HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity *

Ahmed M. Azab Peng Ning Zhi Wang Xuxian Jiang
Department of Computer Science, North Carolina State University
{amazab, pning, zhi_wang, xuxian_jiang}@ncsu.edu

Xiaolan Zhang
IBM T.J. Watson Research Center
cxzhang@us.ibm.com

Nathan C. Skalsky
IBM Systems & Technology Group
nskalsky@us.ibm.com

ABSTRACT

This paper presents *HyperSentry*, a novel framework to enable integrity measurement of a running hypervisor (or any other highest privileged software layer on a system). Unlike existing solutions for protecting privileged software, HyperSentry does not introduce a higher privileged software layer below the integrity measurement target, which could start another race with malicious attackers in obtaining the highest privilege in the system. Instead, HyperSentry introduces a software component that is properly isolated from the hypervisor to enable *stealthy* and *in-context* measurement of the runtime integrity of the hypervisor. While stealthiness is necessary to ensure that a compromised hypervisor does not have a chance to hide the attack traces upon detecting an up-coming measurement, in-context measurement is necessary to retrieve all the needed inputs for a successful integrity measurement.

HyperSentry uses an out-of-band channel (e.g., Intelligent Platform Management Interface (IPMI), which is commonly available on server platforms) to trigger the stealthy measurement, and adopts the System Management Mode (SMM) to protect its base code and critical data. A key contribution of HyperSentry is the set of novel techniques that overcome SMM's limitation, providing an integrity measurement agent with (1) the same contextual information available to the hypervisor, (2) completely protected execution, and (3) attestation to its output. To evaluate HyperSentry, we implement a prototype of the framework along with an integrity measurement agent for the Xen hypervisor. Our experimental evaluation shows that HyperSentry is a low-overhead practical solution for real world systems.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*invasive software*

General Terms

Security

Keywords

Virtualization, Hypervisor Integrity, Integrity Measurement

1. INTRODUCTION

A hypervisor, a.k.a. Virtual Machine Monitor, is a piece of software that manages the sharing of a hardware platform among multiple guest systems. Hypervisors have a relatively small code base and limited interaction with the external world. Thus, they were assumed to be well-protected and easily verifiable. Therefore, hypervisors played an important role in many security services proposed recently (e.g., Terra [14], Lares [23], HIMA [3], vTPM [5] and SIM [30]).

Unfortunately, hypervisors did not turn out to be completely secure. A perfect example is Xen [1], which is a popular hypervisor used in Amazon's Elastic Compute Cloud (EC2) [2]. Recent attacks showed that Xen's code and data can be modified at runtime to allow a backdoor functionality [36]. Although all known backdoors were immediately patched, the growing size of Xen (currently ∼230K lines of code) clearly indicates that there would be more vulnerabilities and consequently more attacks. As a matter of fact, there are at least 17 vulnerabilities reported for Xen 3.x [28].

The growing size of hypervisor's code base is not limited to Xen. It is a general trend in most popular hypervisors due to the need of supporting multiple production hardware and combinations of different guest operation modes. For instance, there are at least 165 vulnerabilities reported in the VMware ESX 3.x bare-metal hypervisor [27].

The above discussion indicates that hypervisors do face the same or similar integrity threats as traditional operating systems. It is thus necessary to protect, measure, and verify the integrity of hypervisors. It is relatively easy to verify the integrity of the hypervisor at system boot time (e.g., via trusted boot). The true challenge lies in the measurement of hypervisor integrity at runtime.

*This work is supported by the U.S. Army Research Office (ARO) under grants W911NF-09-1-0525 and W911NF-08-1-0105 (the later is managed by NCSU Secure Open Systems Initiative (SOSI)), the U.S. National Science Foundation (NSF) under grant 0910767, and an IBM Open Collaboration Faculty Award. The authors would like to thank David Doria and Andy Rindos at IBM for facilitating the conversation with IBM production groups. The authors would also like to thank the anonymous reviewers for their insightful comments that helped improve the paper's presentation.

CCS'10, October 4–8, 2010, Chicago, Illinois, USA.

1.1 Previous Attempts

The protection of the highest privileged software on a computer system has been investigated for several decades. A common approach is to place another smaller higher privileged software layer (e.g., a hypervisor or a micro-kernel) below the intended measurement target. However, we have repeatedly encountered the same issue: Once a higher privileged software is introduced (and becomes the new highest privileged software), how to provide protection for this newly introduced software layer?

A few recent attempts [24, 36, 22, 34] tried to tackle a more specific problem: using an independent component (that is out of the control of the highest privileged software) to measure the integrity of the highest privileged software on a system. Unfortunately, none of them was successful.

Copilot [24] uses a special PCI device dedicated to measure the code base of a running system. Although Copilot is neither modifiable nor detectable by the running system, it has two main drawbacks. First, there is a semantic gap between the code running on the PCI device and the running system. The PCI device cannot access the CPU state (e.g., `CR3` register), which is essential to the integrity measurement process. Second, there are existing attacks [25] that can prevent Copilot, or any other PCI RAM acquisition tool, from correctly accessing the physical memory using the hardware support of protected memory ranges (e.g., Intel VT-d).

HyperGuard [36] and HyperCheck [34] are two frameworks that provide integrity measurement of hypervisors. Both frameworks rely on the SMM, which provides hardware protection for the integrity measurement code. However, both HyperCheck and HyperGuard suffer from serious limitations. First, none of these frameworks provide a way to trigger the integrity measurement without alerting the Hypervisor. Hence, they are both vulnerable to the *scrubbing* attack, where a compromised hypervisor can clean up the attack traces before the integrity measurement starts. Second, neither HyperGuard nor HyperCheck solve the technical problems associated with running a measurement agent in the SMM on a platform with hardware assisted virtualization (e.g., Intel VT). For example, the hypervisor context is hidden in the CPU if the SMM interrupts a guest VM rather than the hypervisor [7].

Flicker [22] uses the measured late launch capability to run a verifiable and protected program from a secure state. One of the proposed uses of Flicker is to run a rootkit detector, which can be modified to do integrity measurement, and attest to its output to a remote user. However, Flicker's rootkit detector is directly vulnerable to the scrubbing attack because the measurement target (the running system) is the one responsible for invoking the integrity measurement.

1.2 Challenges

The above review testifies to the limitations of past efforts and challenges we face. To overcome these challenges, a good solution has to have the following capabilities. First, the integrity measurement has to be *stealthily* invoked so that a compromised hypervisor does not get the chance to scrub traces of previous attacks. Achieving this capability is complicated by the fact that hypervisors can capture all events that occur inside the system. Second, the measurement agent has to be *verifiable* despite the hypervisor's ability to tamper with any code or data stored in the system memory. Third, the measurement agent execution has to be *deterministic* and *non-interruptible*. In particular, the hypervisor should not be able to interrupt its execution or modify its intermediate or final measurement output. This property is challenged by the fact that many control-flow transfers (e.g., interrupts) normally trap into the hypervisor. Fourth, the measurement agent should be capable of doing *in-context* measurement that reveals the entire CPU state essential for integrity measurement, given that the hypervisor exclusively runs at the highest privilege level. Finally, the measurement technique should provide *attestation* to the authenticity of the measurement output, given that hypervisors have full control over the system both before and after the measurement process.

1.3 Introducing HyperSentry

In this paper, we present HyperSentry, a framework that supports stealthy in-context integrity measurement of a running hypervisor (or any other highest privileged software). HyperSentry's main focus is not on integrity measurement itself. Instead, HyperSentry aims to provide all the required support for an integrity measurement agent to verify the integrity of the highest privileged software.

