IEICE TRANS. INF. & SYST., VOL.E103–D, NO.7 JULY 2020

PAPER

Instruction Filters for Mitigating Attacks on Instruction Emulation in Hypervisors*

Kenta ISHIGURO[†a)], *Nonmember and* **Kenji KONO**[†b)], *Member*

SUMMARY 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 and Skylake micro-architectures, and the runtime overhead is negligible.

key words: *virtualization, hypervisor, instruction emulator, security*

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 2018, 111 CVEs are reported for KVM [1] and 275 vulnerabilities are in Xen Security Advisories (XSA) [2].

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 [3], [4]. Instructions other than sensitive ones [5], [6] that must not be executed directly in the guest 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 [7].
- **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. [3] 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* (derived from "Instruction Firewall"), which raises the bar for attacks on instruction

Manuscript received July 1, 2019.
Manuscript revised November 30, 2019.
Manuscript publicized April 6, 2020.

[†]The authors are with the Department of Information and Computer Science, Keio University, Yokohama-shi, 223–8522 Japan.

[*]An earlier version of this paper has appeared in the proceedings of the 11th ACM European Workshop on Systems Security (EuroSec '18).

a) E-mail: kentaishiguro@sslab.ics.keio.ac.jp
b) E-mail: kono@sslab.ics.keio.ac.jp

DOI: 10.1587/transinf.2019EDP7186

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 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 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 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 4.8.1), 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.

Our contribution can be summarized as follows. We show the design and implementation of *FWinst*, a novel protection mechanism against vulnerabilities in the instruction emulator. *FWinst* narrows the attack surface by restricting the instruction emulation to the legitimate sets of the effective emulation context. We have identified six emulation contexts and defined a legitimate set of instructions for each context. The evolution of the hardware support for virtualization reduces the number of instructions that should be emulated by the instruction emulator and contributes to narrowing the attack surface. We demonstrate the effectiveness of *FWinst* against 17 reported vulnerabilities in the instruction emulator. This paper extends our workshop paper [8] with more details and experiments.

The rest of this paper is organized as follows. Section 3 describes the threat model and analyzes the vulnerabilities in instruction emulation. Section 4 shows the design and implementation of *FWinst*, and Sect. 5 reports the experimental results. Section 6 relates our work with others and Sect. 7 concludes the paper.

2. Background

2.1 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 virtualization support (VT-x).

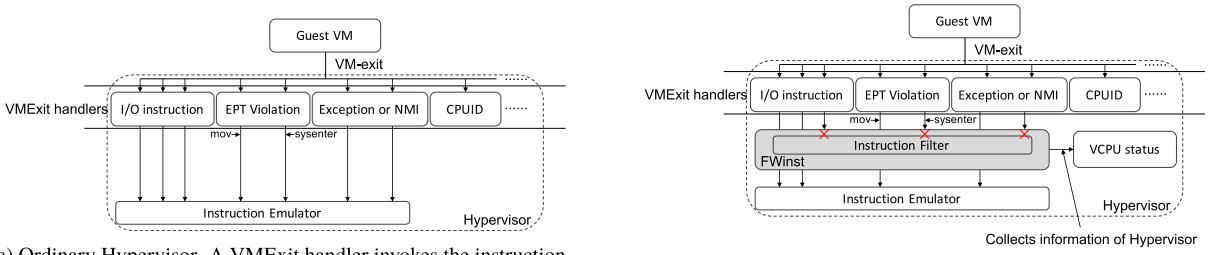
In Intel VT-x, two execution modes, the root and non-root modes are added. Hypervisor code runs in the root mode, whereas a guest VM code runs in the non-root mode. Both the root and non-root modes have traditional execution modes (i.e. real mode and protected mode) and privilege levels (i.e. ring protections). 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.

2.2 Instruction Emulation in Hypervisors

Some instructions executed in a guest VM must be emulated in the hypervisor although most instructions execute natively on host CPUs. For example, if an MMIO region



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.

