On the Feasibility of Software Attacks on Commodity Virtual Machine Monitors via Direct Device Assignment

Gábor Pék
CrySyS Lab, BME
Budapest, Hungary
pek@crysys.hu

Andrea Lanzi
Univ. degli Studi di Milano
Milan, Italy
andrea.lanzi@unimi.it

Abhinav Srivastava
AT&T Labs Research
New Jersey, USA
abhinav@research.att.com

Davide Balzarotti
Eurecom
Sophia Anitpolis, France
balzarot@eurecom.fr

Aurélien Francillon
Eurecom
Sophia Antipolis, France
francill@eurecom.fr

Christoph Neumann
Technicolor, Rennes, France
christoph.neumann
@technicolor.com

ABSTRACT

The security of virtual machine monitors (VMMs) is a challenging and active field of research. In particular, due to the increasing significance of hardware virtualization in cloud solutions, it is important to clearly understand existing and arising VMM-related threats. Unfortunately, there is still a lot of confusion around this topic as many attacks presented in the past have never been implemented in practice or tested in a realistic scenario.

In this paper, we shed light on VM related threats and defences by implementing, testing, and categorizing a wide range of known and unknown attacks based on directly assigned devices. We executed these attacks on an exhaustive set of VMM configurations to determine their potential impact. Our experiments suggest that most of the previously known attacks are ineffective in current VMM setups.

We also developed an automatic tool, called PTFuzz, to discover hardware-level problems that affects current VMMs. By using PTFuzz, we found several cases of unexpected hardware behaviour, and a major vulnerability on Intel platforms that potentially impacts a large set of machines used in the wild. These vulnerabilities affect unprivileged virtual machines that use a directly assigned device (e.g., network card) and have all the existing hardware protection mechanisms enabled. Such vulnerabilities either allow an attacker to generate a host-side interrupt or hardware faults, violating expected isolation properties. These can cause host software (e.g., VMM) halt as well as they might open the door for practical VMM exploitations.

We believe that our study can help cloud providers and researchers to better understand the limitations of their current architectures to provide secure hardware virtualization and prepare for future attacks.

ASIA CCS'14, June 03–06, 2014, Kyoto, Japan.

http://dx.doi.org/10.1145/2590296.2590299.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: Security and Protection; D.2.5 [**Software Engineering**]: Testing and Debugging—*error handling and recovery*

Keywords

I/O virtualization; Virtual Machine Monitor; Passthrough; Interrupt attack; DMA attack; MMIO; PIO

1. INTRODUCTION

Due to the increasing demand towards server consolidation, virtualization has become a key element of IT infrastructures. For example, the ability to create and manage virtual servers is one of the pillars of Infrastructure-as-a-Service (IaaS) cloud services, such as Amazon EC2 [1] and Google Compute Engine [2]. For this reason, both software and hardware vendors are constantly developing and releasing new technologies to satisfy the ever-growing customer expectations in terms of security, privacy, performance, and usability.

In the last ten years, a large number of papers [3, 4, 5, 6, 7, 8] have been presented to either secure, or enhance the performance and capabilities of VMMs. Several works [9, 10, 11, 12, 13, 14] also mention and implement possible attacks. Unfortunately, most of them are described only from a theoretical point of view, and only a few have been actually implemented and thoroughly tested in realistic settings. Moreover, even when a proof-of-concept implementation exists, it is often difficult to understand what the prerequisites are for the attack to work, what the real impact is, and to which extent the results can be generalized to other environments and/or VMMs. Most of these questions are difficult to answer, and it is not uncommon also for experts to disagree on these points. Finally, to make things even more complex, current VMMs are rapidly evolving. Each new release contains new technologies that can potentially introduce new vulnerabilites as well as new countermeasures that can make existing attacks obsolete.

For example, several techniques have recently been introduced to increase the efficiency and security of I/O operation for guest virtual machines (VMs). *Direct device assignment* (also known as device passthrough) is such a mechanism, where the VMM assigns a device exclusively to one VM instead of sharing it with other virtual machines. This is achieved by directly mapping the device into a VM address space, redirecting the corresponding interrupts to the correct VM. Clearly, assigning the hardware to be directly

controlled by a VM improves the performance. At the same time, this approach also introduces a wide range of security problems that eventually led hardware manufacturers to introduce hardware assisted protection extensions for the CPU and chipset.

In this paper, we demonstrate a wide range of known *and* unknown attacks that can be launched via device passthrough. First, we chose to re-implement attacks that have been proposed by security researchers in order to systematically study their impact and their limitations against recent versions of Xen and KVM virtual machine monitors. We believe that repeating experiments is fundamental in computer science, as well as in many other scientific fields (e.g., physics) to validate the results of different researchers. For this reason, we carefully replicated our tests under seven different VMM configurations.

Second, we complemented the existing attacks by exploring new directions and unknown corner cases. In particular, we propose two *novel* attacks, one based on the modification of the Peripheral Component Interconnect express (PCIe) configuration space and the other based on the creation of host-side Non-Maskable Interrupts (NMIs). More precisely, our interrupt attack is the consequence of a misunderstanding between the hardware and software. In addition, it is the only interrupt attack to date that works on configurations in which all available hardware protections are turned on. By discussing our results with vendors, we realized that our attack is pervasive and especially affects Intel based server platforms.

To perform our experiments, we implemented several tools designed to reveal configuration weaknesses, VMM vulnerabilities, or deeper hardware problems. Some attacks were manually tested, for example by remapping specific I/O memory ranges and trying to read or write them. In other cases, however, it was impossible to manually cover the space of all possible values, so we implemented a fuzzer, called PTFuzz, to thoroughly explore different problems.

In summary, this paper makes the following contributions:

- We re-implement a wide range of previously proposed attacks to evaluate the threat that they carry on contemporary VMMs.
- We introduce two *novel* attacks for *passthrough devices*: A new variation of an attack against the PCIe configuration space *and* an interrupt attack that violates the security settings in all tested configurations. While the former was discovered manually, the latter was revealed by a fuzzer, called PTFuzz, that we built to automatically reveal low-level problems during DMA operations. In addition, PTfuzz revealed another unexpected hardware behaviour during testing interrupt attacks.
- We test all the attacks on various configurations of two commodity VMMs (Xen and KVM), and discuss how different features contribute to the security of these VMMs. On the one hand, our experiments show that it can be quite difficult to properly configure a VMM to operate securely. In addition, we show that in some circumstances the overall security can only be guaranteed by disabling device passthrough for untrusted guests. On the other hand, once the system is properly configured, most of the attacks (except our new interrupt attacks) are either ineffective or restricted to the attacker's virtual machine.

2. BACKGROUND

A virtualized environment consists of three main software components: the host operating system (or privileged VM), a number of guest operating systems running inside isolated virtual machines (VMs), and a Virtual Machine Monitor (VMM) responsible for controlling the access to hardware resources[1]. In reality, the distinction between these three components is not always clear. For example, in Type I VMMs (such as Xen), there is no host OS but only the VMM and a privileged VM (e.g., Xen's Dom0) that manages other unprivileged guest virtual machines. Type II VMMs (such as KVM), however, include a host OS which also contains the hypervisor in charge of uniformly managing all the system resources.

Prior to the introduction of specific hardware support, the execution of unmodified guest OSs (i.e., full virtualization) was implemented by performing instruction emulation, for example, by using *binary translation*. Performance was further improved by the introduction of *paravirtualization*, however, this requires a modified version of the guest OS.

As these pure software solutions have various weaknesses in terms of scalability, performance and security, a new approach called *hardware assisted virtualization* was introduced by AMD/V [15] and Intel-VT [16] technologies.

In the rest of this section, we will introduce the main technologies (both hardware and software) that are required to understand the security of VMMs and the experiments we present in this paper.