HyperSentry differs from the rich body of research on assuring the integrity of privileged software. HyperSentry does not introduce a higher privileged layer to measure the integrity of privileged software. Instead, by harnessing existing hardware and firmware support, HyperSentry introduces a software component properly isolated from the hypervisor to enable the integrity measurement. In other words, HyperSentry relies on a Trusted Computing Base (TCB) composed of hardware, firmware and a software component properly isolated from the highest privileged software.

HyperSentry is triggered by an out-of-band communication channel that is out of the control of the system's CPU and consequently the highest privileged software. HyperSentry uses a novel technique to maintain the stealthiness of its invocation despite the ability of the hypervisor to block or reroute all the system's communication channels. The out-of-band channel is used to invoke a System Management Interrupt (SMI) on the target platform to trigger HyperSentry. In this paper, we use Intelligent Platform Management Interface (IPMI) [17], which is commonly available on server platforms, to establish this out-of-band channel. Nevertheless, HyperSentry can also use any mechanism that can trigger SMIs without going through the CPU (e.g., Intel Active Management Technology (AMT) [16]).

HyperSentry resides in the SMM, which provides the protection required for its base code. However, the SMM does not offer all the necessary contextual information needed for integrity measurement. To achieve in-context measurement, HyperSentry uses a set of novel techniques to (1) set the CPU to the required context, and (2) provide a verifiable and protected environment to run a measurement agent in the hypervisor context. Hence, it has full access to the correct CPU state and consequently all the required input for integrity measurement. Finally, HyperSentry presents a novel technique to attest to the measurement output.

Although there are existing approaches that rely on the SMM to initiate runtime integrity measurement, HyperSentry solves the real-world problems involved in this process using commodity hardware. In particular, HyperSentry provides a stealthy measurement invocation, in-context integrity

measurement, and attestable output. No existing solution provides these capabilities combined.

We implement a prototype of HyperSentry on IBM BladeCenter H chassis with HS21 XM blade servers [6]. To validate HyperSentry, we implement an integrity measurement agent to verify the integrity of the Xen hypervisor [1]. Xen is chosen because it is both popular and open source, and has previously struggled with some security bugs. However, HyperSentry can be adopted to verify the integrity of any privileged software (e.g., another hypervisor or OS kernel).

We perform a set of experiments to evaluate our HyperSentry prototype. The end-to-end time for the measurement process, excluding output signing, is about 35ms, which is reasonable given that the hypervisor integrity measurement should not be a frequent operation. To support this hypothesis, we use a benchmark to measure the performance overhead on guest VMs. If HyperSentry is periodically invoked every 16s, the average overhead is less than 1.3%.

1.4 Summary of Contributions

We make the following contributions in this paper:

- For the first time, we provide a complete and feasible solution to measure the integrity of privileged system software without placing a higher privileged software layer below. This enables the integrity measurement of the highest privileged software layer on a system.
- We develop a concrete framework to enable an agent to measure the integrity of the highest privileged software. It harnesses the isolation provided by existing hardware and firmware components to build the TCB required by integrity measurement and to offer stealthy invocation, in-context protected execution, and attestation to the measurement agent.
- We solve a set of challenging technical issues associated with the HyperSentry framework (e.g., revealing the hypervisor context upon receiving an SMI, creating an isolated environment for the measurement agent).
- We provide a thorough security analysis of HyperSentry. We prove that it is tamper-proof against any attacks that can compromise the hypervisor. Moreover, the measurement process is neither detectable nor interruptible by a compromised hypervisor.
- We implement a prototype of HyperSentry on an IBM BladeCenter using Xen as the measurement target. We evaluate both the execution time needed by our measurement code and its performance overhead on the overall system performance.

The rest of this paper is organized as follows. Section 2 provides background information on SMM and IPMI. Section 3 discusses our assumptions, threat model, and security requirements. Section 4 presents HyperSentry in detail. Section 5 discusses a case study of HyperSentry using Xen as the measurement target. Section 6 presents the implementation and experimental evaluation of our HyperSentry prototype. Section 7 discusses related work, and Section 8 concludes this paper with some future research directions. Additional implementation details are given in the appendix.

2. BACKGROUND

In this section, we briefly give some background information on System Management Mode (SMM) and Intelligent Platform Management Interface (IPMI), which are used to develop HyperSentry.

SMM: SMM is an x86 operating mode designed to handle system management functions. The CPU enters the SMM upon receiving an SMI, triggered by either software or hardware events. SMM is an independent and protected environment that cannot be tampered by software running on the system. Moreover, SMM's code is stored in a designated, lockable memory called SMRAM. Locking SMRAM (through the memory controller's `D_LCK` bit) prevents all access to it except from within the SMM. Currently, all BIOS manufactures lock the SMRAM before the system boots to prevent SMM misuses. In this research, we installed a customized BIOS on our hardware platform, which allows us to add our SMI handler to the SMRAM before it is locked.

When an SMI is invoked, the hardware saves the current CPU state to a dedicated state save map and switches the context to the SMM. After the SMI finishes, it executes the `RSM` instruction to resume the interrupted CPU operation. All interrupts, including the non-maskable ones, are disabled upon entering the SMM. Thus, even the hypervisor cannot interfere with the SMI handler execution as long as the SMRAM is locked. Additional information on the SMM can be found in [12] and [13], while more detailed information can be found in Intel [7] and AMD [9] developer's manuals.

IPMI: The current HyperSentry prototype takes advantage of IPMI to introduce an out-of-band channel to stealthily trigger the integrity measurement. IPMI is a server-oriented platform management interface directly implemented in hardware and firmware [17]. It is adopted by all major server hardware vendors. The key characteristic of the IPMI is that its management functions are independent of the main processors, BIOS, and system software (e.g., OS, hypervisor), and thus can bypass the hypervisor's observation. IPMI relies on a microcontroller embedded on the motherboard of each server, called the Baseboard Management Controller (BMC), to manage the interface between system management software and platform management hardware. Remote access to IPMI is usually authenticated (e.g., via SSH).

In this research, we use IPMI to reach the BMC on the target platform's motherboard to remotely trigger a hardware SMI, which in turn triggers the integrity measurement of the hypervisor. Nevertheless, the techniques developed in this research can be used with any mechanism that can remotely trigger hardware SMIs (e.g., Intel AMT [16]).

3. ASSUMPTIONS, THREAT MODEL, AND SECURITY REQUIREMENTS

Assumptions: We assume that HyperSentry runs on a system that is equipped with an out-of-band channel that can remotely trigger an SMI (e.g., IBM BladeCenter). We also assume that the target platform is physically secured (e.g., locked in a server room) so that the adversary cannot launch any hardware attack. Moreover, we assume that the target platform is equipped with the TCG's [31] trusted boot hardware (i.e., BIOS with Core Root of Trust for Measurement (CRTM) and Trusted Platform Module (TPM) [32]). Thus, a freshly booted hypervisor can be measured and trusted initially through trusted boot. Finally, we assume that the SMRAM is tamper-proof. Recent incidents showed that attackers were able to subvert the SMRAM using cache poisoning [11, 37]. Fortunately, such attacks can be prevented using proper hardware configurations (e.g., System Management Range Register (SMRR) [7]).

Threat model: Our primary objective is to develop an

effective and efficient stealthy in-context measurement framework. We address the "scrubbing attack", which removes the attack traces upon detecting a measurement attempt. We assume that the adversary, once compromising the hypervisor, will attempt to attack the measurement software and/or forge measurement output. We focus on periodic integrity measurement, and thus do not handle attacks that do not cause a persistent change to the hypervisor.

Note that the exact integrity properties to be measured (e.g., code integrity, control flow integrity) are determined by the measurement agents supported by HyperSentry; they are not the concern of HyperSentry. Nevertheless, in our case study with the Xen hypervisor (Section 5), we do target hypervisor code integrity and integrity of memory isolation between different guest VMs.

