Requirements for an Integrity-Protected Hypervisor on the x86 Hardware Virtualized Architecture

Amit Vasudevan[1], Jonathan M. McCune[1], Ning Qu[2], Leendert van Doorn[3], and Adrian Perrig[1]

[1] CyLab, Carnegie Mellon University, Pittsburgh, PA, USA
[2] Nvidia Corp., Santa Clara, CA, USA
[3] Advanced Micro Devices (AMD) Corp., Austin, TX, USA

Abstract. Virtualization has been purported to be a panacea for many security problems. We analyze the feasibility of constructing an integrity-protected hypervisor on contemporary x86 hardware that includes virtualization support, observing that without the fundamental property of hypervisor integrity, no secrecy properties can be achieved. Unfortunately, we find that significant issues remain for constructing an integrity-protected hypervisor on such hardware. Based on our analysis, we describe a set of *necessary* rules that must be followed by hypervisor developers and users to maintain hypervisor integrity. No current hypervisor we are aware of adheres to all the rules. No current x86 hardware platform we are aware of even allows for the construction of an integrity-protected hypervisor. We provide a perspective on secure virtualization and outline a research agenda for achieving truly secure hypervisors.

1 Introduction

Virtualization allows a single physical computer to share its resources among multiple *guests*, each of which perceives itself as having total control of its *virtual machine* (VM) [30]. Virtualization is an effective means to improve hardware utilization, reduce power and cooling costs, and streamline backup, recovery, and data center management. It is even making inroads on the client-side. However, in all of these roles, the *hypervisor* (or *Virtual Machine Monitor (VMM)*) becomes yet another maximally privileged software component from the perspective of the guest's trusted computing base (TCB). This stands in direct violation of several well-known principles of protecting information in computer systems [36]. In many scenarios, the hypervisor may support guests for two or more mutually distrusting entities, thereby putting to the test the hypervisor's ability to truly protect its own integrity and isolate guests [7].

Unfortunately, today's popular hypervisors are not without their share of vulnerabilities (e.g., [4, 49]), and appear to be unsuitable for use with highly sensitive applications. Despite recent enhancements to hardware support for virtualization [6, 21, 32], low-level systems problems (e.g., System Management Mode exploits [12, 50] and vulnerable BIOSes [26, 35]) continue to plague existing solutions. We distinguish between threats to *hypervisor integrity* and threats to hypervisor and guest data secrecy, observing that an integrity-protected hypervisor is a necessary, but not sufficient, condition for maintaining data secrecy in the face of mutually distrusting guests. We define *integrity-protected* to mean that the hypervisor's code cannot be modified in any fashion and the hypervisor's data cannot be maliciously changed. The secrecy of guests' data is explicitly defined to be outside the scope of the current paper.

Are today's virtualization and security extensions to the x86 platform sufficient to maintain the integrity of a hypervisor? This is a challenging question to answer for current platforms due to their high complexity. Challenging practical issues that we consider include per-device idiosyncrasies that arise from devices that are not completely standards-compliant, and the need to offer the precise (i.e., bug-compatible) environment expected by unmodified guest operating systems.

Given the challenges in designing and implementing an integrity-protected hypervisor, we define threats to data secrecy and availability (such as covert channels, side channels, timing channels, and resource exhaustion attacks) to be outside the scope of this paper. Data secrecy and availability can be ensured only if the fundamental property of hypervisor integrity is realized. For example, without integrity-protection, portions of the hypervisor that manage the isolation of memory pages between guests may be maliciously modified, thereby allowing one guest to make modifications to the code or data of another guest. These modifications may include releasing secrets.

We enumerate core system elements (e.g., buses and system components) required to protect the integrity of the hypervisor in §2, and present rules for an integrity-protected hypervisor in §3. In §4, we discuss specific details of AMD's and Intel's hardware virtualization support in the context of an integrity-protected hypervisor. We believe our rules represent a strong first approximation of the necessary requirements for an integrity-protected hypervisor on today's x86 hardware. We write these rules as hypervisor developers with years of experience investigating hypervisor integrity. We leave for future work the demonstration that these rules are also sufficient.

Given the complexity of the current x86 hardware virtualization architecture and the plethora of available devices, it may be difficult to conclude definitively that an integrity-protected hypervisor can be created when its VMs are

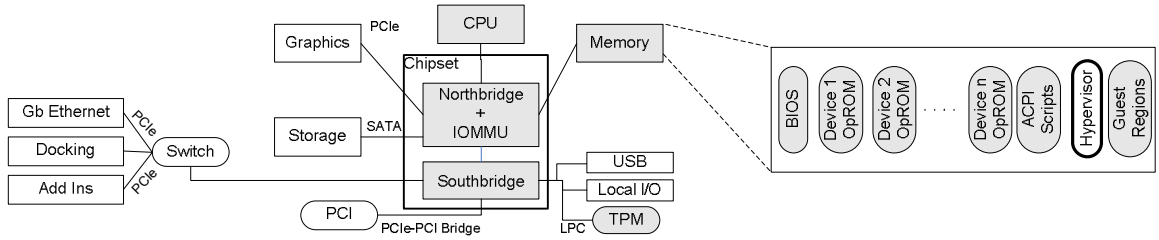

Fig. 1: Elements of today's x86 hardware virtualization architecture. Shaded regions represent elements that *must* be access-controlled to ensure hypervisor integrity. We discuss the TPM in §3.1.

commodity operating systems expecting a rich set of peripheral devices and capable of running arbitrary code. Thus, in §5, we also describe a spectrum of different VM environments, illustrating why it may be non-trivial to ensure that some combinations of OSes can execute unmodified on an integrity-protected hypervisor. Further, in §6 we describe the design of several popular hypervisors and discuss the adherence of their design to our integrity-protected hypervisor rules. This section also includes a table summarizing our rules and the level of adherence of these popular hypervisors.

To summarize, this paper makes the following contributions: (1) an analysis of the platform elements that must be properly managed to integrity-protect a hypervisor, (2) a set of rules that hypervisor developers must follow, (3) the manifestation of these rules on contemporary systems, (4) the implications of these rules with respect to supporting concurrent VMs on top of the hypervisor, and (5) an analysis of the adherence of existing hypervisor designs to these rules. While we believe our results are interesting in their own right, we also intend this paper to serve as a call to action. Significant additional research is needed to determine whether the rules presented here are not just necessary but also sufficient. We also hope to inspire subsequent investigations to ascertain the data secrecy properties attainable with commodity virtualization solutions, particularly given the effectiveness of recently disclosed attacks (e.g., [31]).

2 Elements of x86 Hardware Virtualization

Our goal is to preserve the integrity of a hypervisor, i.e., preventing inadvertent or malicious manipulation of hypervisor memory regions (both code and data). Consequently, only system components that can directly access memory or mediate memory accesses become critical to preserving hypervisor integrity. AMD and Intel—the two major x86 CPU vendors that support hardware virtualization—design their CPUs to work with the Peripheral Component Interconnect Express (PCIe [10]) system architecture and build on the existing non-hardware virtualized x86 architecture. Both the PCIe and x86 architectures define precise methods for moving data to and from system memory. This standardization serves as a guide for the remainder of this section. To maintain generality, where applicable, we adhere to the PCIe terminology instead of using CPU/vendor specific terms.

2.1 Overview

Current x86 hardware virtualization encompasses both hardware and software elements (Figure 1). The hardware elements are connected via the PCIe bus (though both Intel and AMD employ proprietary buses – Quick Path Interconnect and Hypertransport, respectively – to connect the northbridge and southbridge).

The Northbridge (or Memory Controller Hub – MCH – on recent Intel-VT platforms) connects the CPU(s) (which include an integrated Memory Management Unit, or MMU) and Memory/Cache. The northbridge also supports other performance-critical components such as the graphics controller. Recent Intel-VT platforms use the term Graphics and Memory Controller Hub (GMCH) to describe a MCH with integrated graphics controller. The northbridge further contains an IO Memory Management Unit, or IOMMU, which is responsible for managing Direct Memory Access (DMA) transactions between the Memory and all attached peripherals without intervention by the CPU.

The Southbridge (or IO Controller Hub – ICH) supports less performance-critical IO capabilities such as some types of disk and network interfaces, USB, audio, and legacy ports such as the serial and parallel port. The southbridge also connects an optional Trusted Platform Module (TPM) which is used to measure a dynamic root of trust (which we treat in §2.2 and §3.1).