2.1 Direct Device Assignment

One of the main tasks of the VMM is to control how guest virtual machines can access physical I/O devices. Three main approaches exist to perform this task: emulation, paravirtualization and direct device assignment (also known as *direct access* or *passthrough*). The first two techniques share virtualized I/O devices among multiple virtual machines. On the contrary, the passthrough approach assigns one physical device exclusively to one VM that has full control and direct access to most parts of the assigned hardware. This has the advantage of significantly reducing the main bottleneck of virtual environments: the overhead of I/O operations [17, 18, 19, 20, 21]. Unfortunately, direct device assignment also raises several security concerns. In fact, bus mastering capable directly assigned devices can initiate DMA transfers, and have access to arbitrary memory locations [10]. A malicious virtual machine may misuse them to access the memory of other VMs. Moreover, buggy device drivers can also be the root cause of system instability [22, 23, 24, 25]. These problems can be mitigated by using an IOMMU, as explained later in this section.

2.2 PCI/PCIe Address Space Access

On the x86 architecture, a PCI device can be accessed in two different ways: using a Port Mapped I/O (PIO) or using a Memory Mapped I/O (MMIO) mechanism. Each PCI device configuration is stored in the device configuration memory. This memory is accessible either by using special PIO registers or through an MMIO space. The configuration space is typically accessed by the BIOS or the operating system kernel to initialize or configure the Base Address Registers (BAR). Base Address Registers are defined by the PCI standard and used to specify the address at which the device memory is mapped in the PIO or MMIO address spaces.

Access to configuration space registers is usually emulated for fully virtualized guests, and in some cases also for privileged VMs. In this case, whenever a guest accesses a configuration space, the request is intercepted by the VMM, which incurs a significant performance overhead. Therefore, in order to improve the performance, some VMMs (e.g., KVM) allow to directly pass PIO or

[1]We deliberately use the term *VMM* instead of *hypervisor*, as the latter traditionally is not capable of providing a full-fledged guest environment and does not support multiple VMs.

MMIO accesses [21], except for the accesses targeting the device configuration memory.

2.3 Hardware IOMMU

To improve the isolation, performance, and security of I/O devices, hardware supported I/O Memory Management Units (IOMMUs) were introduced [26]. In particular, Intel VT-d [27] provides hardware support for DMA and interrupt virtualization. DMA virtualization (a.k.a. DMA remapping) enables system software to create multiple isolated DMA protection domains by allocating a subset of the host physical memory to a specific domain.

The DMA isolation is performed by restricting memory access from I/O devices to a specific physical memory domain set. More precisely, the DMA isolation mechanism is able to exclusively assign a set of physical pages to a particular I/O device. For this purpose, the VT-d architecture defines a multi-level page table structure for the DMA address translation.

2.4 Interrupt Remapping

In a multiprocessor environment, hardware interrupts are handled either by the per-processor Local Advanced Programmable Interrupt Controller (LAPICs) or by the external I/O APIC, which is part of the system chipset. LAPICs are responsible for receiving and sending local interrupts from various sources, including special pins in the processor, the APIC timer, or the performance-monitoring counters. LAPICs can also receive interrupts via the Inter-Processor Interrupt (IPI) mechanism to get notifications from other processors. Finally, interrupts can also originate from external interrupt sources (e.g., I/O devices) that are connected to the I/O APIC. In this case, the I/O APIC translates these requests to corresponding interrupt messages using its *redirection table*, and delivers them to the target LAPIC. The LAPIC then decides whether to forward the interrupt to the processor or not.

There are two main types of interrupts that can be generated by I/O devices: legacy interrupts and Message Signaled Interrupts (MSI). Legacy interrupts use dedicated wires, while MSI interrupts use an in-band mechanism. MSIs are created by a DMA write to the memory mapped LAPIC region as step (1) and step (2) of the *numerical* path in Figure 3 show it. Such interrupts encode their own attributes (interrupt vector, destination processor, delivery mode, etc.) into the address and data of the DMA request. Basically, it means that a guest VM with a directly assigned device can also use this mechanism to signal an MSI to a physical processor via a simple DMA write operation. However, this can be fatal for a system as an arbitrary interrupt vector could be called from unprivileged guests [9].

To protect against such interrupt-based attacks, Intel introduced the interrupt remapping architecture as a part of the VT-d technology (see block *IR* in Figure 3). This mechanism is responsible for isolating and routing device interrupts to corresponding virtual machines. More precisely, when VT-d is turned on, all the interrupt requests (both MSI and legacy interrupts) are extended with a *source-id* attribute to identify the device that issues the interrupt request. Moreover, the *new, remappable* MSI format uses only simple attributes such as the *index* value that points out an entry in the so-called *interrupt remapping table* to find a requested interrupt vector. In this way, the device cannot call an arbitrary interrupt vector directly, but only after the validation of the hardware. After successful validation, interrupts are remapped by the hardware module to the corresponding physical interrupt vectors with the proper attributes (e.g., delivery mode). However, if Extended Interrupt Mode (x2APIC mode) [27] is disabled, *old, compatibility*-format MSI interrupts can still be generated if VMM software enables them during runtime.

2.5 A Closer Look at Commodity VMMs

Since direct device assignment has a large impact on the overall system security and performance, we discuss that in more detail in the case of the two VMMs we use in our tests: Xen and KVM.

Xen

Xen supports two types of passthrough modes: the software-only Xen PCI passthrough for paravirtual guests, and the hardware IOMMU based passthrough available for both paravirtualized and fully virtualized (HVM) guests. The software-only paravirtual Xen PCI passthrough requires VMM intervention to handle DMA requests [28]. From a security point of view, software-only paravirtual Xen PCI passthrough gives full control over the device, which allows a compromised paravirtualized guest to write arbitrary machine memory locations using a DMA attack. In contrast to that, the hardware IOMMU based passthrough allows for remapping all the DMA requests by the corresponding DMA remapping hardware units. As Xen enables device passthrough for HVM guests only when hardware based IOMMU is turned on, it is a more secure solution. In this paper, we test only the hardware *IOMMU based* passthrough model as this is the preferred configuration in current public clouds. In addition, the software-only paravirtual passthrough is well-known to be insecure as it does not use a hardware IOMMU, we therefore do not evaluate it in this paper.

KVM

KVM guests normally use either emulated devices or virtio devices to perform I/O operations. Virtio is an efficient I/O virtualization abstraction layer that allows the development of paravirtualized drivers for guests. Similarly to Xen, direct device assignment is only allowed when the hardware based protection (either Intel VT-d or AMD-V) is turned on. In this setting, PCI/PCIe devices can be assigned directly to a guest VM, thus allowing these devices to be used almost at native speed with minimal I/O overhead.

3. SETUP AND TEST CONFIGURATION

In this paper, we describe the implementation, validation, and execution of three classes of software-based virtualization attacks based on direct device assignments. The attacks we present are based on abusing the PCI/PCIe configuration space to generate device memory collisions, on performing unauthorized memory access through DMA and MMIO regions, and on generating interrupts to the VMM from a guest VM.

3.1 Threat Model

In our experiments, we run each attack against different hardware and VMM configurations, according to two possible attack scenarios. In the first scenario, we assume that the attacker has full access to a guest machine configured with a pass-through device. This is a common setup for IaaS cloud providers that offer, for example, direct access to video cards (e.g., Amazon EC2 Cluster GPU).

In the second scenario, we assume that the attacker is able to control or compromise the privileged VM (e.g., by exploiting vulnerabilities such as CVE-2008-3687, or CVE-2007-4993). Even though this case is certainly more difficult to achieve, it still represents an important threat model that needs to be carefully evaluated. In fact, the privileged VM is often a large piece of software, therefore, it is more prone to contain vulnerabilities that can be exploited by an attacker. However, unrestricted access to the privileged VM

does not give full privileges over the physical machine [10]. For example, VMMs (e.g., Xen) execute the privileged VM (Dom0) with the role of a host OS, and strictly divide the physical address space and privileges between Dom0 and the VMM. In other words, the VMM is specifically designed to be protected against a malicious privileged VM.

We also launched our attacks on KVM's host OS to identify the differences with Xen's Dom0. However, we performed this test only for completeness, as host OS privileges on KVM are equivalent of having entire control over the VMM as well.