Security Requirements: To defend against the above threats, particularly attackers with control of the hypervisor, HyperSentry needs to meet the following requirements:

- (SR1) **Stealthy Invocation:** HyperSentry needs to be invoked without alerting the measurement target. Otherwise, a malicious hypervisor would clean up previous attack traces before the measurement session begins (i.e., the scrubbing attack).
- (SR2) **Verifiable Behavior:** The code base of the measurement agent, along with the input data, should be measured and verified before being invoked. This is critical in ensuring that the adversary cannot modify the measurement agent to influence its output.
- (SR3) **Deterministic Execution:** After the measurement agent is invoked, it should be neither changeable nor interruptible. If a compromised hypervisor regains control during the measurement process, it can scrub attack traces to mislead the measurement agent.
- (SR4) **In-context Privileged Measurement:** The measurement agent should be privileged and in the right context to access the hypervisor's code and data, and to gain full access to the CPU state.
- (SR5) **Attestable Output:** The measurement output needs to be securely conveyed to the remote verifier. The hypervisor should not be able to alter or forge the measurement output.

4. THE HYPERSENTRY FRAMEWORK

Figure 1 shows the architecture of HyperSentry. HyperSentry assumes trust in the IPMI channel that is used to trigger SMIs. (Note that HyperSentry can support any platform that is equipped with an out-of-band channel that can trigger SMIs.) Moreover, it trusts target platform's hardware, including the BMC, the TPM, and the SMRAM's hardware protection mechanism. HyperSentry is composed of two software components: (1) the SMI Handler (located in the SMRAM), and (2) the Measurement Agent (located in the hypervisor).

HyperSentry's Out-of-band Channel: Remote users use the IPMI/BMC out-of-band channel to trigger HyperSentry to start the hypervisor integrity measurement. The main challenge for this channel is how to maintain the stealthiness required to defend against the scrubbing attack.

HyperSentry's SMI Handler: We establish trust in the SMI handler through trusted boot, as shown in Figure 2. The CRTM, which is a part of the BIOS, measures itself and the code to be executed next. This process continues until

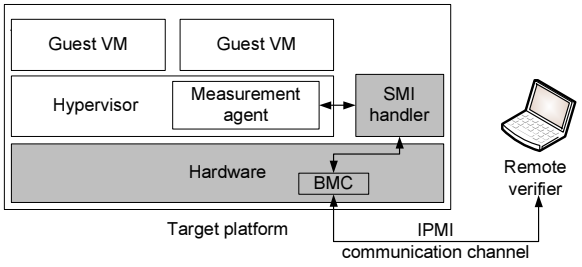

Figure 1: HyperSentry's architecture (Trusted components are shown in gray.)

all components in the boot process are measured. During the trusted boot, an initialization code copies the HyperSentry SMI handler to the SMRAM and locks the SMRAM immediately to prevent any code, regardless of its privilege level, from accessing or modifying it. The initialization code, along with the SMI handler, is measured and extended into one of the TPM's Platform Configuration Registers (PCRs). The TPM can further attest to the PCR values by signing them using a protected attestation key.

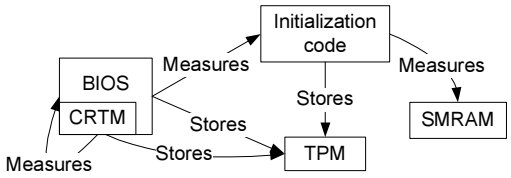

Figure 2: Building trust in SMI handler

Note that the hypervisor is also measured during trusted boot. However, the trust in the hypervisor cannot be sustained due to its interaction with potential attackers. In contrast, no software component can modify the SMI handler's code and data, and the trust in it can be maintained.

HyperSentry's In-context Measurement Agent: The main contribution of this paper lies in our novel techniques to enable an in-context measurement agent. Upon receiving an integrity measurement request, HyperSentry needs to access the hypervisor's code, data, and CPU state to carry out the measurement process. However, the SMM does not provide all the contextual information required by integrity measurement (e.g., the hypervisor's CPU state, which is not entirely retrievable in the SMM). To solve this problem, we develop new techniques that exploit the hardware features to enable the actual measurement agent to run in the protected mode in the context of the hypervisor.

In the following, we present the key techniques in HyperSentry in detail.

4.1 Stealthy Invocation of HyperSentry

Although IPMI/BMC provides an out-of-band channel to invoke HyperSentry, the stealthiness of this channel is still threatened by potential attacks. The reason is as follows. First, the SMIs generated by the BMC can be either disabled or rerouted by the hypervisor. This poses a threat on both the stealthiness and the availability of HyperSentry. Moreover, the hypervisor has the ability to trigger SMIs with different methods (e.g., trapping `in` or `out` instructions). As a result, a compromised hypervisor can mask the original SMI invocation, scrub attack traces, and then invoke a fake

measurement request. To thwart this attack, it is critical for HyperSentry to differentiate between SMIs generated by the out-of-band channel and other fake ones.

To understand how the BMC invokes an SMI, we examined the architecture of an IBM HS21 blade server as an example of platforms that rely on a BMC. Figure 3 shows a high-level block diagram of the components involved in remote SMI invocation on this platform. The BMC has a direct connection to the platform's south bridge (the I/O control hub), or more specifically, the first General Purpose Input port (GPI 0). The south bridge is then connected to the CPU through the north bridge (the memory control hub) and the Front Side Bus (FSB). Note that recent Intel and AMD architectures include the memory control hub as a part of the CPU.

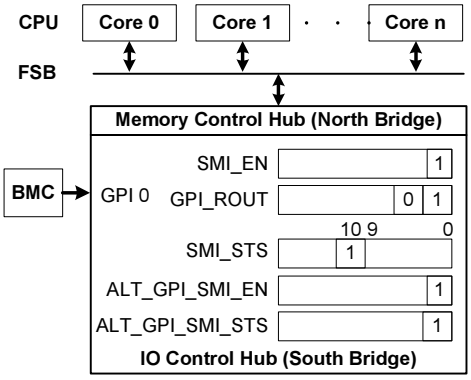

Figure 3: Hardware components involved in remote SMI invocation

In the rest of this subsection, we discuss how to guarantee the stealthiness of the out-of-band channel on this architecture. It is worth mentioning that this method is generic to all platforms that have a connection between the platform management module (e.g., BMC, IBM's Integrated Management Module (IMM) [8] or Intel AMT's Manageability Engine (ME) [16]) and GPI ports on the I/O control hub. However, if the out-of-band SMI is triggered through a connection to a different hardware component (e.g., the SMBus), this technique will need to be slightly adapted to reflect the registers that control this alternative connection.

When the BMC needs to trigger an SMI, it generates a signal on GPI 0. The south bridge responds to this signal according to the values of two registers. The first is `GPI_ROUT`, which specifies the interrupt generated by the GPI. When the two least significant bits have a value of "01", the signal triggers an SMI. (Other values either ignore the signal or trigger other interrupts.) The second register is `SMI_EN`. A low value in that register's least significant bit suppresses all SMIs. While this bit is lockable by the memory controller's `D_LCK` bit, `GPI_ROUT` is always writable by the CPU and consequently the hypervisor.

A set of status registers show the reason of an SMI invocation. Bit 10 of `SMI_STS` indicates that the SMI was triggered by a GPI. The exact GPI port that triggers the SMI is identified by `ALT_GPI_SMI_EN` and `ALT_GPI_SMI_STS`. All status registers are sticky (i.e., they cannot be set by software).

