symposium on Security and Privacy*. IEEE, 380–395. https://doi.org/10.1109/SP.2010.30
[17] Zhi Wang, Chiachih Wu, Michael Grace, and Xuxian Jiang. 2012. Isolating Commodity Hosted Hypervisors with HyperLock. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, New York, NY, USA, 127–140. https://doi.org/10.1145/2168836.2168850
[18] Chiachih Wu, Zhi Wang, and Xuxian Jiang. 2013. Taming Hosted Hypervisors with (Mostly) Deprivileged Execution.. In *NDSS (NDSS '13)*. The Internet Society.