3.2 VMM and Hardware Setup

For our experiments, we installed the last versions of Xen and KVM available at the time we ran our tests (i.e., Xen 4.2 and KVM 3.5). Table 1 shows the default boot settings for both VMMs and the configurations we used in our tests.

Xen 4.2 came with a set of new virtual machine management tools named XL that support passthrough devices by default. KVM enables both DMA and Interrupt remapping by default, as well as the x2APIC mode if it is supported by the hardware. x2APIC combined with interrupt remapping is known to be a *secure* configuration against interrupt attacks [9]. Device passthrough is instead not enabled by default, but it can be manually turned on by the administrator.

Properly configuring a VMM can be a daunting and confusing task, even for experienced system administrators. For example, hardware IOMMU requires the support from the CPU, the BIOS, and the motherboard chipset. Unfortunately, these dependencies are not always correctly documented for VMMs. For example, our original test machine was equipped with an i5-2400 CPU (with IOMMU support), but contained a BIOS and chipset (H67) without IOMMU support. Both KVM and Xen refused to put our network card in passthrough mode in this setup. However, after updating the BIOS to a new version with IOMMU support, KVM let us put the device in passthrough mode. This means that in practice KVM trusted the Advanced Configuration and Power Management Interface (ACPI) data structures reported by the updated BIOS, but did not pay attention to the real chipset capabilities. This may carry serious threat for integrated devices which have no PCIe Function Level Reset (FLR) capabilities [29]. More precisely, an integrated device assigned to a guest VM can be configured to send DMA requests continuously to memory addresses belonging to the host OS/privileged VM. While the device is assigned to the guest VM, all these requests are blocked by the IOMMU hardware. When the device is reassigned to the host OS/privileged VM, the VMM first resets the device via an FLR or bridge-level reset, and then reconfigures the IOMMU hardware to allow device access to the host OS/privileged VM memory. However, if the VMM cannot reset the device (e.g., FLR capability is missing), the device keeps sending DMA requests which now overwrite host OS/privileged VM memory addresses.

In conclusion, KVM lets the administrator believe the device is in a secure passthrough mode, while the support is actually incomplete. To prevent this issue, system administrators are advised to carefully check whether the CPU, the BIOS *and* the motherboard chipset support hardware IOMMU. This is a clear example of configurations problems related to properly setting up passthrough devices. For this reason, we executed our attacks on *another* machine with Intel i5-2500 CPU, Q67 chipset and Intel VT-d support enabled in the BIOS. The machine was equipped with an Intel 82579LM PCIe network card that was used as passthrough device for the experiments. The guest OSs were running Linux kernel 2.6.32-5-amd64 and 3.5.0.

3.3 Results Overview

In the rest of the paper, we use an abbreviated notation to refer to the scenario and the target of each attack. We use the following notation:

- Guest-to-VMM (**g⇒vmm**) attacks.
  This is the most critical scenario, in which an attacker controlling a guest VM can launch an attack against the system VMM.

- Guest-to-Host (**g⇒h**) attacks.
  An attacker, with full control on a guest virtual machine, can compromise the host OS/privileged VM. Even though it is not as powerful as the previous one, the consequences of this attack are often as serious.

- Host-to-VMM (**h⇒vmm**) attacks.
  As we explained above, the VMM is often separated and protected from the privileged guest VM (e.g., Xen Dom0). In this attack, an attacker who was able to gain control in the privileged guest OS can escalate her privileges in the machine by launching an attack against the VMM. As we highlighted before, we tested our attacks on KVM host OS only to show the differences with the Xen Dom0 case.

- Guest-to-Guest (**g⇒g**) attacks.
  These attacks aim at compromising a target guest VM starting from a separate guest VM controlled by the attacker.

- Guest-to-Self (**g**↺) and Host-to-Self (**h**↺) attacks.
  These are less severe scenarios in which the attack is confined inside the attacker's virtual machine. For this reason, we also refer to these attacks as *intra-guest* and *intra-host* attacks. In most of the cases, since the attacker has already root privileges on the same virtual machine, the consequences of these attacks are negligible. However, we still report these cases for completeness. Also, they may still be relevant if the attacker does not fully control the machine, but can exploit a vulnerability that allow her to run the attack, or the system has to provide some security properties in an untrusted environment (e.g., as presented by Zhou et al. [30]).

The previous classification is based only on the originator and target of each attack. We intentionally use this approach, instead of a more traditional one based on the possible consequences of each attack (e.g., Denial of Service (DoS), code execution, information leakage, etc), because it better summarizes the results of our tests in terms of violating the *isolation* property. In fact, our main goal is to clarify which attack works under which conditions. The fact that it can be later used to DoS the VMM, or steal some information from other virtual machines, highly depends on other environmental conditions that are not under our control (e.g., the presence of other vulnerabilities in the system, and the capabilities and motivation of the attacker). However, we will briefly describe and comment on the possible consequences of each successful attack, case by case, in the next three sections.

All the attacks, presented in this paper are summarized in Table 2. The second column (*Ref*) reports the original source in which the attack was first proposed. The table also shows the *Previous Status* of the attack, (i.e., whether we found any evidence that the attack had been successfully implemented before our study). A value of *NC* (not confirmed) means that the attack was only theoretically described, *C* (confirmed) means that it was already tested in practice by other authors, and *NEW* means that the attack is proposed for the first time in this paper.

Feature	Default Values		Our Xen Setup					Our KVM Setup	
	Xen	KVM	Dom0	HVM-1	HVM-2	HVM-3	HVM-4	Host OS	HVM-1
VT-d DMA Remapping	✓	✓	✓	✓	✓	✓	✓	✓	✓
VT-d Dom0 DMA passthrough	✗	✓	✓	✓	✓	✓	✓	✓	NA
VT-d interrupt remapping	✓	✓	✓	✗	✗	✓	✓	✓	✓
Direct Configuration Space Access	✗	NA	✗	✓*	✗	✗	✗	✓	✗
x2APIC mode on the Host	✓	✓	✓	✗	✗	✗	✓	✓	✓

Table 1: Test Configurations. The table shows both the default and test configurations in our Xen 4.2 and KVM 3.5 setup on an Intel Core (Quad) i5-2500 CPU 3,30 GHz CPU and Q67 motherboard chipset. While the *Default Values* column is placed only for comparison purposes, the *Our Xen Setup* and *Our KVM Setup* columns summarize those configurations we tested our attacks on. For example, the HVM-4 configuration on Xen means that all the hardware protection features were enabled in this configuration, but we did not give direct configuration space access to the passthrough device being attached to the Xen guest VM. The sign *NA*, refers to a cell that cannot be evaluated in the given configuration.
*Direct PIO access to device configuration memory was granted explicitly.

Attack	**Ref.**	**Previous Status**	**Xen** Host	HVM-1	HVM-2	HVM-3	HVM-4	**KVM** Host	HVM-1
PCI/PCIe Configuration Attacks									
PCI/PCIe config. space access (PIO)	[31]	C*	h↺	g↺	g↺	g↺	g↺	h↺	g↺
PCIe config. space access (MMIO)	-	**NEW**	h↺	NA	g↺	g↺	g↺	h↺	g↺
I/O port overlapping (PIO)	[30]	NC	h↺	g↺	g↺	g↺	g↺	-	g↺***
I/O memory overlapping (MMIO)	[30]	NC	h↺	NA	g↺	g↺	g↺	h↺	g↺
Unauthorized Memory Access									
Unauthorized MMIO memory region access	[30]	NC	h↺	g↺	g↺	g↺	g↺	h↺	g↺
DMA	[10]	C**	h↺	g↺	g↺	g↺	g↺	h↺	g↺
Interrupt Attacks									
NMI	-	**NEW**	h↺	*g⇒vmm*	*g⇒vmm*	*g⇒vmm*	*g⇒vmm*	*g⇒vmm*	*g⇒vmm*

Table 2: Overview of the results of the attacks implemented in our study.
* The attack was previously confirmed as h↺, g⇒h, and g⇒vmm against an old version of Xen.
** The attack was previously confirmed as h⇒vmm, against an old version of Xen without DMA remapping hardware.
*** KVM detects the port overlapping and kills the guest. The state of CPU registers is also dumped in the Host OS.