We take advantage of these hardware features to defend against the above threat. Specifically, the HyperSentry SMI handler checks the status registers upon invocation. The measurement process only starts if the SMI is generated by the GPI connected to the BMC. Furthermore, to avoid confusion with other GPI signals, HyperSentry requires that `GPI_ROUT` is configured so that only GPI 0 can generate SMIs. Since the adversary cannot tamper with the hardware, the source of signals generated on that specific GPI cannot be changed. Moreover, a compromised hypervisor cannot fake SMIs triggered by the BMC, because the status registers are non-writable. In other words, attempts to change the SMI generation mechanism will be detected by HyperSentry. A compromised hypervisor may attempt to disable SMI by overwriting `GPI_ROUT`. However, this can be easily detected by the remote user due to the lack of response from HyperSentry. As long as `GPI_ROUT` has the correct value and SMIs can be triggered, a compromised hypervisor will not be able to detect the SMI until our SMI handler is invoked and finishes execution.

4.2 Acquiring Measurement Context

After invoking the SMI, HyperSentry exclusively runs on the target platform's CPU. However, this fact does not prepare HyperSentry sufficiently to carry out the measurement process. Indeed, it is non-trivial to acquire the desired execution context of the hypervisor to ensure a successful measurement. In the following, we use Intel processors, which are used on our experimental platform, to illustrate the challenge and our solution. However, AMD processors have similar components and can easily adopt our solution [9].

The challenge comes from the uncertainty of the CPU's operation mode when it is interrupted by the SMI. On a CPU that supports hardware assisted virtualization (e.g., Intel VT), when interrupted by the SMI, the CPU may run in either the hypervisor (VMX root operation) or one of the guest VMs (VMX non-root operation). In order to measure the integrity of the hypervisor, the measurement agent needs to access the hypervisor's code, data and CPU state. In particular, it needs the VMX data structures, which contain contextual information essential to integrity measurement (e.g., current VMX operation, the hypervisor's VM exit handler and its `CR3` register value). However, when the CPU runs in VMX non-root operation (guest VM) at the time of the SMI, Intel manuals clearly specifies that all pointers to VMX data structures are saved internally to the CPU and cannot be retrieved via software [7].

Although AMD CPUs use the MSR `VM_HSAVE_PA` to point to the physical address containing the host state information, they suffer from a similar limitation: AMD manuals clearly specify that software must not rely on the contents of this state save area because some or all of the host state can be stored in a hidden on-chip memory [9]. This implies that integrity measurement cannot be done unless the CPU is in VMX root operation when interrupted by the SMI.

To overcome this challenge, we develop a novel fallback technique in HyperSentry, which guarantees that the CPU falls back to VMX root operation (i.e., the hypervisor context) using two SMIs, even if the first SMI interrupts VMX non-root operation. HyperSentry acquires the measurement context without allowing the hypervisor to take control of or even detect this fallback.

This fallback technique takes advantage of two architectural components: performance counters and Local Advanced Programmable Interrupt Controller (LAPIC). Performance counters are used to count certain events (e.g., Last Level Cache (LLC) misses) for performance evaluation. Once a

performance counter overflows, an interrupt is generated. The type and destination of this interrupt are determined by the LAPIC, which is responsible for receiving interrupts and forwarding them to the CPU. The LAPIC is configured by a set of registers called the Local Vector Table (LVT); one of these registers controls the interrupts generated by the performance counters.

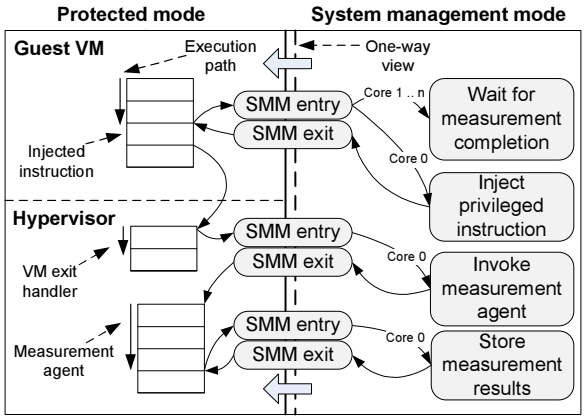

Figure 4: HyperSentry's SMI handling workflow

As illustrated in Figure 4, the main idea behind our fallback technique is to force one of the CPU cores to jump to the hypervisor by injecting an instruction that unconditionally causes a VM exit, which is a jump from the guest VM to the hypervisor. At the same time, performance counters are used, in coordination with the LAPIC, to guarantee that only one instruction is fetched before the CPU returns to the SMM. This technique is carried out by the SMI handler using the following steps: (1) Store the values of all registers along with the next instruction and its address; (2) inject a privileged instruction in place of the next instruction; (3) set one of the performance counters to count cache misses; (4) set the performance counter to overflow as soon as one event is counted; (5) modify the LAPIC so that performance counter overflows cause an SMI; (6) flush the cache; and (7) return from the SMM.

After returning from the SMM, the CPU will directly execute the injected instruction. If the SMI was triggered in VMX non-root operation, the instruction will cause a VM exit. On the other hand, if the SMI was triggered in VMX root operation, this instruction will simply execute. Due to the cache flush, looking up any instruction from the memory will cause a cache miss. In turn, the cache miss will increment the performance counter, which will cause another SMI that jumps back to the SMM. Upon this second SMM entry, the CPU is guaranteed to be in the hypervisor's context.

To find the location to inject the privileged instruction, HyperSentry reads the `EIP` register stored in the SMM's state save map, which contains the virtual address of the instruction to be executed after the SMI returns. Afterwards, HyperSentry walks page tables, using the current `CR3` value, to identify its physical address. However, if the SMM interrupted a guest VM that relies on a hardware assisted paging technique (e.g., Extended Page Tables (EPT)), HyperSentry can directly modify the EPT entry that corresponds to the guest-physical address of the current `CR3`, so that looking up any instruction from the guest memory causes an unconditional VM exit (in the form of an EPT fault). In this case, a pointer to the current EPT can be directly obtained from the SMM's state save map.

A subtle issue still needs to be addressed. As mentioned earlier, all interrupts, including non-maskable ones, are disabled upon entering the SMM. If an interrupt is received during handling the SMI, its handler will be executed right after the SMI handler, thus changing the (planned) execution flow inside the guest VM. To address this issue, HyperSentry needs to inject another copy of the instruction at each interrupt handler, and that interrupt handler will have the same required effect. Interrupt handlers can be located from the SMM using the `LIDT` instruction, which retrieves their virtual addresses. Finally, HyperSentry restores all changes it does as soon as the measurement operation is done.

The fallback technique is completely transparent to the hypervisor because only the injected instruction is executed. Thus, it preserves stealthy invocation (SR1). Moreover, the hypervisor does not get the chance to interrupt the measurement process. The LAPIC setting is deterministic to invoke an SMI as soon as the performance counter overflows. The performance counter overflow is deterministic to occur as soon as a single instruction is looked-up from the memory. Both actions solely depend on the trusted hardware and the hardware configuration.The hardware configuration correctness is guaranteed due to the SMI handler's control over the platform during the first SMI invocation. Thus, this technique preserves deterministic execution (SR3). Finally, assuring that the CPU is running in the hypervisor context provides the correct environment for the measurement agent to provide in-context privileged measurement (SR4).

4.3 In-context Integrity Measurement

The fallback technique discussed above ensures that HyperSentry interrupts the hypervisor's context in the second SMI handling. However, HyperSentry still cannot measure the hypervisor directly yet. There are several reasons: First, there is some limitation in reading the CPU state in the SMM. For example, some of the Intel TXT late launch registers are hidden from the SMM. Although we do not need these registers in the present HyperSentry prototype, this may affect other integrity measurement agents launched through HyperSentry. Second, the SMM is relatively slow. Our experience with Intel Xeon CPUs indicates that the SMM is about two orders of magnitude slower than the protected mode. This slowdown, which can be contributed to the fact that the SMM physical memory needs to be uncacheable to avoid cache poisoning attacks (e.g.,[11],[37]), is anticipated to affect the system performance if the measurement agent runs in the SMM.