The software elements in the x86 hardware-virtualized architecture are the system boot and runtime firmware (BIOS), firmware code on various peripherals, power management scripts within the BIOS, memory regions belonging to individual VMs, and the hypervisor itself.

Throughout this paper we use the terms northbridge, southbridge and IOMMU functionally rather than referring to physical components. As an example, on the recent AMD-V and Intel-VT architectures the northbridge and IOMMU are physically a part of the CPU. However, their functionality remains the same. The following sections discuss these hardware and software elements in the context of an integrity-protected hypervisor.

2.2 Hardware Elements

The hardware elements in the context of preserving hypervisor integrity are the CPU, Northbridge and Southbridge.

CPU An x86 hardware virtualization-capable CPU, like a normal x86 CPU, includes registers, caches, and an instruction set. The CPU has two over-arching operating modes: host (more privileged) and guest (less privileged). The guest mode is used to execute a guest OS environment in a controlled manner, i.e., in a virtual machine. The host mode can *intercept* certain critical operations that are performed in guest mode such as accessing CPU control registers and performing IO. There can be multiple concurrent guest instantiations, but only one host mode execution environment. Both host and guest modes can further execute in any of four privilege *rings* 0 (most privileged) through 3.

System Management Mode (SMM) SMM code (part of the BIOS) executes at the highest privilege level and is used to handle system events such as system and CPU temperature control, and legacy support for USB input devices. SMM is entered whenever a System Management Interrupt (#SMI) occurs. The #SMI is an external hardware interrupt and can occur at any point during system operation. When SMM is entered, all normal execution state is suspended (including host and guest modes) and firmware code (#SMI handler) is executed with full access to system physical memory. The #SMI handlers are stored in a special memory region called the System Management RAM (SMRAM) which is only accessible by programming certain CPU registers or IO locations within the southbridge (§2.2).

Memory Management Unit (MMU) The MMU is the CPU component that enables virtual memory management and handles all memory accesses from the CPU. Its main function is to translate virtual addresses to physical addresses using paging structures while enforcing memory protections, in both the host and guest modes. Recent x86 hardware-virtualized CPUs introduce the concept of hardware physical memory virtualization where memory addresses are separated into guest virtual, guest physical, and system physical. The guest virtual addresses are translated to guest physical addresses using guest paging structures. The guest physical addresses are translated into system physical addresses using another set of paging structures within the hypervisor.

Microcode CPU microcode resides in a special high-speed memory within the CPU and translates instructions into sequences of detailed circuit-level operations. Essentially, microcode enables the CPU to reconfigure parts of its own hardware to implement functionality and/or fix bugs in the silicon that would historically require procuring a new unit. Microcode updates are loaded by the BIOS or the OS into the CPU dynamically.

All CPU(s) are shared between the hypervisor and the VM(s) that it runs, as the portion of the hypervisor that handles guest intercepts will always execute on the same CPU as the VM that generated the intercept. This can lead to hypervisor integrity compromise if not managed properly. As an example, a malicious VM may attempt to manipulate CPU cache contents so that unintended code runs as if it is hypervisor code (e.g., [50]). An attacker may also change existing SMI handlers in BIOS so that the malicious handlers execute as SMM code with sufficient privileges to modify hypervisor physical memory regions [12, 50]. An attacker can also alter a legitimate microcode[4] update to execute a CPU instruction that would normally be illegal and instead "trick" the memory caches into thinking the CPU is in host mode [37]. From there, the attacker can gain access to hypervisor memory regions.

Northbridge A *northbridge* (aka memory controller hub, MCH, or memory bridge) typically handles communication between the CPU, memory, graphics controller, and the southbridge (§2.2). The northbridge handles all transactions to and from memory. The northbridge also contains an IO Memory Management Unit (IOMMU) that is responsible for managing direct device accesses to memory via DMA.

IOMMU An IO Memory Management Unit (IOMMU) [5] manages Direct Memory Accesses (DMA) from system devices. It allows each device in the system to be assigned to a specific *protection domain* which describes the memory regions that are accessible by the device. When a device attempts to access system memory, the IOMMU intercepts the access and determines whether the access is to be permitted as well as the actual location in system memory that is to be accessed. In systems with multiple physical CPUs, there may be multiple IOMMUs, but logical CPUs on a single die currently share an IOMMU.

Most devices today perform DMA to access memory without involving the CPU. DMA increases system performance since the CPU is free to perform computations, but a malicious device may attempt DMA to hypervisor memory regions, potentially compromising its integrity. As an example, Firewire is a serial bus that allows endpoints to issue remote DMA requests. One system may be able to issue DMA requests on the other system via the Firewire controller, thereby gaining read/write access to the full memory contents of the target and compromising its integrity [8].

[4] Intel digitally signs microcode updates and hence altering a legitimate microcode update is not straightforward. However, AMD microcode updates are not signed, thereby allowing an attacker to freely modify bits [37].

[5] x86 CPUs also include a more limited graphics-related address translation facility on-chip, called a GART. However, unlike the IOMMU, the GART is limited to performing address translation only and does not implement protections.

Southbridge The *southbridge* (also known as the IO Bridge) is a chip that implements the lower-bandwidth IO in a system, e.g., USB, hard disks, serial ports, and TPM. It is also responsible for providing access to the non-volatile BIOS memory used to store system configuration data. The southbridge contains certain IO locations that may be used to compromise hypervisor integrity. For example, SMRAM access and SMI generation are controlled by IO locations that reside within the southbridge. An attacker could implant a malicious SMI handler by enabling SMRAM access [12] and execute the handler by generating a SMI. The malicious SMI handler then has unrestricted access to hypervisor memory regions. Similarly, system configuration data copied from firmware into system memory at boot time can be manipulated using the southbridge, potentially preventing the BIOS from setting the system to a known correct state during boot-up.

2.3 Software Elements

In addition to the hardware elements that comprise the current x86 hardware virtualization architecture, there are various software elements. The software elements include firmware such as the BIOS, option ROMs, power management scripts that are embedded into the platform hardware, and OS and applications that run within a VM on top of the hypervisor. These software elements can contain bugs or can be altered to compromise the integrity of a hypervisor. Further, certain software elements such as the BIOS and option ROMs execute even before a hypervisor is initialized and can set the system into a malicious initial state that compromises hypervisor integrity.

BIOS / UEFI The Basic Input and Output System (BIOS) is by far the most prevalent firmware interface for x86 platforms. The BIOS prepares the machine to execute software beginning from a known state – a process commonly known as system bootstrapping. The Universal Extensible Firmware Interface (UEFI) is a specification that defines a software interface between an operating system and platform firmware [23]. UEFI is a much larger, more complex, OS-like replacement for the older BIOS firmware interface but is only recently making its way into commodity platforms.

The BIOS is typically stored on a Flash (EEPROM) chip that can be programmatically updated. This allows for BIOS vendors to deliver BIOS upgrades that take advantage of newer versions of hardware or to correct bugs in previous revisions. Unfortunately, this also means that a legitimate BIOS can be overwritten with a malicious one that may compromise hypervisor integrity, e.g., hardware virtualization rootkits such as BluePill [34] that emulate nested hypervisor functionality. Thus, an integrity protected hypervisor thinks it is executing at the lowest level; BluePill code however has complete control over hypervisor memory regions.

Note that certain bootstrapping firmware such as Intel's EFI [23] and Phoenix's SecureCore BIOS [41] only allow signed updates to the relevant Flash chip. However, since they have to include OEM customizable sections, parts of the BIOS image are not signature verified. Such areas (e.g., the BIOS boot logo) have been successfully changed by attackers to run malicious code [18].

Option ROMs A system can contain several BIOS firmware chips. While the primary BIOS typically contains code to access fundamental hardware components , other devices such as SCSI storage controllers, RAID devices, network interface cards, and video controllers often include their own BIOS, complementing or replacing the primary BIOS code for the given component. These additional BIOS firmware modules are collectively known as *Option ROMs*, though today they are rarely implemented as read-only, instead using Flash to support updates.

The BIOS invokes option ROM code for all system devices during bootstrapping. This gives the option ROMs the chance to intercept system interrupts and occupy system memory, in order to provide increased functionality to the system at runtime. The option ROM code is often legacy code that accesses physical memory directly. An attacker may replace a legitimate option ROM with a malicious one which may then be invoked at runtime by an OS running within a VM [16]. This code can then have unrestricted access to hypervisor physical memory regions, thereby compromising its integrity. Certain BIOS code (e.g., Intel Active Management Technology) execute on a seperate processor in parallel to the main CPU and can be used to compromise hypervisor integrity via DMA.