Each cell in the table lists the results of a given attack against a particular configuration of Xen and KVM, according to the abbreviations introduced above. Whenever the attack was not applicable, it is marked as *NA*. Finally, we mark in red color the most critical results, i.e., any successful guest-to-host or guest-to-VMM attacks. We do not highlight here the unexpected hardware behavior we discovered while carrying out interrupt attacks. The unexpected behaviors are described in Section 6.

The next three sections present an in-depth analysis of the three classes of attacks and the result of our experiments.

4. DEVICE MEMORY COLLISION ATTACKS

In this section, we systematically examine how a VM can access the configuration space of directly assigned devices, and what security risks are associated to these actions. In particular, by reconfiguring a device, an attacker can create a *conflict* with another device. Therefore, without enforcing the proper PCI/PCIe configuration space access restrictions, an attacker could ex-filtrate data from one virtual machine to another that is under her control. Moreover, unmediated PCI/PCIe configuration space accesses can result in either privilege escalation [31] or DoS attack against the host/privileged VM. All the attacks we present are implemented by modifying the *e1000e* device driver of an Intel 82579LM *PCIe* network card.

PIO Attack against the PCI/PCIe Configuration Space

Duflot et al. [31] described the dangers of enabling a VM to directly access the configuration registers of a PCI device via the PIO space. The authors showed that delegating PIO access to userspace or a guest virtual machine can lead to several attacks, such as privilege escalation between isolated virtual machines.

According to the literature, KVM emulates PIO accesses (using QEMU code [32]), thus, accessing directly the configuration space I/O ports is not possible in normal circumstances. Xen originally allowed direct access to the PIO configuration space from Dom0, but this was eventually emulated as well [33]. However, direct access can still be allowed to fully virtualized guests (HVM) via a guest configuration option. There is no clear conclusion still (e.g.,[34]) whether certain guest VM configurations are insecure if direct PCI/PCIe configuration space accesses are enabled for passthrough devices.

For this reason, we decided to test this condition to show what the real risks are in a practical scenario. In our experiments, we modified the e1000e network driver to overwrite the PIO Base Address Register (BAR) in the configuration address space of arbitrary

devices (identified by their *source-id*) by writing to the `0xcf8` I/O port. The goal of our test was to address PCI devices that were invisible from our VM. We tested our attack in various setups as shown in Table 2, but all cases turned out to be ineffective on current Xen and KVM versions. The reason is that PIO configuration space accesses are always emulated, no matter what configurations we used. In other words, we could access only the devices of our VM, therefore restricting the attack to an *intra-guest* or *intra-host* scenario.

MMIO Attack against the PCIe Configuration Space

PCI Express (PCIe) devices have an extended configuration space that can be accessed via traditional memory operations (i.e., MMIO). To test this situation, we implemented a *new* device configuration space attack that can be launched via the MMIO with the goal of manipulating the memory mapped registers of the target device.

Similarly to the PIO access attack, we addressed the PCIe configuration space of the targeted devices by using their source-id [35], and then tried to modify some of their configuration registers (e.g., BAR). Again, we were not able to address devices that were not in the scope of our VM, thus the attack is limited to an *intra-guest* or *intra-host* scenario.

PIO Overlapping Attack

Zhou et al. [30] proposed several device-related attacks that could affect already compromised operating systems. For example, PIO overlapping is similar to the PIO configuration space attack, but in this case the attacker can only reconfigure the configuration space of a specific device she controls. In particular, by changing the PIO BAR register of a directly assigned PCI/PCIe device, an attacker can overwrite the BAR value with the one of another device attached to another VM. In this way, the device memory of the two devices will overlap, leading to data ex-filtration from one device to the other.

To test this attack, we modified our e1000e PCIe network card driver by changing the PIO BAR value of the card with the value of the keyboard. We observed unresponsive keyboard and mouse under Xen both in case of Dom0 (with dom0-passthrough mode enabled) and HVM guests. On a KVM host, however, the port overlapping was successful but without any apparent effect. Interestingly, when the attack was launched from the KVM guest (HVM-1), the host kernel detected our overlap attempt, and killed the guest VM instantly. Additionally, the host OS provided a clear debug message about a hardware error that occurred during the registration of the I/O port read operation for the keyboard.

MMIO Overlapping Attack

In an MMIO overlapping attack (Figure 1), an attacker controlling one guest VM with an assigned passthrough device can access the device memory space of another device attached to another VM.

The attack was implemented by changing the MMIO BAR values of our PCIe passthrough network card to overlap with the BAR value of a graphics card. In all configurations, we observed that the Ethernet card became unresponsive inside the attacker VM. In contrast with the previous attack, we did not find any mechanisms implemented in Xen and KVM to notify users about these overlapping I/O memories.

To summarize the results, all the configurations that we tested are protected against PIO/MMIO configuration space manipulations in both Xen and KVM.

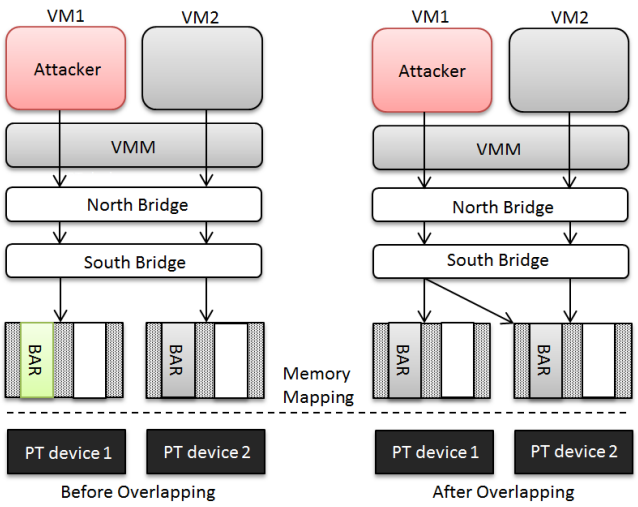

Figure 1: MMIO overlapping attack. An attacker on VM1 sets the MMIO Base Address Register (BAR) value of a passthrough device (PT device1) to that of another device (PT device 2), assigned to VM2. As a result of this manipulation, the attacker can get access to the device memory of PT device 2.

5. UNAUTHORIZED MEMORY ACCESS

In this section, we describe two types of attacks to access protected memory regions via unauthorized memory device requests. These attacks can be used to steal and ex-filtrate information in a cloud environment, or to control devices assigned to other VMs.

Unauthorized MMIO Region Access

A device can be accessed through physical memory in two ways: using MMIO and using Direct Memory Access (DMA). As the authors mention in [30], an attacker can manipulate the device behavior by writing into these memory regions. The attack can be accomplished in two steps: 1) the attacker remaps an MMIO region, belonging to a victim device, into a new virtual address by using `ioremap_nocache` Linux kernel function, 2) she injects malicious code/data into the remapped memory region by using the `iowrite32()` Linux kernel function. By doing so, the memory of the victim device is manipulated and can be controlled by the attacker.

In order to test the protection mechanisms offered by Xen and KVM, we implemented a proof-of-concept attack in which we overwrote the entire MMIO address space of a *second* network adapter by modifying the e1000e driver of the *attacker* network card. As the second adapter, which had Internet connection, was assigned to the same VM as the attacker network card, we could access its MMIO space. In our tests on Xen and KVM guests, the second adapter's Internet connection was lost and the guests became isolated from the network.

The same experiment was also performed on our KVM host, and it completely crashed the operating system. Therefore, we can conclude that none of the tested VMMs implemented a detection technique for checking these types of unauthorized memory region modifications. However, these attacks only work in *intra-guest* and *intra-host* scenarios.