To overcome these limitations, we develop several techniques to enable HyperSentry's measurement agent to run in the protected mode within the same context of the hypervisor. These techniques address the following challenges: (1) possible tampering with the measurement agent's code or data before the measurement session, and (2) control-flow change that may send the execution back to a potential adversary controlling the hypervisor.

Measurement Agent Verification: The integrity of the measurement agent needs to be verified before its invocation. This is done by the SMI handler through calculating the hash of the measurement agent's code. HyperSentry locates the measurement agent using its prior knowledge of its virtual address range. It walks the hypervisor's page ta-

bles, using the current CR3 value, to identify its location in the physical memory. Moreover, completely verifiable behavior requires the agent to be developed with the following integrity constraints: (1) It should not include execution jumps to any unmeasured code; (2) its execution path should not rely on any dynamic data (e.g., function pointers) that can change its control flow; (3) it should be stateless (i.e., it should not rely on the output of any previous execution); and (4) it should only rely on unchangeable static data that can be verified through hashing. Fortunately, these requirements can be easily verified by static analysis of the measurement agent's code.

Non-interruptible, Non-modifiable Measurement: If an interrupt or an exception is triggered during the execution of the measurement agent, the control flow will directly jump to the corresponding handler. To retain the isolated execution environment, the SMI handler disables all maskable interrupts by clearing the corresponding bit in the EFLAGS register before jumping to the measurement agent. However, neither exceptions nor Non-Maskable Interrupts (NMIs) can be blocked by disabling interrupts. Although the measurement agent code should avoid exceptions, some exceptions, like those resulting from accessing a non-present memory page, are not avoidable.

To solve this problem, the SMI handler modifies the Interrupt Descriptor Table (IDT) so that it points to a new set of handler functions that are part of the measurement agent. (Note that this modification is done in the SMM via physical memory and thus not detectable by the hypervisor). The original IDT, which is measured as a part of our integrity measurement, is restored as soon as the measurement process is done. As interrupts are disabled, this handler will be only called if an exception or an NMI is called. NMIs are blocked until the measurement is done, while exceptions or other interrupts indicate an unexpected behavior that may require invalidation of the measurement operation.

Malicious DMA writes are another threat that can modify the measurement agent. This threat is handled by verifying that the agent is included in the DMA protected memory ranges provided by Intel VT-d.

Handling Multi-core Platforms: Most today's computing platforms, including our experimental platform, support multi-core CPUs. While each core runs a separate execution thread, all cores share the same physical memory. Multi-core architecture introduces a potential threat: When one core measures the hypervisor, other cores may attempt to either interrupt the measurement process or manipulate the memory to hide attack traces.

HyperSentry mitigates the multi-core CPU threat by temporarily freezing all cores other than the one executing the measurement agent (i.e., keep the non-measurement cores in the SMM until the measurement is done). As shown in Figure 3, the south bridge is connected to all cores through the north bridge and the same FSB. Whenever an SMI is triggered by the south bridge, all cores are interrupted and jump to the SMM. As shown in Figure 4, only the Boot Strap Processor (BSP), denoted as "core 0", will further execute HyperSentry's measurement task; the other cores will freeze in an empty loop until the BSP notifies them about the end of the measurement operation.

The isolated environment provides HyperSentry with the needed security guarantee to satisfy the following security requirements: verifiable behavior (SR2), deterministic execution (SR3) and in-context privileged measurement (SR4). The checksum of the measurement agent is used by the remote user to identify the binary, and hence the exact behavior, of the invoked measurement agent. The interrupt handling and other CPU cores suspension provide a non-interruptible (and hence deterministic) execution. Finally, the code runs in the hypervisor context, the most privileged CPU execution mode.

4.4 Attesting to the Measurement Output

After the measurement operation is done, the measurement agent calls the SMI handler to securely store its output. This output, along with hash value initially calculated of the measurement agent, forms the complete output of the measurement process stored in the SMRAM. The last step in HyperSentry's life cycle is to attest to its measurement output to remote users. Since the out-of-band channel that starts the measurement is one-way, the main challenge is how to deal with the absence of a secure communication channel from the SMM back to remote users.

One approach to handle this challenge is to generate evidence that proves the integrity of the measurement output. However, the TPM, which is the hardware commonly used to attest to measurement output, cannot be used in our case. This is because the hypervisor has direct control over the TPM except when HyperSentry measures the hypervisor. Thus, a compromised hypervisor can block HyperSentry's invocation, as discussed in Section 4.1, and use the TPM to generate a forged integrity evidence.

Our solution takes advantage of the following observations: (1) The system software, including the hypervisor, is initially trusted at system boot, and (2) The hypervisor does not have access to the SMRAM.

To exploit these observations, HyperSentry generates a public/private key pair during the system boot. Before locking the SMRAM, the initialization code stores the private key K_{smm}^{-1} in the SMRAM and extends the public key K_{smm} to one of the TPM's static PCRs. The authenticity of the public key is guaranteed due to the fact that the TPM's static PCRs can only be changed by extending (hashing) its current value with a new value. Thus, the history of values stored in these PCRs, including the initialization code and K_{smm}, cannot be changed. The confidentiality of the private key is guaranteed by locking the SMRAM after storing K_{smm}^{-1} so that it cannot be accessed by the hypervisor.

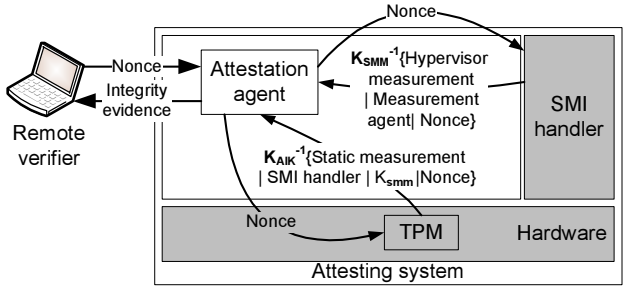

Figure 5: HyperSentry's attestation process

Figure 5 shows the attestation process. To retrieve the output of the measurement process, remote users send a request, accompanied with a fresh nonce, to an attestation agent, which may run anywhere on the target platform. The attestation agent uses the nonce to generate two dif-

ferent signed values. The first is the static attestation output, signed by the TPM private attestation integrity key (K_{AIK}^{-1}). The AIK is a private key stored inside the TPM, which can only be used to sign the current values of the TPM's PCRs. The second signed value is the output of the measurement agent, which is signed by the SMI handler's private key (K_{smm}^{-1}). The remote user accepts the measurement output only if the private key (K_{smm}^{-1}) matches the public key (K_{smm}) extended in the corresponding PCR of the TPM. In both cases, the attestation agent passes the nonce to the TPM and the SMM to be included in the signatures to guarantee their freshness.

4.5 Security of the HyperSentry Framework

In this section, we discuss how HyperSentry meets all the security requirements presented in Section 3.

Stealthy invocation requirement (SR1), which is essential to protect against the scrubbing attack, is satisfied based on both the physical security assumption and the invocation technique presented in Section 4.1. While physical security provides us with a correctly connected and configured out-of-band channel, the invocation technique assures that HyperSentry detects faked measurement invocations. During the measurement process, the control is never given back to the hypervisor except for the well-controlled context switching instruction. Thus, a potential adversary that resides inside the hypervisor will never have the chance to detect the initiation of the integrity measurement.

As discussed in Section 4.1, a compromised hypervisor can reroute the SMIs generated by the BMC and consequently detect HyperSentry's initiation requests. However, a successful scrubbing attack requires initiating the integrity measurement again after the attack traces are cleaned, and thus will be detected (and ignored) by HyperSentry. The compromised hypervisor may also ignore the BMC-triggered SMIs. In both cases, a remote verifier will detect such attacks by the absence of a fresh integrity measurement output.