Power Management Scripts Most systems today are equipped with power management capabilities where the entire system, including devices, can be transitioned into a low-power state to conserve energy when idle. Power management on current commodity systems is governed by the Advanced Configuration and Power Interface (ACPI) specification [20]. With an ACPI-compatible OS, applications and device drivers interact with the OS kernel, which in turn interacts with the low-level ACPI subsystem within the BIOS.

An ACPI subsystem provides an OS with certain power management data structures in memory. A Differentiated System Descriptor Table (DSDT) provides power management code for system devices in a bytecode format called the ACPI Machine Language (AML). The OS kernel typically parses and executes the DSDT scripts to set device and CPU power states. Popular OSes such as Windows parse AML scripts in a CPU mode that allows accessing physical memory directly. An attacker that can insert malicious code within AML scripts will then have unrestricted access to physical memory when executed [17].

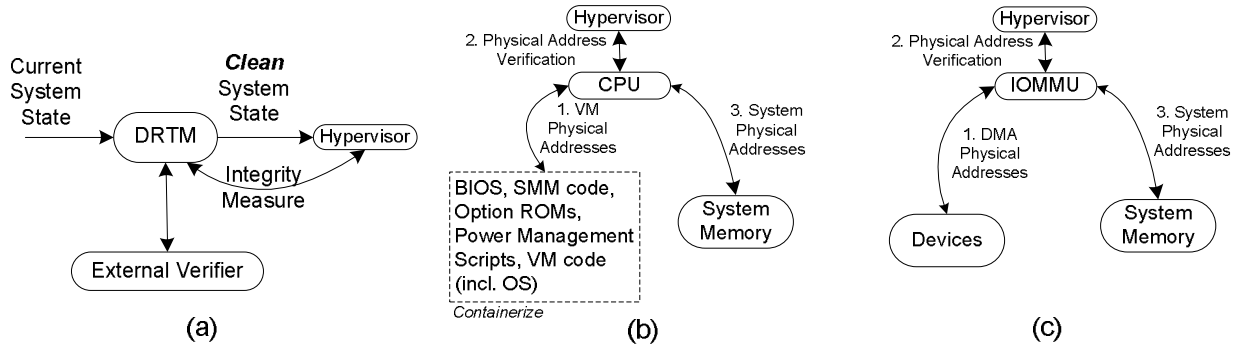

Fig. 2: An integrity-protected hypervisor: (a) must use a dynamic root of trust to startup, (b) must protect itself against any code via physical memory virtualization, and (c) must prevent any device in the system from directly accessing hypervisor memory. Finally, the hypervisor itself must be free of vulnerabilities.

Other Code A VM running on a hypervisor can run a full commodity OS. The OS itself may be subverted and may attempt to attack the hypervisor. As an example, malicious code within a VM may attempt to manipulate the caching policies of hypervisor memory, thereby effectively gaining access to hypervisor memory regions.

3 Integrity-Protected Hypervisor

We now present our assumptions and the rules an integrity-protected hypervisor must observe, with related discussion. For the hypervisor to protect its integrity, it must ensure that it starts up in an unmodified fashion and continues to run without any inadvertent modifications to its code and data.

We assume that: *The target system on which an integrity-protected hypervisor runs is physically protected.* An attacker who can physically tamper with or replace with malicious versions system components such as the northbridge, southbridge, CPU, or TPM may successfully compromise hypervisor integrity. As an example, commands generated by the CPU during *dynamic root of trust* establishment (see §3.1) must reach the TPM with their integrity intact. The TPM connects via the Low Pin Count (LPC) bus to the southbridge. The LPC bus is a relatively inexpensive and low-speed bus in modern systems, and is thus susceptible to physical tampering. Therefore, the platform on which an integrity-protected hypervisor runs must be physically protected at all times.

The rules for an integrity-protected hypervisor can be divided into rules that must be followed for (a) startup, (b) runtime, and (c) hypervisor design (Figure 2). From §2 we note that there are only two ways in which hypervisor memory regions can be accessed on an x86 hardware-virtualized platform: (a) via code executing on the CPU,[6] and (b) via system devices performing DMA operations. Accordingly, we present the *exact* rules that must be followed by an integrity-protected hypervisor during startup and runtime, and consider its design in the context of cases (a) and (b) above. Consequently, a hypervisor that follows these rules is automatically protected from integrity compromise.

3.1 Startup Rules

These rules allow for the requirement that an integrity-protected hypervisor must start up in an unmodified fashion.

Definition 1. *A* dynamic root of trust *(DRT) is an execution environment created through a disruptive event that synchronizes and reinitializes all CPUs in the system to a known good state. It also disables all interrupt sources, DMA, and debugging access to the new environment. An explicit design goal of a DRT mechanism is to prevent possibly malicious firmware from compromising the execution of a hypervisor.*

Rule 1 *An integrity-protected hypervisor* **must** *be initialized via the creation of a dynamic root of trust.*

Discussion: The traditional BIOS initialization and boot sequence is plagued by having existed for several decades. As such, modern security requirements and virtualization capabilities did not exist when it was first conceived. The result of this is that there may exist legacy code in a system's BIOS that should not be trusted, since it was never subjected to rigorous analysis for security issues.[7] Further, many devices have option ROMs (§2.3) that are invoked by the system's BIOS at boot time. Thus, a malicious option ROM may take control of a platform before the hypervisor can

[6] This includes code executing from the CPU caches. For example, an attacker could attempt to tamper with the execution of the hypervisor not by changing the memory of the hypervisor directly, but by changing the view of the hypervisor's code when it is executed on the CPU by tampering with the values stored in the CPU code caches.

[7] For a closed system where only known firmware is executed at boot-time, a dynamic root of trust may not be necessary. However, most (if not all) x86 systems do not fall under this category.

be initialized. The dynamic root of trust mechanism provides a means for integrity-protected hypervisor initialization without breaking compatibility with myriad legacy devices.

Rule 2 *A dynamic root of trust mechanism* **must** *allow for an external verifier to ascertain the identity (e.g., cryptographic hash) of the memory region (code and data) in the new execution environment.*

Discussion: In some cases the DRT establishment can be unsuccessful (e.g., not all CPUs were able to synchronize). Further, if the DRT is successful, it only guarantees that the environment that is initialized is clean. The code that executes within the new environment may be a hypervisor whose integrity is already compromised. Therefore, there must be a mechanism to securely communicate to an external verifier whether a DRT was successfully established, as well as a cryptographic hash of the code and data in the clean execution environment, so that the identity of the loaded hypervisor can be verified to be one known to enforce integrity protections. There are currently both hardware (TPM-based [44]) and software [38] based hardware mechanisms for DRT establishment. The dynamic root of trust mechanism available on today's systems from AMD and Intel also includes a facility to perform an *integrity measurement* of the new environment using the platform's TPM chip.

3.2 Runtime Rules

Once an integrity-protected hypervisor has started in an unmodified form, it must continue to run without any inadvertent modifications to its memory regions (code and data). The following are the set of rules that must be followed at runtime by a hypervisor to ensure its integrity protection.

Rule 3 *An integrity-protected hypervisor* **must** *employ physical memory virtualization to prevent any code executing within a VM from accessing hypervisor memory regions.*

Discussion: A VM running on top of an integrity-protected hypervisor can run any commodity OS and applications. Such an OS can use the BIOS, option ROM, and Power Management Script code during runtime. E.g., Windows uses PCI BIOS functions during startup and employs the video BIOS to control the video subsystem during runtime. Further, it parses and executes Power Management Scripts as a part of system power management.

Code running within a VM may manipulate the MMU's virtual memory data structures to map and access hypervisor memory regions. Further, it may disable virtual memory support and directly access system physical memory. Therefore, an integrity-protected hypervisor must verify any physical address originating from a VM before it reaches the memory controller. Consequently, an integrity-protected hypervisor must virtualize physical memory.