DMA Attack

User-space processes are prevented from accessing protected memory regions by a memory controller known as Memory Manage-

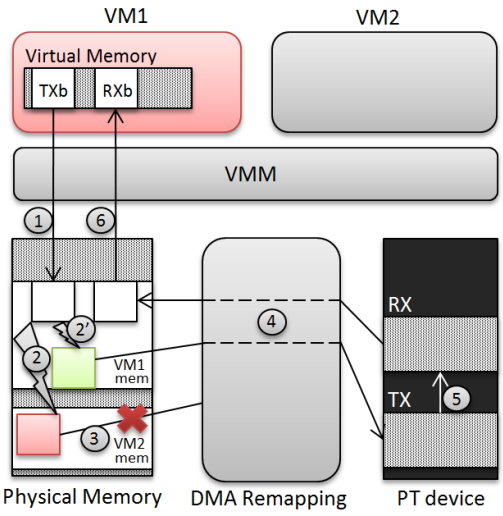

Figure 2: Intra-guest DMA attack. An attacker, controlling VM1, can read/write arbitrary intra-guest memory locations by a loopback mode passthrough device (PT device).

ment Unit (MMU). However, drivers that communicate to PCI/PCIe devices can directly access any physical memory address space by using Direct Memory Access (DMA) operations. Since the physical address space includes the whole system memory, a malicious driver could potentially read and write the entire memory of the system. To prevent this from happening, hardware vendors designed an hardware protection mechanism (called IOMMU). As we already explained in Section 2, IOMMU is a memory management unit that maps I/O bus addresses to physical memory addresses for all DMA memory transactions on the bus. The role of the IOMMU is similar to that of a traditional MMU: 1) it translates the memory I/O address range of one device to the corresponding real physical address, and 2) it prevents any unauthorized access from one device address space to another. Even though the IOMMU establishes memory barriers among different device address spaces, which parts of the memory should be assigned to which devices remains an open research problem [36]. Since it is not clear how different VMMs deploy such a protection mechanism, we implemented and tested different DMA attacks under three main threat models: intra-guest, guest-to-guest and guest-to-host.

To launch the attack, we put our passthrough network card into loopback mode similarly to Wojtczuk [10] to directly connect the card's internal transmission buffer (TX) to the receiving buffer (RX) (i.e., to receive all the packets that were sent by the card). We then associated a transmission (TXb) and a receiver buffer (RXb) to the DMA channel by using the `dma_map_single()` function. This allows us to link the guest virtual address of the transmission buffer (TXb) to the physical address that we want to access (Figure 2).

After the setup phase was completed, we performed two different experiments. First, we tried to perform a DMA operation using a physical memory location belonging to a different virtual machine. In this case (points 2 and 3 in Figure 2), the DMA Remapping Engine (DMAR) successfully blocked the attack as one VM is not allowed to access the memory of other VMs for I/O operations. We then repeated the operation using a physical address belonging to another driver that runs inside the attacker VM (step 2' in Figure 2). This time, the DMA operation succeeded as the DMA remapping was not setup to create intra-guest protection domains. As a result (points 4 to 6 in Figure 2), the stolen data was transferred into the card's internal TX buffer, and then into the RX buffer thanks to the loopback mode. Finally, another DMA transaction moves the content of RX buffer to the receiver buffer in the VM (RXb), making the data accessible to the driver (and therefore to the attacker). A similar approach can also be used to perform an arbitrary memory write operation.

We implemented our DMA attacks by extending the e1000e network driver. It is important to note that *DMA remapping was always turned on* during our attack, otherwise we could not have put the network card into passthrough mode.

We were able to successfully run *intra-guest* and *intra-host* DMA attacks on both VMMs to retrieve the code of the kernel pages of the guest OS. Considering the *guest-to-guest* case, we verify that both VMMs implement DMA remapping correctly and does not allow inter-guest DMA operations. We obtained the same results for guest-to-host attacks, as the protection isolated the host address space, and the guest was redirected to its own address space.

6. INTERRUPT ATTACKS

In this section, we present a number of interrupt-related attacks. In particular, we introduce a novel attack that evades *all* currently available hardware and software protection mechanisms. After reporting this attack to vendors, we concluded that the problem stems from a misunderstanding between the hardware and software, and cannot be resolved by existing technologies without limiting platform capabilities. We also describe other unexpected hardware conditions that we discovered by fuzzing DMA operations.

Interrupt attacks could be used to DoS a VMM or a privileged VM, or, in more severe cases, even to execute arbitrary code inside them.

6.1 Abusing Message Signalled Interrupts

Recent PCIe devices generate interrupts by using the Message Signalled Interrupt (MSI) technology. Wojtczuk et al. [9] demonstrated that MSIs can also be generated by enforcing a device's scatter-gather mechanism during a DMA transaction by writing to a specific MMIO space that belongs to the LAPIC (0xfeexxxxx). In this case, the passthrough device writes specific information (e.g., interrupt vector, and delivery mode) to that predefined interrupt address range, which results in an interrupt generation with the preset interrupt vector. From an attacker's point of view, MSIs are advantageous as they can be generated without the need to compromise the firmware of a device: only the device driver has to be under her control. Practical interrupt attacks via passthrough devices have already been discussed in previous works [9], where the authors showed how an attacker could execute arbitrary code with VMM privileges by invoking Xen hypercalls from a guest OS.

However, interrupt remapping introduced remappable-format MSIs, which prevent an attacker from generating MSI interrupts with arbitrary interrupt vectors (i.e., compatibility-format MSI) as the *numerical* path on Figure 3 shows it. The only way to entirely forbid the generation of compatibility-format MSIs is to switch on the x2APIC mode (Section 2.4). In our experiments, we observed that x2APIC mode is turned on by default in KVM, but needs to be manually selected in older versions of Xen. This is considered to be the most secure configuration. To the best of our knowledge, we present here the first attack that succeeds when both interrupt remapping and x2APIC mode are enabled.

Fuzzing the Interrupt Generation

To test for the presence of low-level problems in the interrupt generation and handling phase, we designed and implemented a tool called PTFuzz, by extending Intel's e1000e network driver. PT-

Fuzz is optimized to launch any type of MSI by fuzzing both the MSI *address* and its *data* components as well as the size of DMA requests. It works by writing data (i.e., MSI data component) to the LAPIC MMIO range using DMA. As PTFuzz is capable of fuzzing each field of an MSI separately, it can be fine-tuned to create both *compatibility* and *remappable* MSIs formats. The operation of PTFuzz can be summarized in a few steps (see Figure 2 for context information):

1. Prepare a transmission buffer (TXb) in the guest OS, and populate it with the MSI *data* component.
2. Prepare a receiver buffer (RXb) in the guest OS.
3. Change the physical address of the RXb buffer according to the MSI *address* component to point to the memory mapped interrupt space (i.e., MMIO LAPIC).
4. Move the MSI data component via a DMA transaction into the card's internal TX buffer.
5. Send the data in loopback mode into the card's RX buffer.
6. Move the MSI data from the card's internal RX buffer into the corresponding MMIO LAPIC address range specified by the MSI address (0xfeexxxxx) with a given DMA request size.
7. If the MSI data component is fuzzed, then select a new MSI data value and repeat from Step 1.
8. If the MSI address component is fuzzed, then select a new MSI address value and repeat from Step 3.

Fuzzing the entire MSI data and address spaces would require an extensive amount of work to manually verify and validate each result. For this reason, we decided to focus our effort on those MSI fields that were either more interesting from an attacker's point of view, or had clear constraints set by the vendors.

In particular, here we present the results we obtained by fuzzing the *vector* field of a compatibility format MSI *data* component and the *don't care* field of a remappable format MSI *address* component. Whenever we observed an unexpected hardware behavior as a result of our test cases, we instrumented the code of the VMM to collect all the information required to understand the problem in detail. The following two sections discuss our results.