Verifiable behavior requirement (SR2) is based on transitive trust starting from the CRTM. HyperSentry's SMI handler is measured as a part of the system boot sequence. Its integrity can be attested using the TPM's standard attestation technique. The measurement agent's code-base and its input data are measured and verified by HyperSentry's SMI handler. In short, any attempt to modify HyperSentry's code or data, regardless of the privilege of the adversary, will be either unsuccessful or detectable.

Remote users can confidently trust HyperSentry because it has a small code base, no interaction with any software running on the target system, and is freshly started every time the measurement operation is initiated.

The deterministic execution requirement (SR3) is fulfilled for both the SMI handler and the measurement agent. While the SMI handler directly benefits from the SMM's protection, HyperSentry provides a perfectly isolated environment for the measurement agent as discussed in Section 4.3. Consequently, any attempt to tamper with the measurement operation will fail because the control-flow remains within the verified integrity measurement code until the measurement output is securely stored in the SMRAM.

The in-context privileged measurement requirement (SR4) is guaranteed by (1) forcing the CPU to return to the VMX root operation, and (2) running the measurement agent at the highest privileged level as a part of the hypervisor. Thus, all attacks that rely on privilege escalation to hide adversaries (e.g., Blue Pill or HVM rootkits) are detectable. In other words, fulfilling this requirement assures that there is no place to hide attack traces within the measured system.

The attestable output requirement (SR5) is fulfilled by signing the measurement output as shown in Section 4.4. Attestation faces two main threats: (1) key leakage, and (2) replay attacks. Key leakage is prevented by locking the SMRAM and flushing the cache upon SMI returns. Replay attacks are prevented by (1) appending a nonce to the signed value to guarantee freshness, and (2) allowing only one output retrieval per measurement session. The latter constraint aims at mitigating the risk of an adversary blocking a measurement request and using a fresh nonce to retrieve an old integrity evidence in the SMRAM.

The only type of attacks that cannot be reliably handled by HyperSentry is transient attacks. This limitation is generic to all integrity verification tools that rely on periodic invocations (e.g., [24], [18]). In a transient attack, the adversary may cause the harm (e.g., stealing data from the guest memory) and then hide its traces. HyperSentry only detects attacks that causes a persistent change to the hypervisor code or data.

The above discussion shows that the security of HyperSentry relies on various hardware components (e.g., IPMI, SMI, TPM, IOMMU, APIC). However, this should not be a concern to the security of the whole framework because most of these components need to be trusted to preserve the correctness of most, if not all, security systems.

5. VERIFYING THE INTEGRITY OF THE XEN HYPERVISOR – A CASE STUDY

In this section, we present a case study of HyperSentry using Xen [1]. That is, we develop a measurement agent and use it in HyperSentry to verify the integrity of Xen at runtime. Note that our main objective is to evaluate the HyperSentry measurement framework; more research is needed to develop a complete solution for measuring the integrity of Xen. Nevertheless, HyperSentry supports any measurement agent embedded as a part of the hypervisor as long as it follows the constraints presented in Section 4.3.

Xen is a bare-metal hypervisor that runs directly on the hardware. All guest VMs run in a less privileged environment on top of Xen. The integrity of Xen, like any other piece of software, is proved through verifying the integrity of both the running code and critical data.

Xen Code Integrity: We calculate the SHA-1 hash of Xen's code based on our knowledge of the virtual memory range where Xen is loaded. The calculated hash, which is a part of our integrity measurement output, is verified by the remote verifier. However, this step does not guarantee that execution will not jump to unmeasured code. To solve this problem, we adopt the approach previously presented by Petroni and Hicks [18] to verify persistent control flow integrity. Similar to their work, we verify the correctness of all possible jump locations that form entry points to the hypervisor code. There are three types of dynamic (i.e., determined in runtime) control-flow transfers: (1) hardware control-flow transfers, (2) indirect control-flow transfer (i.e., function pointers), and (3) function call returns.

To assure hardware control-flow integrity, we verify (1) the IDT, (2) segment descriptors, (3) VM exit handlers, and (4) all MSRs that cause execution jumps (e.g., `SYSENTER_EIP`).

To verify indirect function calls, we use CodeSurfer [15], a software static analysis tool, to analyze Xen's code. Using CodeSurfer, we developed a program to identify all indirect function call sites within Xen. We identified 781 such calls within Xen. Due to CodeSurfer's limitation, we manually identified the function pointers and the set of legitimate values corresponding to each indirect call site. Since our analysis does not rely on the type information of data variables, it identifies all variables used as function pointers regardless of their type definition. Thus, our analysis avoid the limitation of the approach by Petroni and Hicks [18], which can be misled by ambiguous type definitions (e.g., `void*`).

Finally, verifying function call returns is not a part of our prototype, since such an attack does not cause persistent integrity violation. As discussed in Section 3, HyperSentry can only discover persistent changes in the hypervisor. More research is needed to address such integrity violations.

Xen Guest Memory Isolation Integrity – An Attempt on Data Integrity: Although data integrity is an essential part of the integrity of any system, no existing research provides, or even gets close to providing, a complete solution to this problem. In this case study, we tackle a subset of this problem. Specifically, Xen has to guarantee the isolation between the physical memory provided to guest VMs. In this case study, we use our measurement agent to verify if Xen indeed achieves this guarantee.

Xen's memory isolation mechanism mainly relies on the integrity of dynamic hypervisor data. Memory pages that belong to each guest VM are stored in a list called `page_list`. Each instance of this list is a member of another list called `domain_list`, which represents the running guest VMs (i.e, domains). Xen directly relies on the correctness of these two lists to provide memory isolation. For instance, Xen's code assumes that these lists are securely initiated so that each memory page exists in only one page list at a time. Thus, if a memory page concurrently exists in two page lists, legitimate Xen code will violate memory isolation by concurrently mapping the same memory page to two different guest VMs.

To verify guest memory isolation, our measurement agent checks if all instances of `page_list` are mutually exclusive. We use a bitmap where each bit represents one 4KB memory page. The measurement agent parses the page list of each VM and sets a corresponding bit in the bitmap. Any duplicated usage of a memory page will be detected if we try to set a certain bit twice.

Besides page lists, there is another data structure that can affect guest memory isolation, which is the actual guest page tables. If an entry in a guest VM's page table points to a memory page that belongs to another guest VM, then the memory isolation guarantee is violated. To verify the integrity of guest page tables, our measurement agent verifies that the page tables of each guest VM only point to memory pages that belong to the same guest VM.

Our experiments show that our technique can assure memory isolation across regular guest VMs. However, Xen's management VM (i.e., Dom0) legitimately maps pages of hardware assisted (HVM) guest VMs to emulate memory mapped I/O. This behavior limits our agent's ability to detect memory isolation violations between Dom0 and other HVM guests. However, this limitation is specific to the technique Xen uses for device emulation. More research is needed to provide a verifiable technique to share guest VM memory pages with Dom0.

6. IMPLEMENTATION AND EXPERIMENTAL EVALUATION

We implemented a HyperSentry prototype on an IBM BladeCenter H chassis with HS21 XM blade servers [6]. Each HS21 XM blade server has two 3GHz Intel Xeon 5450 quad-core processors with Intel-VT support. We use Xen 3.3.1 as the target hypervisor. Dom0 is a 32-bit Fedora 8 Linux with 2 GB of RAM. We also use two HVM guest VMs: Fedora 10 and Ubuntu 8 with 1 GB and 512 MB RAM, respectively.

Our HyperSentry prototype includes two main components: (1) the SMI handler, and (2) the measurement agent. Both components adhere to the design presented in Sections 4 and 5. More technical details associated with our implementation are presented in Appendix A

We perform a set of experiments to evaluate HyperSentry's performance overhead on the target platform. In general, hypervisor integrity measurement should not be a frequent operation. On the other hand, HyperSentry freezes the whole system from the time the measurement is invoked until it finishes. During this time, all interrupts are disabled and no other code is allowed to utilize any of the processing resources. This freeze forces us to pay an extra attention to HyperSentry's performance overhead.