Definition 2. *We define a* hypervisor core *to be the part of a hypervisor that is responsible for initializing, creating, and terminating VMs and for handling any intercepts that occur during VM execution.*

Definition 3. *We define* critical system operations *as operations that can result in compromising hypervisor integrity, e.g., changing page tables mapping within a VM to map and access memory regions belonging to the hypervisor.*

Rule 4 *An integrity-protected hypervisor* **must** *execute its core in the highest privilege level so it can interpose on critical system operations.*

Discussion: An integrity-protected hypervisor is responsible for setting up guest environments, running them, and tearing them down. These operations require execution of privileged instructions and hence the hypervisor must be in an operating mode that allows the use of such instructions. Further, an integrity-protected hypervisor must be able to detect any attempts to modify its memory regions by any other code within the system (e.g., code within the guest environments or device firmware). Thus, an integrity-protected hypervisor must execute in a CPU mode that enables the hypervisor to intercept and handle critical system operations such as device IO and writing to CPU control registers. Other parts of the hypervisor can execute at lower privilege levels contingent on hypervisor design.

Definition 4. *Critical CPU registers are the set of CPU registers that can be used to compromise the integrity of a hypervisor. On the x86 hardware virtualized architecture they can be divided into: (i) control registers – used for controlling the general behavior of the CPU such as interrupt control, switching the addressing mode (16/32/64-bit), and floating-point/multimedia unit control, (ii) segment registers – used to define the memory region and access type for code, data and stack segments, (iii) debug registers – used for debugging purposes, and (iv) machine specific registers (MSR) – special-purpose control registers exposing CPU implementation-specific features. E.g., MSR_EFER is used on both Intel-VT and AMD-V CPUs to enable extended features such as NX (no-execute) memory protections.*

Rule 5 *An integrity-protected hypervisor* **must** *have an independent set of critical CPU registers and* **must** *sanitize values of CPU data registers during control transfers to and from VMs.*

Discussion: Sharing critical CPU registers between the hypervisor and a VM can lead to hypervisor integrity compromise. Code within a VM may use the control registers to turn off MMU-based virtual memory and set the data segment to address all of physical memory to gain access to hypervisor physical memory regions. Certain MSRs are employed by the CPU to save host mode state. As an example, on AMD-V CPUs, the VM_HSAVE MSR is used to set the host mode save area which describes the physical memory region used to save certain host mode runtime state. A guest environment that can change the contents of this MSR can then change the host mode registers and compromise hypervisor integrity. Memory Type Range Registers (MTRRs) are another type of MSR which are used to set caching policy for a range of physical memory. A guest environment can setup the MTRRs such that the CPU cache contents can be accessed and manipulated at runtime [50]. Since parts of hypervisor code and data will often be in the CPU cache, such MTRR manipulation can be used to compromise hypervisor integrity if the hypervisor's page tables for a guest map hypervisor code or data with read permissions.[8] Therefore, an integrity-protected hypervisor must have an independent set of critical CPU registers which are *always* in effect when the CPU is operating in the host mode.

The CPU data registers are used for data movements to and from system memory. Data registers can be divided into: integer, floating-point, and multimedia registers. Guest modes can run a full-fledged OS which typically use these data registers for their functioning. If a hypervisor uses these registers (or a subset of them) for its operation, values of these registers carried over from the guest environment during an intercept can result in the compromise of hypervisor data integrity. Therefore, an integrity-protected hypervisor must either set data registers to a defined state (e.g., zero them) or save and restore contents of used data registers during control transfers to and from guest modes.

Rule 6 *An integrity-protected hypervisor* **requires** *the MMU to maintain independent states for the hypervisor and guest environments.*

Discussion: The MMU is the CPU component that includes support for virtual memory (using paging) and memory access protections. The MMU interface is exposed via a set of CPU registers. The MMU also employs a Translation Lookaside Buffer (TLB) for caching address translations when using virtual memory. Since the MMU is involved in nearly every instruction that is executed by the CPU, a guest environment can compromise hypervisor memory regions if the MMU register set and internal states are shared between host and guest modes. As an example, if a hypervisor does not have its own TLB, the TLB entry loaded in guest mode can lead to unexpected address translations or access permissions. Therefore, an integrity-protected hypervisor needs the MMU on the CPU to maintain independent states for the hypervisor and guest environments.

Rule 7 *An integrity-protected hypervisor* **must** *intercept all x86 hardware virtualization instructions.*

Discussion: An x86 hardware virtualized CPU provides a set of virtualization specific instructions that are used to create, modify, run and terminate guest environments and to save and load guest environment states. A hypervisor can be compromised if these instructions are allowed to execute within a guest environment. For example, a guest environment could load its own state devoid of protections set by the hypervisor. However, an integrity-protected hypervisor can choose to implement recursive virtualization by emulating such instructions.

Definition 5. *We define* containerization *as the process by which a hypervisor isolates some given code and associated data and executes them under complete control.*

Rule 8 *An integrity-protected hypervisor* **must** *containerize any System Management Mode code and BIOS, option ROM or Power Management Scripts it uses.*

Discussion: SMM code, BIOS, option ROMs, and Power Management Scripts are low-level code that have unrestricted access to all system resources such as critical CPU registers, memory, and device IO locations. A buggy or malicious #SMI handler can therefore access memory regions belonging to the hypervisor and compromise its integrity [12, 50]. Malicious code can be embedded within the BIOS, option ROM, or Power Management Scripts [16–18] and these in turn can alter hypervisor memory regions. Therefore, if an integrity-protected hypervisor requires the use of BIOS, option ROM, or Power Management Script code, it must run them in isolation (e.g., in a VM). Further, since #SMIs can occur at any point during system operation, an integrity-protected hypervisor must always containerize any SMM code regardless of the CPU operating mode.

Rule 9 *An integrity-protected hypervisor* **must** *prevent system devices from directly accessing hypervisor memory.*

[8] Note that an adversary cannot exploit cache synchronization protocols in multi-core CPUs in order to manipulate CPU cache contents. All current x86 CPUs supporting hardware virtualization implement cache coherency in hardware, thereby maintaining a uniform view of main memory.

Discussion: An integrity-protected hypervisor can choose to let a VM use a physical device without employing any form of device virtualization. Alternatively, the hypervisor might need to virtualize a physical device and let the VM use a virtual device. As an example, in many systems, a single physical USB controller device controls all the available USB ports. The only way to share the USB ports between VMs would be to present each VM with its own virtual USB controller device that then controls a subset of the physical USB ports on the system. The virtual USB controller devices reside within the hypervisor and interpose on the USB protocol and direct the requests to the appropriate physical USB controller.

USB, Firewire, Storage and Network devices can directly access physical memory via DMA, potentially bypassing the hypervisor. These devices can be programmed by an attacker to access any portion of the physical memory including those belonging to the hypervisor [8]. Malicious firmware on a device can also accomplish the same goal by replacing legitimate physical memory addresses passed to it with hypervisor physical memory regions. Therefore, an integrity protected hypervisor must prevent devices from directly accessing its memory regions.

Rule 10 *An integrity-protected hypervisor* **must** *enumerate all system devices at startup and be able to detect hot-plug devices at runtime.*

Discussion: As discussed in the previous rule, an integrity-protected hypervisor must restrict devices from accessing hypervisor memory regions. This requires the hypervisor to configure memory access restrictions for every device within the system. Consequently, the hypervisor needs to uniquely identify each device. While a device can be uniquely identified (e.g., the bus, device and function triad on a PCIe bus), the identification can change depending on system configuration. As an example, the triad on the PCIe bus is dependent on the physical location of the device on the system board, which may change between hardware upgrades. Therefore, an integrity-protected hypervisor must always enumerate all system devices during its startup to configure permissible memory regions for each device. Further, with hot-plugging capabilities in current systems (where a device can be added or removed from the system at runtime), an integrity-protected hypervisor must be able to dynamically detect such additions and removals and enforce memory access restrictions for such devices. (An alternative, non-technical solution is to maintain stringent physical security to prevent devices from being hot-plugged. This may not be economical in practice.)

Definition 6. *We define critical system devices to be devices that must be properly managed to prevent hypervisor integrity compromise. On the x86 hardware-virtualized architecture, these devices are the functional equivalents of the northbridge, southbridge, and IOMMU, as they can constrain the behavior of all other devices.*

Rule 11 *An integrity-protected hypervisor* **must** *prevent access to critical system devices at all times.*

Discussion: Critical system devices, like any other device, can expose their interface through either legacy IO or memory-mapped IO. For example, Intel-VT systems expose the IOMMU as a DMA device through ACPI while AMD-V systems expose the IOMMU as a PCI device. A VM on top of the hypervisor may perform direct IO to these devices, effectively compromising the integrity of the hypervisor. Therefore, an integrity-protected hypervisor *must* prevent access to these critical system devices at all times.