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

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 microarchitecture, 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 microarchitecture, 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 para-virtualization. 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 [9]. 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 Sect. 2.3.

2.3 Evolution of Intel VT-x

Intel VT-x has evolved since the first introduction on Pen-

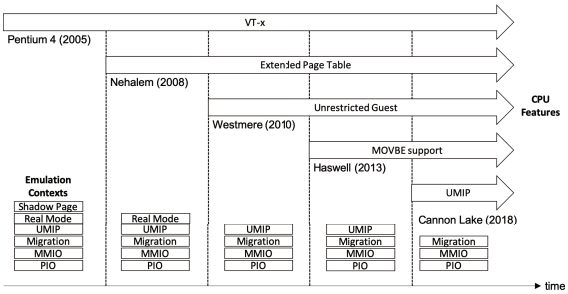


Fig. 2 Evolution of Intel VT-x and corresponding emulation contexts over time.

tium 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 2 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 [10], [11]

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

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 new attack vector that enables attackers to force the emulation of arbitrary instructions to exploit its emulation has been proposed [3]. As a result, in spite of the new features of VT-x, the attack surface in the instruction emulator is still large.

3. Threat Model and Vulnerability Analysis

3.1 Threat Model

Before describing the threat model, this section explains the detail of a new attack vector. This attack vector is necessary to exploit a wide range of vulnerable instructions. To exploit the instruction emulator, an attacker has to force the

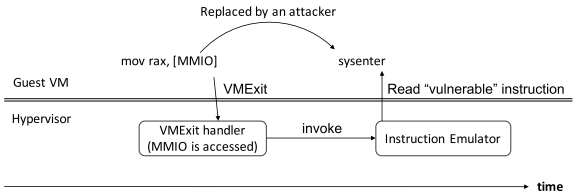


Fig. 3 Timing Attack on Instruction Emulation.

hypervisor to perform emulation of a vulnerable instruction. However, as described in Sect. 2.2, 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. [3] have proposed the 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 Fig. 3, 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.

3.2 Vulnerability Analysis

As described in Sect. 2.3, 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.

4. Design and Implementation

The vulnerability analysis in Sect. 3 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.

4.1 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.

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`
UMIP	Access to GDTR or IDTR or Access to LDTR or TR	`sgdt`, `sidt`, `sldt` `smsw`, `str`

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 4.3 describes the approach *FWinst* has taken to determine the legitimate set.

4.2 Identifying Emulation Contexts

Table 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 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 [10]. 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.

```
#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; \
}
```

Listing 1: Example of MMIO accessor in Linux Kernel 4.8.1 arch/x86/include/asm/io.h line 46

4.3 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 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 [9].

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 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 1 and 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.

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

Listing 2: Example of MMIO accessor in wdm.h line 17433 that is included in the Windows Driver Kit, build version 0162

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 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.

4.4 Implementation

A prototype of *FWinst* has been implemented on Linux KVM (Linux Kernel 4.8.1) 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. *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.

4.4.1 Building the Legitimate Instruction Set

For PIO and UMIP contexts, the legitimate set of instructions is straightforward, as described in Sect. 4.3.

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 Sect. 4.2, 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.

4.4.2 Implementation of the Instruction Filter

The control and data flow between the components of the instruction emulator are depicted in Fig. 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

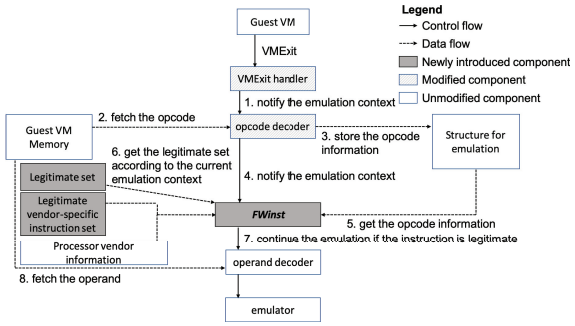


Fig. 4 The control and data flow between the components of the instruction emulator in KVM with *FWinst*.