We evaluate two aspects of HyperSentry's performance overhead: (1) HyperSentry's end-to-end execution time , and (2) the performance overhead imposed on the guest system operations if HyperSentry is called periodically.

6.1 End-to-end Execution Time

We measure the execution time using the `RDTSC` instruction that reads the CPU's Time Stamp Counter (TSC). On our experimental platform, the TSC increments at a constant rate regardless of the current CPU frequency or operating mode. We convert cycles to milliseconds based on the TSC speed.

To measure the end-to-end execution time, we invoke the measurement operation by triggering an SMI through overwriting a power management register. This technique allows us to precisely read the TSC before the measurement operation starts. Writing to power management registers is a privileged operation that can only be done by the hypervisor. Thus, this experiment is limited to interrupting the hypervisor rather than guest VMs.

Table 1 shows the experimental results. Both the average and the standard deviation of the execution time are calculated over 15 rounds. The first measurement shows the time needed by the SMI handler to acquire the hypervisor context and to prepare the protected environment needed by the measurement agent. The 2.78 milliseconds average execution time of the first measurement includes two SMI invocations, context switching to the hypervisor, verifying the measurement agent, and modifying the IDT table.

Table 1: End-to-end execution time (in ms)

Operation	Average Time	Standard Deviation
Agent Invocation	2.78	0.0616
Checksum Code	8.82	0.0007
Retrieve Hypervisor Checksum	0.85	0.0010
Verify Data (Dom0 and 1 VM)	18.84	0.0162
Verify Data (Dom0 and 2 VMs)	21.39	0.0288
Finalize Measurement Session	1.15	0.1132
Total (Dom0 and 2 VMs)	35.04	0.1648

We also observe that the measured execution time for the SMI handler has a relatively high standard deviation. This can be attributed to the non-deterministic nature of the system when the SMI is invoked (i.e., the variation of the time needed to send the command to the hypervisor to write to the power management register).

The next two operations are hashing the hypervisor code (around 660 KB in size) and invoking another SMI to securely store the output. The time needed by the two operations are 8.82 and 0.85 milliseconds, respectively. The low standard deviation of these two operations indicates that they run in a highly deterministic environment.

The next operation is to verify the hypervisor data. As this step depends on the number of running VMs, we perform this experiment in two scenarios. We only use a 2GB Fedora 8 Dom0 and 512MB Ubuntu 8 guest VM in the first one, and run an additional 1GB Fedora 10 in the second.

In the first scenario, the data verification needed 18.84 milliseconds. After running a second 1GB Fedora guest VM, the execution time increased by 2.55 milliseconds. We also observed that data verification shows higher standard deviation, which can be attributed to the dynamic nature of the data that changes between measurement sessions.

The final step, which includes storing the measurement output and resuming the operation of halted CPUs, needed 1.15 milliseconds to complete.

6.2 Guest Performance Overhead

Our next experiment is to understand the performance overhead on the guest system operations if HyperSentry is invoked periodically. The main purpose is to calculate the indirect performance overhead, which is imposed by HyperSentry due to the TLB and cache flushing.

In this experiment, we invoke periodical SMIs every 8 and 16 seconds, respectively. The system performance is calculated using UnixBench benchmark [33]. We compare the results of running the benchmark on the Fedora 10 guest operating system and compare it to the normal guest performance without HyperSentry. We chose to evaluate the performance overhead on a guest VM because they represent the overall system performance from the perspective of the end users in a virtualization environment.

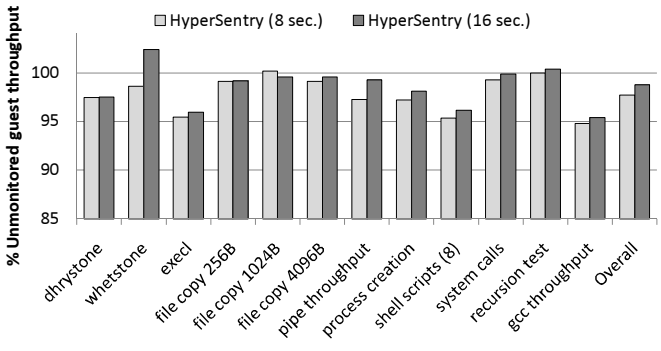

Figure 6: Performance impact on guest VMs when HyperSentry is invoked every 8 and 16 seconds

The results of our experiment are shown in Figure 6. HyperSentry's end-to-end execution time, which is about 35.04 milliseconds, implies 0.4% and 0.2% direct performance overhead if invoked every 8 and 16 seconds, respectively. However, the benchmark output reports an average overhead of 2.4% if HyperSentry is invoked every 8 seconds, and 1.3% if HyperSentry is invoked every 16 seconds. The difference between the anticipated and the actual overhead can be directly attributed to HyperSentry's indirect overhead on the guest system.

As we can see from Figure 6, there is a variation in the overhead imposed on different tests that compose the benchmark. These are mostly due to the difference in the number of HyperSentry interruptions of each test. We also noticed that some of the test results obtained while HyperSentry is interrupting the system are so close to, or even exceeding, those obtained with no HyperSentry interruption. This can be attributed to the very low overhead imposed by HyperSentry, which can be smaller than the normal change in the dynamic state of the system.

7. RELATED WORK

From the architectural perspective, the concept of integrity measurement can be handled by different approaches. The first approach is to measure the static integrity of the system as it is loaded (e.g., IMA [26]). However, this approach cannot protect against runtime attacks (e.g., data attacks) that can change the system state. The second approach is to target the runtime system integrity. However, such systems suffer from the lack of the proper hardware to provide the needed dynamic root of trust and isolation. In the introduction, we discussed the difficulties that challenge Flicker [22], Copilot [24] and HyperGuard [36]. Moreover, relying on software (e.g., Pioneer [29]) suffers from serious security problems due to the absence of a strong root of trust. The third approach is to rely on the multi-level structure provided by virtualization to ensure the runtime integrity of VMs. For example, HIMA [3], REDAS [19], Lares [23] and Patagonix [21] all rely on the hypervisor to monitor the integrity of VMs. In this paper, we move one step further in the trust chain to measure the integrity of the hypervisor.

In terms of the techniques used to verify the integrity of running systems, we benefit from the work done in [18] to detect persistent kernel control flow attacks. Other work in this direction (e.g., [10, 4]) can be used to enhance HyperSentry's measurement agent.

Two recently proposed systems aim to overcome potential threats against commodity hypervisors. The first is seL4 [20], which aims to formally verify certain security properties in the seL4 micro-kernel implementation, such as the absence of certain types of software bugs (e.g., buffer overflows and null pointer dereferences). The second is HyperSafe [35], which aims to extend the hypervisor implementation to enable its self-protection from control-flow hijacking attacks. Our work complements them by exploring a stealthy mechanism that is independent of a running hypervisor but still allows for in-context measurement of various hypervisor integrity properties. These approaches can be integrated to verify and protect commodity hypervisors.

Zimmer and Rasheed filed a patent that describes a method to measure the checksum of the hypervisor at runtime using modification to the CPU's microcode [38]. However, their hypervisor measurement is triggered by events monitored by the hypervisor itself, and thus is prone to the scrubbing attack. Moreover, this technique only supports verifying the checksum of the hypervisor code; it does not provide any detail on how to support other measurement tasks. In con-

trast, HyperSentry is not vulnerable to the scrubbing attack, and can support any integrity measurement tasks.