3.3 Design Rule

A hypervisor's runtime integrity can be compromised by manipulating its memory regions. On an x86 hardware virtualized platform memory can be accessed either via code executing on the CPU or system devices using DMA. In this section, we discuss the rule governing the design of an integrity-protected hypervisor in the above context.

Rule 12 *An integrity-protected hypervisors' code* **must** *be free of vulnerabilities.*

Discussion: A hypervisor needs to be configured with guest environment state (guest OS and allocated resources) before a guest environment can be run. Further, contingent on hypervisor design, configuration changes can be needed at runtime during guest environment operation (e.g., adding or removing resources at runtime). Depending on the hypervisor design, inter-VM communication (e.g., drag and drop between different guest environments) and guest runtime interfaces to the hypervisor (e.g., accelerated IO drivers for virtualized devices) might be supported. Such runtime interfaces might also directly access hypervisor data (e.g., accelerated drivers within the guest may access temporary data buffers that are mapped within hypervisor memory regions for fast IO). All these configuration options and interfaces pose significant risk to hypervisor integrity if they are complex [4]. An integrity-protected hypervisor must therefore ensure that such configuration and runtime interfaces are minimal. Further, designers must also ensure that the hypervisor's core operating logic is simple and its code-base is within limits to perform manual and analytical audits to rule out any vulnerabilities [11, 14, 27].

4 Integrity-Protected Hypervisor on AMD-V and Intel-VT

We present details on *if* and *how* the rules described in §3 can be enforced on AMD-V and Intel-VT.

Rule 1 An integrity-protected hypervisor must be initialized via the creation of a dynamic root of trust. To date, hardware- and software-based [38] mechanisms for creating a dynamic root of trust have been proposed. AMD introduced a new CPU instruction called *SKINIT* [6], and Intel introduced a family of instructions called *GETSEC* [22], where *GETSEC* [*SENTER*] is the most similar to AMD's *SKINIT*.

Rule 2 A dynamic root of trust mechanism must allow for an external verifier to ascertain the identity of the code that executed on a system. The TPM's PCR 17 is reset during the establishment of a DRTM. It resets to 0 (20 bytes of 0x00) during successful establishment, and -1 (20 bytes of 0xff) in the event of an error. The contents of the code to be executed in the new environment are sent to the TPM itself, where they are hashed and extended into the newly reset PCR. On the AMD-V this is PCR 17 while on the Intel-VT it is PCR 18. (An Intel-provided *Authenticated Code Module*, or ACMod, is extended into PCR 17 on Intel systems.)

Rule 3 An integrity-protected hypervisor must virtualize physical memory to prevent access to its memory regions. There are both software and hardware approaches to physical memory virtualization.

Software physical memory virtualization performs guest virtual to system physical address translations on behalf of the guest environment by using shadow page tables within the hypervisor. The shadow page tables are synchronized with the guest page tables during guest page table modifications and are enforced using MMU configuration registers of the guest environment. Note that if the guest environment attempts to run without virtual memory support (e.g., real-mode), the switch must be intercepted and virtual memory support must be enabled transparently.

On both AMD-V and Intel-VT, all guest accesses to MMU registers can be intercepted by the hypervisor, which facilitates enforcement of shadow page tables. Further, TLB flushes are performed using the CR3 and CR4 registers and the INVLPG instruction, all of which can be intercepted by the hypervisor to synchronize shadow page tables. Furthermore, CPU mode switches within a VM can be intercepted by the hypervisor on both architectures, to enable virtual memory transparently.

Both AMD-V and Intel-VT have support for hardware physical memory virtualization in the form of nested page tables and extended page tables, respectively. With hardware physical memory virtualization, the guest has its own set of MMU configuration registers and page tables which need not be tracked by the hypervisor. Instead, the guest page tables translate guest virtual addresses into guest physical addresses. The guest physical addresses are then translated to system physical addresses by the MMU using nested (extended) page tables which reside within the hypervisor for each guest. All page faults incurred in the nested (extended) page tables lead to a control transfer to the hypervisor. This guarantees that the hypervisor has full control over system physical memory.

Rule 4 An integrity-protected hypervisor must execute its core in the highest privilege level to allow it to interpose on critical system operations. Consequently, an integrity-protected hypervisor on AMD-V or Intel-VT must run in the host mode in ring 0. On AMD-V, a CPU can be switched to host mode in ring 0 using the CR0 register and by enabling host mode using MSR EFER (once in host mode, MSR VM_HSAVE_PA should be initialized to point to the host save area). On Intel-VT, a CPU can be switched to host mode in ring 0 using the CR0 register and by initializing VMXON region contents, enabling host mode using the CR4 register, and executing the VMXON instruction.

Rule 5 An integrity-protected hypervisor must have an independent set of critical CPU registers and must sanitize values of CPU data registers during control transfers to and from VMs. Both AMD-V and Intel-VT CPUs provide the host and each guest mode with their own set of control, debug, and segment registers. Thus, guest-mode changes to these registers only impact the guest mode operation and *cannot* result in any changes within the host mode.

On both AMD-V and Intel-VT CPUs, certain MSRs are shared between the host and guest modes. AMD-V has support for an MSR Bitmap structure for every guest mode instance. If a bit corresponding to a particular MSR is set in the bitmap, it results in an intercept to the hypervisor when a guest mode accesses the MSR. Intel-VT has a similar mechanism using MSR Lists for guest mode. On AMD-V CPUs an integrity-protected hypervisor must intercept accesses to VM_HSAVE_PA (used to store host mode state), EFER (used to control enabling/disabling virtualization) and the SMRAM MSR (used to control SMRAM access and SMI generation).

In both AMD-V and Intel-VT, MTRRs are implemented by using a set of MSRs. The MTRRs are divided into fixed range (for setting caching type for 1 MB and below) and variable range (for greater than 1 MB). There is also a default-range MTRR which sets the default caching policy for all other physical memory regions apart from the fixed- and variable-range MTRRs. AMD-V CPUs have another MSR related to the MTRRs which is responsible for global configuration of MTRR registers. An integrity-protected hypervisor must intercept access to all the MTRRs and ensure that the memory ranges specified by guests do not belong to the hypervisor.

Both AMD-V and Intel-VT CPUs share integer registers (except registers R/EAX, R/ESP, and R/EFLAGS), floating-point registers and multimedia registers between the host and guest modes. Thus, if a hypervisor makes use of these registers, it *must* sanitize their values across guest mode switches.

Rule 6 An integrity-protected hypervisor requires the MMU to maintain independent states for the hypervisor and guest environments. Both AMD-V and Intel-VT CPUs provide the host and each guest mode with their own set of MMU-related configuration registers. Thus, any changes to these registers only impact MMU operation within the specific mode. Further, both AMD and Intel support Address Space Identifiers (ASID). With ASID, a unique ID can be assigned for each guest; the host mode is always allocated ASID 0. The ASIDs are then used to isolate TLB entries of the hypervisor and guests.

Rule 7 An integrity-protected hypervisor must intercept all x86 hardware virtualization instructions. Both AMD-V and Intel-VT CPUs cause an unconditional guest mode intercept if any virtualization-specific instructions are used within a guest environment. Therefore, a hypervisor *must* handle guest mode intercepts caused due to such instructions [6, 21]. The nature of operations performed on such intercepts is contingent on hypervisor design.

Rule 8 An integrity-protected hypervisor must containerize any System Management Mode code and BIOS, option ROM or Power Management Scripts it uses. Code associated with the BIOS, option ROM, and Power Management Scripts can be contained by running them within a VM. The hypervisor can use a similar technique for SMM code by intercepting SMIs and running the SMI handlers in isolation.

Intel-VT provides support for SMM containerization using *dual-monitor* treatment. With dual-monitor, there is a regular hypervisor and an SMM Transfer Monitor (STM) that is in control of a hardware virtual machine solely for running SMM code. The STM gets control on all SMIs (occurring within the hypervisor as well as guests) and is responsible for running the target SMM code.

SMI handlers in production systems typically need to execute with guaranteed execution response. The fundamental question then with STM is whether it can provide real-time execution guarantees. Given that the STM runs within its own hardware virtual machine, the CPU would have to save and restore entire hardware virtual machine execution contexts, which would incur non-negligible runtime cost. Furthermore, given the fact that there are currently no Intel CPUs that implement STM, it is impossible to precisely evaluate whether the STM model is applicable in practice.