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 complicated 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.

5. 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 2 shows the experimental environment.

5.1 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 vulnerabilities reside in the instruction emulator, which are listed in Table 3. For these vulnerabilities we have collected or implemented PoC (proof-of-concept) code and tested it on *FWinst*. As shown in Table 3, *FWinst* can defend against 14 vulnerabilities out of 17 on Haswell, (indicated by √ in column 'Haswell'), 13 vulnerabilities out of 17 on Westmere (indicated by √ in column 'Westmere'), and 13 out of 17 on AMD Jaguar (indicated by √ 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.

Table 2 Experimental Environment

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

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 column in Table 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 Sect. 4.3, 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 em-

Table 3 Summary of vulnerabilties.

CVE #	vulnable 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

ulation 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 '√', 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 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.

5.2 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 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 [12], sysbench [13], ApacheBench [14], and

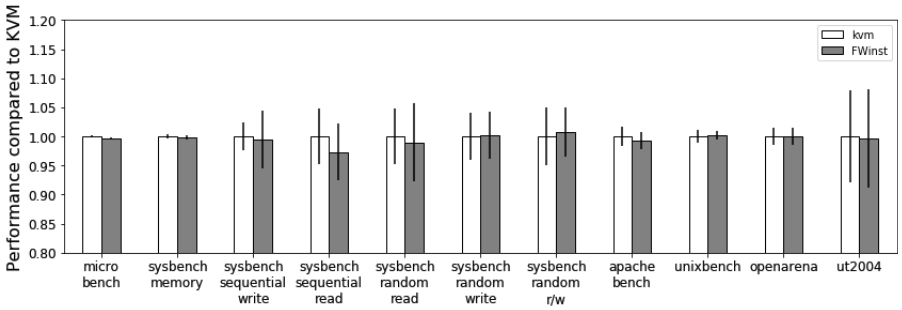


Fig. 5 Normalized performance of UnixBench, Apache Bench, sysbench, micro benchmark and graphic benchmarks on Skylake with the original KVM as the baseline

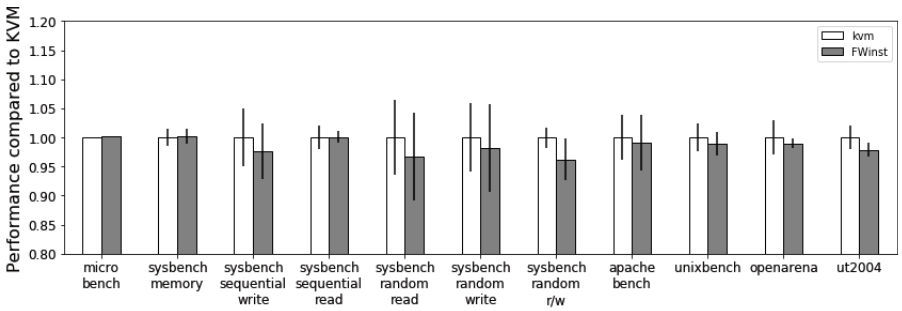


Fig. 6 Normalized performance of UnixBench, Apache Bench, sysbench, micro benchmark and graphic benchmarks on Westmere with the original KVM as the baseline

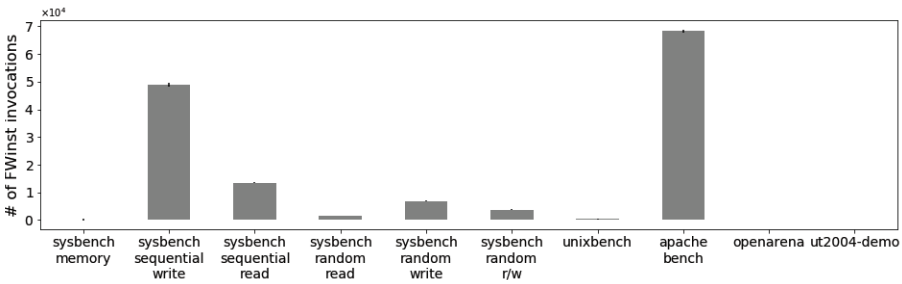


Fig. 7 # of *FWinst* invocations

Phoronix Test Suite [15] 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 5 and 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 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 Fig. 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 [16] 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.