6.2 Interrupt Vector Fuzzing

In our first experiment, we fuzzed the *vector* field of the compatibility format MSI *data* as well as the size of the MSI request. During the tests, we noticed that the VMM/privileged VM received a *legacy* Non-Maskable Interrupt (NMI) for some values of the *vector*. This happens even when all the existing hardware protections mechanisms were turned on. In addition, we got the same results when the size of the MSI request had not conformed with the required MSI transmission size (i.e., it was not 32-bit long).

Non-Maskable Interrupts are normally generated as a result of hardware errors that must be immediately handled by the CPU, in order to prevent system damage. From an architectural perspective, NMIs are exceptions and not interrupts. This is a subtle, but very important difference. Interrupts are asynchronous events that are handled when the CPU decides to do so. On the contrary, exceptions are synchronous events that are served instantly. In our case, the most important difference is that devices do not extend NMIs with the source-id information. As a consequence, NMIs are *not* subject to interrupt remapping. This is a very significant point.

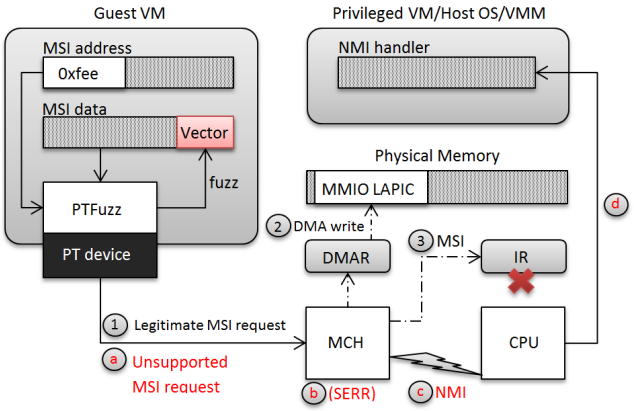

Figure 3: Interrupt generation by PTFuzz. This figure describes two interrupt generation cases indicated by the *numerical* and *alphabetical* paths. On the *numerical* path, PTfuzz requests a legitimate MSI (1) by a DMA write operation to the MMIO LAPIC (2) which is first verified by the DMA remapping engine (DMAR). As a result, a compatibility-format MSI is generated (3) that is blocked by the interrupt remapping engine (IR). The *alphabetical* path, however, shows *our* unsupported MSI request (a), which the platform detects and blocks. However, when System Error Reporting is enabled, the platform sets the SERR status bit on the Memory Controller Hub (MCH) PCI Device (b). As a result, a host-side Non-maskable Interrupt (NMI) is *directly* delivered to the physical CPU (c) executing the privileged VM/host OS/VMM. SERR induced NMIs, however, may cause host software halt or trigger the host-side NMI handler (d) which opens the door for Guest-to-VMM escapes.

Another key observation is that we did not generate an MSI that was delivered as an NMI. Our tests indirectly generated a *host-side legacy* NMI to one of the physical CPUs (i.e., Bootstrap Processor - BSP). More precisely, as a result of performing an unsupported MSI request by a DMA transaction to the memory mapped interrupt space, the platform blocks the MSI request and raises a PCI System Error (SERR#) which is delivered as NMI to report hardware errors. In our case, the SERR status bit is set by the platform on the Memory Controller Hub - MCH (BDF 00:00.0) PCI device. Thus, the unchecked host-side NMI is forwarded to the physical CPU executing the privileged VM/host OS/VMM. Depending on privileged VM/host OS/VMM kernel configuration, such an NMI may be handled by the privileged VM/host OS/VMM or can result in a host software halt (*panic*). Figure 3 gives a high-level overview about the attack. When we took a closer look at this issue, we noticed that the NMI was spawned when a compatibility format MSI is requested with vector numbers below 16 or with an invalid request size (i.e., not 32-bit long). The reason for the former lies in the fact that MSI cannot deliver interrupts with vector less than 16 [16].

All these operations are executed at the chipset level, and it took a considerable amount of time and effort to understand all the details. After discussing the problem with the Xen security group and Intel Product Security Incident Response Team (Intel PSIRT), we concluded that we identified a platform problem that affects all the machines which enable System Error Reporting. As System Error Reporting is an essential feature on server machines to report legitimate hardware errors for providing Reliability, Availability, Serviceability (RAS), this attack seriously threatens the main se-

curity feature of hardware virtualization: the isolation of virtual machines.

The *main* difference with previous interrupt attacks is that our NMI injection attack works on configurations where interrupt remapping is enabled. In fact, the DMA Remapping Engine cannot protect against our DMA write as the attacker intends to manipulate only legitimate, intra-guest physical addresses. Second, the interrupt remapping is circumvented as NMIs are considered to be exceptions by the architecture, so no source information is added during their delivery. Without source-id the interrupt remapping Engine is not able to validate the interrupt. In addition, the NMI was indirectly spawned by the Memory Controller Hub (and *not* by our passthrough device) which is handled by the host. Finally, x2APIC mode, which forbids to reenable compatibility format MSIs during runtime, is also circumvented.

NMI Injection Evaluation

We successfully verified our NMI injection attack on both Xen 4.2 and KVM 3.5.0 in the configurations shown in Table 1, however, every VMM, which runs on a platform with System Error Reporting enabled, can be affected. In order to be sure that the attack overcomes all the available protection mechanisms, we enabled DMA and interrupt remapping as well as x2APIC mode on the privileged VM/host (this configuration was known to be safe against all known interrupt attacks).

Our physical NMI can have three different scenarios with respect to its impact depending on the configuration of the privileged guest/host/VMM kernel. First, we *simulate* a legitimate hardware error induced purely by software from the guest VM which is reported to the privileged VM/Host OS/VMM. As a result, the system administrator believes that the MCH has some serious hardware problems and the motherboard must be replaced as soon as possible. This fact on its own leads to an indirect Denial of Service attack against the host. Second, depending on the privileged guest/host OS/VMM kernel configuration, the system can halt as it cannot recover from NMIs that were signalled as a result of a PCI System Error (`SERR#`). Note that we could reproduce this case as well, which means a direct Denial of Service attack against the host. Finally, if the host kernel does not halt, the attacker has still chance to execute arbitrary code on the host by means of this host-side NMI.

To achieve this, we have to take into consideration similar interrupt attacks that were used in the past to execute arbitrary code [9] by exploiting a race condition. A similar race condition could be used in our case as well:

1. Prepare a shellcode that is made of four parts: a) The code to execute in the VMM, b) reference to a swapped page, c) padding, d) pointer to the code to execute

2. The attacker needs to count the number of pages between the location of the page fault handler entry in the Interrupt Descriptor Table (IDT) and that of the VMM stack to set the length of padding (c) in the shellcode. This padding is used to span that distance and overwrite the page fault entry in the IDT with a pointer (d), that points to the code (a) to be executed in the VMM.

3. Place the shellcode in the MMIO address space of the guest VM that the attacker controls. As the copy operation from the MMIO space is slow enough (MB/s) it can be interrupted with high probability.

4. As hypercalls contain code snippets that copy memory from guest VM space into that of the VMM, call legitimate hypercalls in a loop to catch the very moment when the corresponding function (i.e., *copy_from_user()*) copies the guest buffer with the shellcode.

5. Create a host-side NMI from the guest via a malformed MSI request to interrupt the hypercalls.

6. Modify a register value (e.g., rax) inside the interrupted copy operation in the hypercall by the NMI handler (race condition) to control the length of copy operation. For example, the NMI handler can return with a large value (e.g., error code) in the rax register which register indirectly influences the length of copy operation (e.g., $mov\ rcx, rax$).

7. While copying the shellcode into the VMM space the hypercall handler will end up in a page fault, as we placed a reference to a swapped page (b) into our shellcode. However, the page fault handler entry had already been overwritten, so the injected code (a) is launched.

This exploit sequence is difficult to apply in our case because hardware interrupt handlers (e.g., NMI) do not modify saved register states (see point 6 in the list above). Thus, they cannot influence the behavior of the interrupted handler (e.g., hypercall). However, the NMI injection attack is pervasive and works on all Intel platforms which enable System Error Reporting, which is a typical configuration for server platforms (e.g., IaaS clouds).