AMD-V on the other hand only supports interception of SMIs occurring in the guest mode. Thus, while an integrity-protected hypervisor on the AMD-V can intercept such guest mode SMIs and run them within a VM, it *must* disable SMI generation when in host mode. This can be done by controlling the SMRAM MSRs. However, disabling SMI generation in such a fashion results in two problems in practice: (i) an SMI can occur during the time taken to perform a transition from guest to host mode and before SMI generation is disabled. Such an SMI results in a SMM handler that executes without any form of isolation, thereby potentially compromising hypervisor integrity, and (ii) disabling SMI altogether would result in a system freeze on most platforms which require certain SMM code to execute periodically (e.g., system temperature sensors).

Rule 9 An integrity-protected hypervisor must prevent system devices from directly accessing hypervisor memory regions. The IOMMU is the only system device that can intervene between DMA transactions occurring between a device and memory and hence must be employed by an integrity-protected hypervisor to protect its memory regions from direct access by devices. Both AMD-V and Intel-VT provide an IOMMU as a part of the northbridge. The IOMMU on both architectures allows each peripheral device in the system to be assigned to a set of IO page tables. When an IO device attempts to access system memory, the IOMMU intercepts the access, determines the domain to which the device has been assigned, and uses the IO page tables associated with that device to determine whether the access is to be permitted as well as the actual location in system memory that is to be accessed. An integrity protected hypervisor *must* instantiate IO page tables such that physical addresses corresponding to hypervisor memory regions are marked as inaccessible to any device.

Rule 10 An integrity-protected hypervisor must enumerate all system devices at startup and be able to detect hot-plug devices at runtime. On the PCIe bus, each device is uniquely identified using the bus, device, and function triad. When a hypervisor starts up, it can iterate through all possible bus, device and function locations and query the PCIe configuration space for the triad. If a device is present, the configuration space access returns a valid device identification.

Hot-plug devices on the PCIe bus can be detected by the hypervisor using ACPI. The Hot Plug Parameters (HPP) table is updated by the ACPI subsystem whenever there is a hot-plug device insertion or removal. A hypervisor can periodically scan the table, and obtain the device identification triad on the PCIe bus.

Rule 11 An integrity-protected hypervisor must prevent access to critical system devices at all times. On both AMD-V and Intel-VT CPUs, devices can be controlled via legacy or memory-mapped IO. Legacy IO is performed using a set

of four instructions: IN, OUT, INS, and OUTS. Further, both architectures provide a method for the hypervisor to intercept legacy IO operations on a per-port basis using an IO permission bitmap for each guest. Both AMD-V and Intel-VT support software and hardware physical memory virtualization. An integrity-protected hypervisor can set desired protections using page table entries corresponding to the memory-mapped IO region to intercept accesses.

As seen from the preceding discussions, Rule 1 through Rule 11 depend on the platform hardware and all of them except Rule 8 can be completely implemented on current x86 hardware virtualized platforms. As discussed earlier in this section, Rule 8 cannot be completely implemented as current x86 platforms do not contain adequate hardware support to containerize SMM code. Rule 12 (§3.3) depends on the design of an integrity-protected hypervisor and will be discussed in the following two sections.

5 Guest and Hardware Requirements

In §3, we identified rules that an integrity-protected hypervisor must obey. We further analyzed these rules in the context of commodity hardware from AMD and Intel in §4, and established the feasibility of constructing an integrity-protected hypervisor. In this section, we discuss the impact that such a hypervisor design has on its guests and on hardware requirements. That is, some combinations of guest operating systems may require additional hypervisor functionality that is in conflict with Rule 12.

5.1 Multiple Guests

An integrity-protected hypervisor's ability to support multiple guests is contingent on the hardware and device requirements of the individual guests. For example, an extremely minimal guest that requires only a processor on which to perform basic computations does not introduce any resource contention beyond the CPU time and memory space allocated to it. On the other hand, multiple instances of a fully-interactive, media-rich modern OS may require many of the system's underlying devices to be somehow multiplexed between the guests.

Sharing Devices On a well-behaved system with sufficient peripherals, each guest running on top of the hypervisor can be granted access to a disjoint set of devices. For example, web servers belonging to mutually distrusting entities can each be allocated their own network interface, disk controller, and set of drives. The hypervisor can protect itself by properly configuring the IOMMU, as previously discussed.

However, in certain cases a hypervisor needs to share a single hardware device between two or more guests. For example, many systems today actually do have multiple USB controllers, each of which controls what are generally (though to our knowledge there is no requirement that this remain so) a small number of physically proximal ports, e.g., front-panel vs. rear-panel ports. It is technically feasible to assign distinct USB controllers to distinct guests. Unfortunately, today it generally requires trial-and-error to determine which controller is responsible for which physical ports. For, e.g., a USB flash drive containing corporate secrets, this level of uncertainty is not acceptable. Thus, one may be tempted to design the hypervisor to interpose on some USB traffic, with the intention of ensuring that certain devices are only accessible from the appropriate guests. In practice, we fear that this will significantly complicate the hypervisor, and risk breaking compliance with Rule 12.

Guest BIOS Calls Many legacy operating systems – especially closed-source ones – depend on BIOS calls as part of their basic operation. While BIOS calls can be invoked inside of another virtual environment to protect the hypervisor (recall §2.3 and, e.g., Rule 3), these calls can have lasting effects on the relevant devices. In practice, devices manipulated through BIOS calls cannot be shared without a software emulation or virtualization layer to resolve conflicts between multiple guests that attempt to use the device concurrently, or in conflicting operating modes.

A consequence of the above characteristics of many operating systems is that they cannot be readily executed simultaneously on an integrity-protected hypervisor, as the logic necessary to virtualize, emulate, or otherwise multiplex legacy calls such as BIOS calls may drastically increase the complexity of the hypervisor, again threatening Rule 12.

Sharing Between Guests One solution to sharing a single hardware device between multiple guests is to create an additional guest with the sole responsibility of virtualizing a single physical device and exposing multiple instances of a virtual device. The virtual device may appear identical to the physical device behind it, or the virtual device may expose a different interface. Each option has its advantages. This design space has been explored in great detail as it applies to microkernels [15].

From the perspective of an integrity-protected hypervisor, such an architecture requires a means for sharing information between guests. The primary risk to the hypervisor is the inclusion of a larger configuration interface that enables the creation of, e.g., shared memory and message passing mechanisms for guest intercommunication. We note that any mechanism within the hypervisor that attempts to filter or otherwise restrict traffic between guests will have the effect of further complicating the implementation of the hypervisor. The issue of one guest attacking another via the sharing interface is significant, but it is orthogonal to the hypervisor's ability to protect its own integrity.

5.2 Hardware Considerations

Today, a device is deemed "compatible" with a particular platform architecture and operating system if it implements an interface close enough to the relevant specifications that any differences or discrepancies can be remedied within the relevant device driver. This somewhat sloppy approach has security consequences for an integrity-protected hypervisor. We now discuss the importance of correct hardware, and then relate some examples we have encountered in the wild of devices and systems that do not behave as expected.

To keep the hypervisor minimal, it is of utmost importance that peripheral devices are in compliance with the relevant specification or API. Today, devices abound with bugs or compliance issues that are considered minor from a functionality perspective ("fixed" via a software work-around) but potentially create significant security vulnerabilities.

The problem is that buggy or non-compliant devices generally require a work-around in the form of additional device driver code. In the limit, an integrity-protected hypervisor will need to be aware of these work-arounds, in the form of additional hypervisor code. This effectively bloats the hypervisor codebase and precludes formal or manual verification of the hypervisor's correctness and security properties (i.e., violating Rule 12).

Experiences with Devices Here we relate some of our own experience exploring systems built using hardware virtualization and trusted computing support in the context of an integrity-protected hypervisor.

South Bridge Renders DRTM / PCR 17 Unusable An integrity-protected hypervisor must initialize itself using a dynamic root of trust mechanism (§3.1). A critical component of the DRTM is its ability to extend the hash of the newly loaded code into a PCR in the system's TPM. In practice, we have encountered systems that do not update PCR 17 correctly. We have received one report that there is a bug in the southbridge that results in data corruption on the LPC bus [13], thereby rendering the resulting PCR value meaningless. This bug renders infeasible on the affected systems an entire class of trustworthy computing systems.