6. Related Work

FWinst takes an approach different from other approaches to securing the hypervisor.

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 [17] 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. Monitoring hypervisors at runtime is a promising approach to improve the security of hypervisors. Dancing with Wolves [18] monitors untrusted hypervisors from a secure event-driven monitor. HyperSafe [19] and HyperVerify [20] provide runtime protection for hypervisors.

Testing the Hypervisor. Virtual CPU Validation [3] 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 exploit vulnerabilities in instruction emulators, a new attack vector is shown to force the emulator to emulate arbitrary instructions at any time. MultiNyx [21] generates test cases automatically for modern hypervisors, which rely on complex processor extensions, using the symbolic execution. To avoid the complex specification of the extensions for virtualization, MultiNyx uses the Bochs CPU emulator as an executable specification and generates 206,628 test cases that revealed many inconsistencies between different KVM configuration.

Reducing TCB. Another approach to hardening hypervisors is to reduce TCB (trusted computing base) in the hypervisor. Min-V [22] disables all unnecessary virtual devices when running VMs in the cloud. NoHype [23] eliminates the virtualization layer at runtime, and each VM directly runs on statically assigned resources. NOVA [24] takes a microkernel approach to achieving a smaller TCB. Deconstructing Xen [25] divides Xen's privileged code into per-VM slices, and confines the attacks inside the slices. HyperLock [26] prepares a shadow hypervisor for each VM and provides runtime isolation for the privileged host. DeHype [27] demotes KVM to user mode and runs it as a per-VM deprivileged hypervisor.

7. 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 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.

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.

Acknowledgements

This work was supported in part by Japan Science and Technology Agency (JST CREST JPMJCR19F3) and Japan Society for the Promotion of Science (JSPS KAKENHI JP19K11906).

References

[1] KVM, "KVM." http://www.linux-kvm.org/page/Main_Page, 2016.
[2] X.S. Team, "Xen security advisory." https://xenbits.xen.org/xsa/, 2017.
[3] N. Amit, D. Tsafrir, A. Schuster, A. Ayoub, and E. Shlomo, "Virtual CPU Validation," Proc. 25th Symposium on Operating Systems Principles, SOSP '15, New York, NY, USA, pp.311–327, ACM, 2015.
[4] Andrea Arcangeli, "Using Linux as Hypervisor with KVM." https://indico.cern.ch/event/39755/attachments/797208/1092716/slides.pdf, 2008.
[5] G.J. Popek and R.P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," Commun. ACM, vol.17, no.7, pp.412–421, July 1974.
[6] J.S. Robin and C.E. Irvine, "Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor," Proc. 9th Conference on USENIX Security Symposium - Volume 9, pp.129–144, 2000.
[7] Paolo Bonzini, "KVM: x86 emulator: emulate MOVAPS and MOVAPD SSE instructions. Linux Kernel Mailing List." https://lkml.org/lkml/2014/3/17/384, 2014.
[8] K. Ishiguro and K. Kono, "Hardening hypervisors against vulnerabilities in instruction emulators," Proc. 11th European Workshop