Mitigation

As we discussed above, there is no publicly known hardware protection mechanism available against our NMI injection attack. We responsibly reported the problems and received a Xen Security Advisory (XSA-59) and a CVE number (CVE-2013-3495) from Xen and MITRE, respectively. In addition, after a long discussion period with Xen and Intel, we concluded that there is *no ideal* solution available against our attack. Considering mitigations, SERR reporting can either be disabled on the Memory Controller Hub, or system software can block SERR error signaling due to Unsupported Request error resulting from malformed MSI requests. The former advice is quite intrusive as it suppresses all the system errors coming from the MCH, which affects legitimate hardware errors as well. At the same time, this is supported by all the chipsets. The second option is a more fine-grained solution, however, according to recent debates [37] between Xen and Intel, it seems that the problem cannot be put to rest as software patches are required every time a new chipset/processor is released. In addition, these patches disable SERR which may affect legitimate requests. In summary, both of the above mitigations can be a daunting and very expensive operation especially for cloud operators who expose VM instances for public use with passthrough devices (e.g., Amazon EC2).

6.3 Don't Care Field Fuzzing

In our second test, we used PTFuzz to modify the *don't care* field of remappable format MSI address. *don't care* fields should never influence the behavior of the hardware if manipulated. Interestingly, this is not the case for remappable format MSIs. In our experiments, we combined a remappable format MSI *data* with a corresponding MSI *address* component in a way to create an Interrupt Remapping Table *index* larger than the number of entries in that table. When the fuzzer modified the *don't care* field against a fully protected KVM guest, we observed that three different types of interrupt faults were generated on the host OS/VMM (Figure 4).

```
INTR-REMAP: Request device [00:19.0] fault index 77ff
INTR-REMAP:[fault reason 32] Detected reserved fields in
the decoded interrupt-remapped request
INTR-REMAP: Request device [00:19.0] fault index 8fff
INTR-REMAP:[fault reason 32] Detected reserved fields in
the decoded interrupt-remapped request
```

DC=0

```
INTR-REMAP: Request device [00:19.0] fault index ffff
INTR-REMAP:[fault reason 32] Detected reserved fields in
the decoded interrupt-remapped request
```

DC=1, DC=2

```
INTR-REMAP: Request device [00:19.0] fault index ffff
INTR-REMAP:[fault reason 34] Present field in the IRTE
entry is clear
```

DC=3

Figure 4: Raising different types of interrupt faults on the KVM host by fuzzing the *don't care* (DC) field of a remappable format MSI address. Note that both the fault reason and the fault index values (Interrupt Remapping Table Entries - IRTE) are changing on different DC values.

Interrupt Remapping Fault Evaluation

This case is very similar to the NMI injection attack in the sense that the attacker can spawn a partially controllable host-side hardware fault by fabricating unexpected conditions. We do highlight here that this case also opens a door towards practical guest-to-VMM escapes. Theoretical exploitation scheme can either be a similar race condition presented in Section 6.2 or a buffer overflow by influencing the fault reason or the fault index values in the hardware fault handler. Until now, we could not identify a problem in VMM software that allows for practical exploitation. At the same time, hardware problems are orthogonal to VMM implementation bugs, thus, it is enough to find a single problem in any VMM implementation, and the attacker can succeed.

All the above problems demonstrate the hardware does not always follow the specifications from vendors or the synergy is missing between hardware and software for flawless collaboration.

7. RELATED WORK

To the best of our knowledge, we are the first who systematically discuss and implement a wide range of attacks that exploit the device passthrough technology. However, a considerable amount of related work exists in the three attack areas that we cover in this paper. More information about possible attacks in hardware virtualization can be read in [38].

Attacks via the PCI/PCIe configuration space.

Zhou et al. [30] presented several attacks (e.g., MMIO overlapping) via the PCI/PCIe configuration space. However, the attacks were presented in a different context, and the authors' focus was more on the development of a small hypervisor to prevent these device passthrough attacks from causing damage in a compromised OS. On the contrary, we aim at revealing design, configuration, and implementation weaknesses in commodity VMMs that can be abused to escalate privileges, read restricted memory regions and perform DoS attacks.

A privilege escalation attack via PIO based PCI/PCIe configuration space accesses was discussed for USB UHCI [39] controllers by Duflot et al. [40, 31]. The authors were able to manipulate the secure level of OpenBSD and escalate root access in privileged domains (e.g., Xen Dom0) from arbitrary guests by evading Intel VT-x protection. Since Xen does not allow direct configuration space access any more [33], our PIO attacks were confined inside the attacker guest. We faced the same issues with KVM as well.

DMA attacks. The introduction of DMA opened the door to a new set of attacks. By misusing DMA transactions either a native or a virtualized system can be compromised by reading/writing arbitrary physical memory locations. A long list of DMA attacks exists in the literature, based on USB *On-The-Go* controllers [41], PCI cards [42, 43], or FireWire controllers [44, 45, 46, 47, 48]. Even though, most of these attacks were presented for native environments, proof-of-concept codes for virtual environments exist as well [10]. The main difference with our paper is that *none* of the previous works examined the impact and limitations introduced by existing DMA protection mechanisms (e.g., DMA remapping).

Interestingly, a DMA attack can also be launched by a code residing on the northbridge [49, 14]. From one aspect, this higher privileged code is advantageous for an attacker. However, malicious DMA transactions are still ineffective on systems with DMA remapping enabled (e.g., Intel VT-d is turned on).

Interrupt Attacks. Wojtczuk et al. [9] demonstrated that the x86-64 architecture is vulnerable against Message Signalled Interrupt (MSI) based attacks. Later, Müller et al. [50] pointed out a similar interrupt attack on PowerPC. However, both of the attacks worked only without an active interrupt remapping engine. The authors showed that an attacker can force the generation of unmediated MSIs in these cases, allowing her to send specific interrupts to the physical CPU. This attack even allowed Wojtczuk et al. [9] to execute arbitrary code with VMM privileges in a given Xen installation. In contrast to that, our host-side NMI interrupt cannot be blocked with currently available protection mechanisms (e.g., interrupt remapping engine) without limiting platform functionalities. Also, we can launch our attack from fully-virtualized (HVM) guests that are known to be more isolated than their paravirtualized counterparts (e.g., because of the hardware-supported protection ring for the VMM code).

Protection Evasion. While we focus on *circumventing* all the hardware protection mechanisms currently available, other approaches try to disable them. For example, an attacker can modify hardware-level data structures (e.g., Interrupt Descriptor Table) or configuration tables (e.g., DMA remapping table) [51] to turn off IOMMU. Another approach aims at making the illusion of a non-parsable DMA remapping table. To achieve this, an attacker has to set a zero length for such a table during boot time. Another class of attacks aim at modifying the metadata exposed by I/O controllers to mislead the IOMMU. One such an attack is described by Lone-Sang et al. [13]. Here, the attackers could map two I/O devices into the same physical memory range by impersonating a PCI Express device with a legacy PCI device. This attack, however, requires physical access to the victim machine.

We partially discuss a similar approach in Section 3, where we presented a problem to circumvent the interrupt remapping engine on KVM by updating a BIOS on a motherboard that originally does not support this technology. In this way, an attacker could use a passthrough device without the appropriate protections enabled.

Recently, an interrupt remapping source validation flaw was reported by the Xen security team (CVE-2013-1952). More precisely, MSI interrupts from bridge devices do not provide source information, so the interrupt remapping table cannot check the orig-

inator bridge. In this way, a VM that owns a bridge can inject arbitrary interrupts to the system via MSI interrupts.

8. CONCLUSIONS

In this paper, we presented and tested a wide range of pass-through attacks on commodity VMMs. Some of them were already publicly known while others were presented for the first time in this paper. To discover new vulnerabilities, we designed and implemented an automatic fuzzer called PTFuzz. This tool successfully detected various unexpected hardware behaviors while running on commodity VMMs.