SMRAM locked by the BIOS resulting in no SMI intercept Rule 8 states that an integrity-protected hypervisor must containerize any System Management Mode code (§2.2). For a BIOS which does not authenticate SMM code, an integrity-protected hypervisor can containerize SMM code by intercepting #SMIs and executing #SMI handlers within a VM (§4, Rule 8). However, we have encountered systems in the wild where the SMRAM is locked by a non-integrity measured BIOS, thereby preventing an SMI intercept from being generated when the CPU is in guest mode. In other words, an SMM handler (as a result of an SMI) can execute in SMM mode without the hypervisor having any control over it. This leaves a hypervisor on such hardware potentially vulnerable to malicious SMM handler code.

iTPM The v1.2 TPM specification states that all TPM chips must expose the same, well-defined interface [43]. In practice, this is not the case. For example, the TPM in the Intel GM45 chipset returns an incorrect status message, substituting the `VALID` TIS status message when it should return the `DATA_EXPECT` status [42]. This seemingly minor issue can be worked-around in a few lines of code. However, this serves to illustrate the risk posed to an integrity-protected hypervisor by the plethora of devices available today. If each device requires even just a few lines of code in the hypervisor, then the hypervisor's code size is likely to escalate to the point where compliance with Rule 12 (no vulnerabilities in the hypervisor) is intractable.

The Importance of Correct Hardware If integrity-protection for the hypervisor is a priority, then non-compliant devices are unacceptable. Given that today's market remains largely dominated by features and performance, this situation is troubling. An integrity-protected hypervisor may have to include a blacklist of known non-conformant devices. Or, if blacklists fail to scale, then such a hypervisor may have to include a whitelist of the few hardware devices known to provide the required properties. In practice, this will likely increase the cost of systems.

6 Popular Hypervisors

We now present the designs of popular Type 1 [30] hypervisors and discuss the impact of such designs on the rules discussed in §3. To keep our discussion focused we choose VMware ESX Server, Xen, Hyper-V, L4 and SecVisor as our examples. We believe they encompass the current hypervisor spectrum from general-purpose to ad-hoc.

Figure 3 shows the hypervisors and the integrity rules that each of them adhere to. As seen, no hypervisor adheres to all the rules. None of the hypervisors except Xen and SecVisor load by establishing a dynamic root of trust and hence violate Rule 1. However, adding support to adhere to Rule 1 should be fairly straightforward. More importantly, none of the hypervisors containerize (or can containerize) SMM code (§2.2, §4-Rule 8), thereby violating Rule 8. Finally, implementation of a design of a particular hypervisor leads to increased hypervisor code/data and attack surface that violates Rule 12 as described in the following paragraphs.

A VMware virtual environment consists of the hypervisor and drivers for all supported platform hardware. It also consists of a service console which is used for initial system configuration and ongoing management tasks [45]. The hypervisor also provides a standard API which enables configuration and management via local and/or remote

Integrity Rules	Hypervisors				
	VMware	Xen	Hyper-V	L4	SecVisor
1. An integrity-protected hypervisor must be initialized via the creation of a dynamic root of trust.	✗	✓	✗	✗	✓
2. A dynamic root of trust mechanism must allow for an external verifier to ascertain the identity of the code that has received control in the new execution environment.	✓	✓	✓	✓	✓
3. An integrity-protected hypervisor must employ physical memory virtualization to prevent any code executing within a VM from accessing hypervisor memory regions.	✓	✓	✓	✓	✓
4. An integrity-protected hypervisor must execute its core in the highest privilege level that allows it to interpose on critical system operations.	✓	✓	✓	✓	✓
5. An integrity-protected hypervisor must have an independent set of critical CPU registers and must sanitize values of CPU data registers during control transfers to and from VMs.	✓	✓	✓	✓	✓
6. An integrity-protected hypervisor requires the MMU to maintain independent states for the hypervisor and guest environments.	✓	✓	✓	✓	✓
7. An integrity-protected hypervisor must intercept all x86 hardware virtualization instructions.	✓	✓	✓	✓	✓
8. An integrity-protected hypervisor must containerize any SMM code, BIOS, option ROM or Power Management Script it uses.	✗	✗	✗	✗	✗
9. An integrity-protected hypervisor must prevent system devices from directly accessing hypervisor memory regions.	✓	✓	✓	✓	✓
10. An integrity-protected hypervisor must enumerate all system devices at startup and be able to detect hot-plug devices at runtime.	✓	✓	✓	✓	✓
11. An integrity-protected hypervisor must prevent access to critical system devices at all times.	✓	✓	✓	✓	✓
12. An integrity-protected hypervisors' code must be free of vulnerabilities.	✗	✗	✗	✗	✓*

Fig. 3: No existing hypervisors adhere to all of our integrity rules. In particular, no hypervisor supports (or can support) containerization of SMM code (Rule 8). Note that Xen adheres to Rule 1 using OSLO [26] or tboot [40] at boot time. *SecVisor has gone through a rigorous formal verification process that proves the correctness of its memory protection scheme [14].

applications. Since the VMware hypervisor is monolithic by design, it results in an increased code base (since all device drivers are present in memory irrespective of the platform hardware). The VMware ESX server core is reported to have around 500K lines of code [46]. This can lead to vulnerabilities within the hypervisor core itself [3]. Further, the local and/or remote management interfaces can be exploited in order to execute code with hypervisor privileges [5].

A Xen virtual environment consists of the Xen hypervisor, Domain-0, Domain Management and Control (DMC) software, and paravirtualized and hardware-virtualized guest domains [51]. Domain-0 is a paravirtualized Linux kernel and is a unique virtual machine running on the Xen hypervisor that has special rights to access physical I/O resources as well as interact with the other guest domains. DMC supports the overall management and control of the virtualization environment and executes within Domain-0. Domain-0 and DMC is required to be running before any other guest domains can be started. The Xen hypervisor also exposes a set of hypercalls which can be used by both Domain-0 and guest domains to directly interact with the hypervisor.

Xen Domain-0 uses standard Linux OS drivers to control system hardware. Hence Domain-0 has privileged access to most system hardware. Further, since the Domain-0 TCB is large (due to an entire Linux kernel and supporting drivers) it is easy to exploit vulnerabilities within the OS to gain root access in Domain-0 [1, 2]. Once within Domain-0, an attacker can employ DMA accesses, Xen hypercall functionalities and DRAM controller programming in order to access Xen Hypervisor memory regions [47, 48].

Hyper-V virtualization consists of *partitions*. A partition is a logical unit of isolation, supported by the hypervisor, in which operating systems execute. The Hyper-V hypervisor must have at least one parent, or root, partition, running Windows Server 2008 [29]. The virtualization stack runs in the parent partition and has direct access to the hardware devices. The root partition then creates the child partitions (using the hypercall API) which host the guest operating systems. Given that the Hyper-V root partition consists of a complete Windows Server 2008 installation, any vulnerability that affects Windows Server 2008 will affect Hyper-V. For example, last fall, a routine Windows patch resulted in the compromise of the Hyper-V parent partition [28].

L4 employs a microkernel approach towards virtualization [19]. It provides basic resource and communication abstractions to which guest OSes need porting. While the current L4 code-base has not been verified, the size of the L4 microkernel (order of 10K) suggests that it should be amenable to formal verification. seL4, an embedded microkernel for ARM processors based on L4 has been mathematically verified for correctness [27]. A guest OS requires considerable changes to be ported to L4. This may not be a viable solution for commodity OSes such as Windows. Also, the seL4 microkernel has been verified only on the ARM architecture and the verification process would reportedly take around 10 person years if done again [27]. Thus, it is unclear whether the verification approach is viable on a much more complex architecture such as x86.

SecVisor is a hypervisor which provides lifetime OS kernel code integrity. It is a *pass-through* hypervisor and does not virtualize system devices or resources. Instead, it manipulates page protections to ensure that only approved kernel-code can execute during runtime [39]. The SecVisor architecture is simple and its code base is very small (around 8000 SLOC). It has been formally verified for correctness [14].

7 Related Work

Karger discusses hypervisor requirements for multi-level secure (MLS) systems [25]. He argues that hypervisors are conceptually divided into Pure Isolation and Sharing hypervisors, with Pure Isolation hypervisors only being practical on extremely expensive hardware. Karger's discussion is largely centered on systems that have received some type of security evaluation, e.g., Common Criteria [24]. Implicit in the ability of a hypervisor to receive such certification is that hypervisor's ability to protect its own integrity, which is the focus of the present work.

Roscoe et al. argue that virtualization research today is clouded by commercial applicability, and has not been sufficiently bold in exploring fundamentally new OS and virtualization designs [33]. They also argue that one of the things enabled by hardware virtualization support today is for research systems to become usable for "real work" much more quickly than was previously possible. While the current paper is guilty of not breaking fundamentally new ground, we feel that it is important to explore the limitations of current hardware so that future researchers are not lulled into a false sense of security.