on Systems Security, EuroSec'18, New York, NY, USA, pp.7:1–7:6, ACM, 2018.
[9] Intel corporation, "Intel 64 and IA-32 Architectures Software Developer's Manual," 2019. Reference number: 325384-069US.
[10] Nitin A Kamble, "KVM: VMX: Support Unrestricted Guest feature. Linux Kernel Mailing List." https://lkml.org/lkml/2009/8/16/41, 2009.
[11] Nitin A Kamble, "Unrestricted guest support in VMX. Xen.org mailing list." http://old-list-archives.xenproject.org/xen-devel/2009-05/msg01196.html, 2009.
[12] "Unix Bench." https://github.com/kdlucas/byte-unixbench, 2017.
[13] "sysbench." https://github.com/akopytov/sysbench, 2017.
[14] "ab - Apache HTTP server benchmarking tool." https://httpd.apache.org/docs/2.4/programs/ab.html, 2017.
[15] Phoronix. https://www.phoronix-test-suite.com/.
[16] "The Mesa 3D Graphics Library." https://www.mesa3d.org/llvmpipe.html.
[17] J. Ogasawara and K. Kono, "Nioh: Hardening the hypervisor by filtering illegal i/o requests to virtual devices," Proc. 33rd Annual Computer Security Applications Conference, ACSAC 2017, New York, NY, USA, pp.542–552, ACM, 2017.
[18] L. Deng, P. Liu, J. Xu, P. Chen, and Q. Zeng, "Dancing with wolves: Towards practical event-driven vmm monitoring," Proc. 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '17, New York, NY, USA, pp.83–96, ACM, 2017.
[19] Z. Wang and X. Jiang, "HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity," 2010 IEEE Symposium on Security and Privacy, pp.380–395, 2010.
[20] B. Ding, Y. He, Y. Wu, and Y. Lin, "HyperVerify: A VM-assisted Architecture for Monitoring Hypervisor Non-control Data," 2013 IEEE Seventh International Conference on Software Security and Reliability Companion, SERE-C '13, Washington, DC, USA, pp.26–34, 2013.
[21] P. Fonseca, X. Wang, and A. Krishnamurthy, "Multinyx: A multi-level abstraction framework for systematic analysis of hypervisors," Proc. Thirteenth EuroSys Conference, EuroSys '18, New York, NY, USA, pp.23:1–23:12, ACM, 2018.
[22] A. Nguyen, H. Raj, S. Rayanchu, S. Saroiu, and A. Wolman, "Delusional boot: Securing hypervisors without massive re-engineering," Proc. 7th ACM European Conference on Computer Systems, EuroSys '12, New York, NY, USA, pp.141–154, ACM, 2012.
[23] J. Szefer, E. Keller, R.B. Lee, and J. Rexford, "Eliminating the hypervisor attack surface for a more secure cloud," Proc. 18th ACM Conference on Computer and Communications Security, CCS '11, New York, NY, USA, pp.401–412, ACM, 2011.
[24] U. Steinberg and B. Kauer, "NOVA: A Microhypervisor-based Secure Virtualization Architecture," Proc. 5th European Conference on Computer Systems, EuroSys '10, New York, NY, USA, pp.209–222, ACM, 2010.
[25] L. Shi, Y. Wu, Y. Xia, N. Dautenhahn, H. Chen, B. Zang, H. Guan, and J. Li, "Deconstructing Xen," Proceedings 2017 Network and Distributed System Security Symposium, Feb. 2017.
[26] Z. Wang, C. Wu, M. Grace, and X. Jiang, "Isolating commodity hosted hypervisors with HyperLock," Proc. 7th ACM European Conference on Computer Systems, EuroSys '12, New York, NY, USA, pp.127–140, ACM, 2012.
[27] C. Wu, Z. Wang, and X. Jiang, "Taming hosted hypervisors with (mostly) deprivileged execution.," The Network and Distributed System Security Symposium 2013, NDSS '13, The Internet Society, 2013.

Kenta Ishiguro received his B.E. and M.E. degrees from Keio University in 2017 and 2019. He is currently a Ph.D. student in the Department of Information and Computer Science at Keio University. His research interests include operating systems, virtualization and software security. He is a member of IEEE/CS and ACM.

Kenji Kono received the BSc degree in 1993, MSc degree in 1995, and PhD degree in 2000, all in computer science from the University of Tokyo. He is a professor in the Department of Information and Computer Science at Keio University. His research interests include operating systems, system software, and Internet security. He is a member of the IEEE, ACM, and USENIX.