Our experiments showed that software patches (e.g., when the device configuration space is emulated) and robust hardware protections can indeed prevent all previously discovered attacks. Nonetheless, we demonstrated that the proper configuration of these protection mechanisms can be a daunting task. Unfortunately, VMMs remain vulnerable to sophisticated attacks. In this paper, we discovered and implemented an interrupt attack that leverages unexpected hardware behaviour to circumvent all the existing protection mechanisms in commodity VMMs. To the best of our knowledge, this is the first attack that exhibits such a behaviour and to date it seems that there is no easy way to prevent it on Intel platforms.

The fact that we discovered a major vulnerability as well as an unexpected hardware behaviour in Intel platforms does not necessarily mean that VMMs are threatened in the wild, but certainly raises an alarm to cloud operators. We believe that our study can help them to better understand the limitations of their current architectures to provide secure hardware virtualization and to prepare for future attacks.

Acknowledgment

The research leading to these results was partially funded by the European Union Seventh Framework Programme (contract N 257007) and by the French National Research Agency through the MIDAS project. We would also like to thank anonymous reviewers for their valuable comments, Mariano Graziano for running certain experiments with DMA attacks as well as Pipacs from PaX team, Olivér Pintér and Hunger for their technical advices.

Special thank goes to Rafal Wojtczuk <rafal@bromium.com>, Jan Beulich from Xen Security and Intel PSIRT for technical discussions and feedback on the interrupt attack. Additional thanks to Csaba Krasznay from HP Hungary for providing extra machines to thoroughly test the interupt attack.

9. REFERENCES

[1] Amazon EC2. `http:/aws.amazon.com/`.

[2] Google Compute Engine. `https://cloud.google.com/products/compute-engine/`.

[3] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 335–350, New York, NY, USA, 2007. ACM.

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

[5] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*, pages 143–158, 2010.

[6] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: bare-metal performance for I/O virtualization. *SIGARCH Comput. Archit. News*, 40(1):411–422, March 2012.

[7] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 38–49, New York, NY, USA, 2010. ACM.

[8] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 350–361, New York, NY, USA, 2010. ACM.

[9] Rafal Wojtczuk and Joanna Rutkowska. Following the White Rabbit: Software attacks against Intel®VT-d technology, April 2011.

[10] Rafal Wojtczuk. Subverting the Xen Hypervisor - Xen 0wning Trilogy part I. *Black Hat USA*, aug 2008.

[11] Rafal Wojtczuk. Preventing and Detecting Xen Hypervisor Subversions - Xen 0wning Trilogy part II. *Black Hat USA*, aug 2008.

[12] Joanna Rutkowska and Alexander Tereshkin. Bluepilling the Xen Hypervisor - Xen 0wning Trilogy part III. *Black Hat USA*, aug 2008.

[13] Fernand Lone Sang, Eric Lacombe, Vincent Nicomette, and Yves Deswarte. Exploiting an I/OMMU vulnerability. *MALWARE*, 2010.

[14] Patrick Stewin and Iurii Bystrov. Understanding DMA Malware. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2012.

[15] AMD. AMD 64 Architecture Programmer's Manual: Volume 2: System Programming. *AMD Pub. no. 24593 rev. 3.20*, 2011.

[16] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual - Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C, Aug 2012.

[17] Jiuxing Liu. Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support. In *IEEE Int'l Parallel & Distributed Processing Symp. (IPDPS)*, pages 1–12, April 2010.

[18] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. High performance VMM-bypass I/O in virtual machines. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 3–3, Berkeley, CA, USA, 2006. USENIX Association.

[19] Himanshu Raj and Karsten Schwan. High performance and scalable I/O virtualization via self-virtualized devices. In *Proceedings of the 16th international symposium on High performance distributed computing*, HPDC '07, pages 179–188, New York, NY, USA, 2007. ACM.

[20] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A.L. Cox, and W. Zwaenepoel. Concurrent direct network access for virtual machine monitors. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 306 –317, feb. 2007.

[21] Ben-Ami Yassour, Muli Ben-Yehuda, and Orit Wasserman. Direct device assignment for untrusted fully-virtualized virtual machines. Technical report, 2008.

[22] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, April 2006.

[23] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure resilience for device drivers. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, pages 41–50, Washington, DC, USA, 2007. IEEE Computer Society.

[24] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[25] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schneider. Device driver safety through a reference validation mechanism. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 241–254, Berkeley, CA, USA, 2008. USENIX Association.

[26] Intel. Intel® Virtualization Technology for Directed I/O. *Intel® Technology Journal*, 10, August 2006.

[27] Intel. Intel® Virtualization Technology for Directed I/O, Feb 2011.

[28] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[29] Intel. Intel 6 Series Chipset and Intel C200 Series Chipset. May 2011.

[30] Zongwei Zhou, Virgil D. Gligor, James Newsome, and Jonathan M. McCune. Building verifiable trusted path on commodity x86 computers. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 616–630, Washington, DC, USA, 2012. IEEE Computer Society.

[31] Loïc Duflot and Laurent Absil. Programmed I/O accesses: a threat to Virtual Machine Monitors? *PacSec*, 2007.

[32] Intel. Intel Graphics Virtualization on KVM. *KVM Forum*, Aug 2011.

[33] Emulation of PCI configuration space access. `http://xenbits.xen.org/hg/xen-4.2-testing.hg/rev/1ac2a314aa3c`.

[34] Joanna Rutkowska. pciback: question about the permissive flag. XEN-devel mailing list, Available at `http://old-list-archives.xenproject.org/archives/html/xen-devel/2010-07/msg00257.html`, July 2010.

[35] Sam Fleming. Accessing PCI Express* Configuration Registers Using Intel Chipsets. December 2008.

[36] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. vIOMMU: efficient IOMMU emulation. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, pages 6–6, Berkeley, CA, USA, 2011. USENIX Association.

[37] XSA-59 blog. `http://www.gossamer-threads.com/lists/xen/devel/318464?page=last`.

[38] Gábor Pék, Levente Buttyán, and Boldizsár Bencsáth. A survey of security issues in hardware virtualization. *ACM Comput. Surv.*, 45(3):40:1–40:34, July 2013.

[39] Intel. Universal Host Controller Interface (UHCI), Mar 1996.

[40] Loïc Duflot. Contribution à la sécurité des systèmes d'exploitation et des microprocesseurs. In *PhD thesis, Universite de Paris*, Oct 2007.

[41] David Maynor. 0wn3d by everything else - USB/PCMCIA Issues. *CanSecWest/core05*, May 2005.

[42] Brian D. Carrier and Joe Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 2004.

[43] Christophe Devine and Guillaume Vissian. Compromission physique par le bus PCI. In *Proceedings of the 7th Symposium sur la Sécurité des Technologies de l'Information et des Communications*, SSITC 2009, pages 169–193, June 2009.

[44] Damien Aumaitre. Voyage au coeur de la mémoire. In *Proceedings of the 6th Symposium sur la Sécurité des Technologies de l'Information et des Communications*, SSITC 2008, pages 378–437, June 2008.

[45] Adam Boileau. Hit by a Bus: Physical Access Attacks with Firewire. *Ruxcon*, 2006.

[46] Maximillian Dornseif. 0wned by an iPod. *PacSec*, 2004.

[47] Michael Becher, Maximillian Dornseif, and Christian N. Klein. FireWire - all your memory are belong to us. *CanSecWest/core05*, May 2005.

[48] Antonio Martinm. FireWire memory dump of a windows XP computer: a forensic approach. Technical report, 2007.

[49] Alexander Tereshkin and Rafal Wojtczuk. Introducing Ring -3 Rootkits. *Black Hat USA*, 2009.

[50] Kevin Müller, Daniel Münch, Ole Isfort, Michael Paulitsch, and Georg Sigl. Decreasing system availability on an avionic multicore processor using directly assigned pci express devices. In *EUROSEC 2013*, Apr 2013. Prag.

[51] Rafal Wojtczuk, Joanna Rutkowska, and Alexander Tereshkin. Another Way to Circumvent Intel Trusted Execution Technology, December 2009.