Bratus et al. argue for a significant architectural change to the MMU in today's systems to better enable integrity measurements to reflect the true state of an executing system, and not just that system's load-time state [9]. Their primary concern is to address the time-of-check, time-of-use (TOCTOU) vulnerability inherent in today's TPM-based integrity measurement solutions. We agree that this is a significant limitation, and believe that solutions to such limitations should be complimentary to secure hypervisor designs.

8 Conclusions

We explored the low-level details of x86 hardware virtualization and established rules that an integrity-protected hypervisor must follow. In the process, we identified a number of discrepancies between specification and practice that can potentially compromise a hypervisor's integrity on these platforms. We conclude that while in theory the latest x86 hardware contains sufficient support to integrity-protect a hypervisor, in practice an integrity-protected hypervisor cannot be realized on today's x86 hardware platforms. As an example, System Management Mode (SMM) code that exists on all x86 platforms runs at a higher privilege than the hypervisor itself. Current x86 platforms do not provide adequate hardware support to isolate such SMM code. We also conclude that an integrity-protected hypervisor will be unable to support arbitrarily many legacy guests. Sharing devices and data between multiple guests coupled with guest BIOS invocations significantly complicates the hypervisor, which can result in vulnerabilities that compromise hypervisor integrity. Also, in constructing a system that truly protects hypervisor integrity, hardware must be selected with great care, as numerous devices that exist in the wild fail to adhere to relevant specifications.

We believe the rules presented in this paper represent a strong first approximation of what is necessary to realize an integrity-protected hypervisor. Such a hypervisor will be capable of maintaining its own integrity and preventing mutually distrusting guests from compromising each other's or the hypervisor's integrity. However, we have not discussed mechanisms required to create a trusted path to the user or administrator of a system equipped with an integrity-protected hypervisor. Further work is also required to provide protection of guests' secrets in a virtualized environment. Recent research [31] reveals that the days of ignoring side channels and timing channels are behind us, as these represent very real and practical threats. We hope that this paper will serve as a solid starting point and a call to action for additional investigation into these significant challenges.

9 Acknowledgements

This research was supported in part by CyLab at Carnegie Mellon under grants DAAD19-02-1-0389 from the Army Research Office, and by grant CNS-0831440 from the National Science Foundation. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of ARO, CMU, CyLab, NSF, or the U.S. Government or any of its agencies.

References

1. Elevated privileges. CVE-2007-4993, 2007.
2. Multiple integer overflows allow execution of arbitrary code. CVE-2007-5497, 2007.
3. The CPU hardware emulation does not properly handle the Trap flag. CVE-2008-4915 (under review), 2008.
4. Directory traversal vulnerability in the shared folders feature. CVE-2008-0923 (under review), 2008.
5. Multiple buffer overflows in openwsman allow remote attackers to execute arbitrary code. CVE-2008-2234, 2008.
6. AMD64 virtualization: Secure virtual machine architecture reference manual. AMD Publication no. 33047 rev. 3.01, 2005.

7. J. P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Air Force Electronic Systems Division, Hanscom AFB, 1972.
8. A. Boileau. Hit by a bus: Physical access attacks with firewire. RuxCon, 2006.
9. S. Bratus, N. D'Cunha, E. Sparks, and S. W. Smith. TOCTOU, traps, and trusted computing. In *Proc. Conference on Trusted Computing and Trust in Information Technologies (TRUST)*, 2008.
10. R. Budruk, D. Anderson, and T. Shanley. *PCI Express System Architecture*. Addison-Wesley, 2004.
11. A. Datta, J. Franklin, D. Garg, and D. Kaynar. A logic of secure systems and its applications to trusted computing. In *Proc. IEEE Symposium on Security and Privacy*, 2009.
12. L. Duflot, O. Levillain, B. Morin, and O. Grumelard. Getting into the SMRAM: SMM reloaded. Central Directorate for Information Systems Security, 2009.
13. R. Findeisen. Buggy south bridge in HP dc5750. Personal communication, Apr. 2008.
14. J. Franklin, A. Seshadri, N. Qu, S. Chaki, and A. Datta. Attacking, repairing, and verifying SecVisor: A retrospective on the security of a hypervisor. CMU Cylab Technical Report CMU-CyLab-08-008, 2008.
15. H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of microkernel-based systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 1997.
16. J. Heasman. Implementing and detecting a PCI rootkit. NGSSoftware Insight Security Research, 2006.
17. J. Heasman. Implementing and detecting an ACPI BIOS rootkit. Black Hat USA, 2006.
18. J. Heasman. Hacking the extensible firware interface. Black Hat USA, 2007.
19. G. Heiser, K. Elphinstone, I. Kuz, G. Klein, and S. M. Petters. Towards trustworthy computing systems: Taking microkernels to the next level. In *Proc. ACM Operating Systems Review*, 2007.
20. Hewlett-Packard et al. Advanced configuration and power interface specification. Revision 3.0b, Oct. 2006.
21. Intel virtualization technology specification for the IA-32 Intel architecture. Intel Publication no. C97063-002, Apr. 2005.
22. Intel trusted execution technology – measured launched environment developer's guide. Document no. 315168-005, June 2008.
23. Intel Corporation. The extensible firmware interface specification. http://www.intel.com/technology/efi/, 2002.
24. International Organization for Standardization. Information technology – Security techniques – evaluation criteria for IT security – Part 1: Introduction and general model, Part 2: Security functional requirements, Part 3: Security assurance requirements. ISO/IEC 15408-1, 15408-2, 15408-3, 1999.
25. P. A. Karger. Multi-level security requirements for hypervisors. In *Proc. Annual Computer Security Applications Conference (ACSAC)*, Dec. 2005.
26. B. Kauer. OSLO: Improving the security of Trusted Computing. In *Proc. USENIX Security Symposium*, Aug. 2007.
27. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proc. SOSP*, 2009.
28. Microsoft. Microsoft technet MS08-067: Vulnerability in server service could allow remote code execution, 2008.
29. Microsoft. Hyper-V architecture. Microsoft Developers Network, 2009.
30. G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Comm. ACM*, 17, 1974.
31. T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2009.
32. J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proc. USENIX Security Symposium*, 2000.
33. T. Roscoe, K. Elphinstone, and G. Heiser. Hype and virtue. In *Proc. HotOS Workshop*, May 2007.
34. J. Rutkowska. Subverting Vista kernel for fun and profit. SyScan and Black Hat Presentations, 2006.
35. A. L. Sacco and A. A. Ortega. Persistent BIOS infection. Core Security Technologies, 2009.
36. J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proc. IEEE*, 63(9):1278–1308, Sept. 1975.
37. SecuriTeam. Opteron exposed: Reverse engineering AMD K8 microcode updates. SecuriTeam Security Reviews, 2004.
38. A. Seshadri, M. Luk, E. Shi, A. Perrig, L. VanDoorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proc. SOSP*, 2005.
39. A. Sheshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proc. SOSP*, 2007.
40. tboot. Trusted boot. http://sourceforge.net/projects/tboot/, 2009.
41. P. Technologies. Phoenix securecore. http://www.phoenix.com, 2009.
42. tpmdd-devel. TPM driver problem on GM45. TPM Device Driver Mailing List, Dec. 2008.
43. Trusted Computing Group. PC client specific TPM interface specification (TIS). Ver. 1.2, Rev. 1.0, July 2005.
44. Trusted Computing Group. Trusted platform module main specification, Part 1: Design principles, Part 2: TPM structures, Part 3: Commands. Version 1.2, Revision 103, July 2007.
45. VMware. VMware ESX server system architecture. `http://www.vmware.com/support/esx21/doc/esx21_admin_system_architecture.html`, 2009.
46. VMware Communities. ESX 3.5 or Xen 4.1? `http://communities.vmware.com/message/900657`, 2008.
47. R. Wojtczuk. Detecting and preventing the Xen hypervisor subversions. Invisible Things Lab, 2008.
48. R. Wojtczuk. Subverting the Xen hypervisor. Invisible Things Lab, 2008.
49. R. Wojtczuk and J. Rutkowska. Xen 0wning trilogy. Invisible Things Lab, 2008.
50. R. Wojtczuk and J. Rutkowska. Attacking SMM memory via Intel CPU cache poisoning. Invisible Things Lab, 2009.
51. XenSource. Xen architecture overview. Version 1.2, Feb. 2008.