8. CONCLUSION

In this paper, we presented the development of HyperSentry, an out-of-band in-context measurement framework for hypervisor integrity. The key contribution of HyperSentry is a set of novel techniques to provide an integrity measurement agent with (1) the same contextual information available to the hypervisor, (2) completely protected execution, and (3) attestation to its output. To evaluate HyperSentry, we implemented a prototype for the framework along with an integrity measurement agent for the Xen hypervisor. Our experimental evaluation shows that HyperSentry is a low-overhead practical solution for real world systems. In general, our research demonstrated the feasibility of measuring the integrity of the highest privileged software layer on a system without always keeping the highest privileges.

Our future research will be focused on effective and efficient integrity measurement of popular hypervisors (such as KVM running on Linux) in the HyperSentry framework.

9. REFERENCES

[1] Xen. `http://www.xen.org/`. Accessed in February 2010.

[2] Amazon. Amazon elastic compute cloud (ec2). `aws.amazon.com/ec2`.

[3] A. M. Azab, P. Ning, E. C. Sezer, and X. Zhang. HIMA: A hypervisor-based integrity measurement agent. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC '09)*, pages 193–206, 2009.

[4] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC '08)*, pages 77–86, 2008.

[5] S. Berger, R. Caceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: Virtualizing the trusted platform module. In *Proceedings of the 15th USENIX Security Symposium*, pages 305–320, August 2006.

[6] I. Corporation. IBM BladeCenter products and technology, March 2009.

[7] I. Corporation. Software developer's manual vol. 3: System programming guide, June 2009.

[8] I. Corporation. Integrated management module user guide, February 2010.

[9] A. M. Devices. Amd64 architecture programmer's manual: Volume 2: System programming, September 2007.

[10] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM conference on Computer and Communications Security (CCS '09)*, pages 566–577, 2009.

[11] L. Duflot. Getting into the SMRAM: SMM reloaded. In *Proceedings of the 10th CanSecWest conference*, 2009.

[12] L. Duflot, D. Etiemble, and O. Grumelard. Using CPU system management mode to circumvent operating system security functions. In *Proceedings of the 7th CanSecWest conference*, 2006.

[13] S. Embleton, S. Sparks, and C. Zou. SMM rootkits: a new breed of OS independent malware. In *Proceedings of the 4th international conference on Security and privacy in communication networks*, pages 1–12, August 2008.

[14] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM symposium on Operating systems principles (SOSP '03)*, pages 193–206, 2003.

[15] GrammaTech. `www.grammatech.com/products/codesurfer/`.

[16] Intel. Intel active management technology. `http://www.intel.com/technology/platform-technology/intel-amt/`. Accessed in April, 2010.

[17] Intel, HP, NEC, and Dell. IPMI – intelligent platform management interface specification second generation v2.0. `http://download.intel.com/design/servers/ipmi/IPMIv2_0rev1_0.pdf`, February 2004.

[18] N. L. P. Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and communications security (CCS '07)*, pages 103–115, 2007.

[19] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In *Proceedings of the 39th International Conference on Dependable Systems and Networks (DSN'09)*, 2009.

[20] 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 *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09)*, pages 207–220, 2009.

[21] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *Proceedings of the 17th USENIX Security Symposium*, pages 243–258, 2008.

[22] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: an execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, March/April 2008.

[23] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*, pages 233–247, 2008.

[24] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 13–13, 2004.

[25] J. Rutkowska. Beyond The CPU: Defeating Hardware Based RAM Acquisition Tools. *Blackhat*, February 2007.

[26] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[27] Secunia. Vulnerability report: Vmware esx server 3.x. `http://secunia.com/advisories/product/10757`. Accessed in February 2010.

[28] Secunia. Vulnerability report: Xen 3.x. `http://secunia.com/advisories/product/15863`. Accessed in February 2010.

[29] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the 20th ACM symposium on Operating systems principles (SOSP '05)*, pages 1–16, 2005.

[30] M. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS '09)*, pages 477–487, 2009.

[31] Trusted Computing Group. `https://www.trustedcomputinggroup.org/`.

[32] Trusted Computing Group. TPM specifications version 1.2. `https://www.trustedcomputinggroup.org/downloads/specifications/tpm/tpm`, July 2005.

[33] Tux.Org. `http://www.tux.org/pub/tux/benchmarks/System/unixbench/`.

[34] J. Wang, A. Stavrou, and A. K. Ghosh. HyperCheck: A hardware-assisted integrity monitor. In *Proceedings of the*

13th International Symposium on Recent Advances in Intrusion Detection (RAID'10), September 2010.

[35] Z. Wang and X. Jiang. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, May 2010.

[36] R. Wojtczuk and J. Rutkowska. Xen 0wning trilogy. In *Black Hat conference*, 2008.

[37] R. Wojtczuk and J. Rutkowska. Attacking SMM memory via Intel CPU cache poisoning. Invisible Things Lab, 2009.

[38] V. Zimmer and Y. Rasheed. Hypervisor runtime integrity support. US Patent 20090164770, June 2009.

APPENDIX

A. IMPLEMENTATION DETAILS

In this section, we present more technical details associated with the implementation of HyperSentry's prototype.

Switching to the Hypervisor Context: As discussed in Section 4.2, HyperSentry uses an injected instruction to assure that the CPU switches to the hypervisor context. There are two requirements for this instruction. First, if executed in VMX non-root operation, it should cause an unconditional VM exit (i.e., regardless of the setting of the VM execution controls or the interrupted privilege level). Second, if executed in VMX root operation, it should switch to the highest privilege level so that HyperSentry can run its measurement agent in the hypervisor context. Unfortunately, no single instruction meets both requirements.

In our implementation, we first use `CPUID` because it guarantees a VM exit if it is injected in VMX non-root operation. However, `CPUID` does not guarantee privilege level switch if it is injected in VMX root operation. However, the current privilege level can be detected from the SMM by inspecting the segment selectors. Thus, if HyperSentry detects that the system is still running in an unprivileged level after injecting the first instruction, it can be certain that this system was originally running in VM root operation. It will then inject a second instruction (e.g., `INVD`) to switch the privilege level and run the measurement agent in the correct context.

Pending Interrupts: If an interrupt is received during SMI handling, it will be queued. Thus, execution will jump to its handler right after the SMI handler. As discussed in section 4.2, HyperSentry handles this case successfully. However, if the queued interrupt causes a VM exit directly after returning from the SMM, the first instruction of the VM exit handler, rather than the injected `CPUID`, will be executed before the performance counter overflows. Since the VM exit handler cannot be located, this instruction cannot be *previously known* to HyperSentry. However, executing this instruction does not affect our security because the SMI will be scheduled, due to the performance counter overflow, by the time when this instruction is fetched.

Performance Counter Setting: As discussed in section 4.2, HyperSentry uses the performance counters and the LAPIC to assure that a single instruction is executed by the CPU before returning to the SMM. We configure a performance counter to count instruction fetches that miss the Instruction Table Look-aside Buffer (ITLB). We choose ITLB counting because only one ITLB miss, versus several misses of other types of cache, occurs per instruction fetch. To guarantee this behavior, HyperSentry flushes the TLB, including global pages, before returning from the SMM.

CPU Slow-down in SMM: As mention in section 4.3, the SMM code runs at a lower speed than protected mode code. Our preliminary performance evaluation showed that this slow-down makes hashing the measurement agent in the SMM cause an unacceptable performance overhead on the system. To overcome this problem, HyperSentry first verifies a simple checksum code (∼1.2KB) whose only job is to calculate the hash of the hypervisor from the SMM, and then returns to the hypervisor context to execute this checksum code in the protected mode. The checksum code verifies the hypervisor code along with the remaining part of the measurement agent (∼8.9KB). The checksum code then issues an SMI to safely store its output in the SMRAM before executing the measurement agent. This approach reduces the code to be verified in the SMM, and shortens the total execution time by 85%. Figure 7 shows the whole transitive trust chain from the CRTM up to the hypervisor.

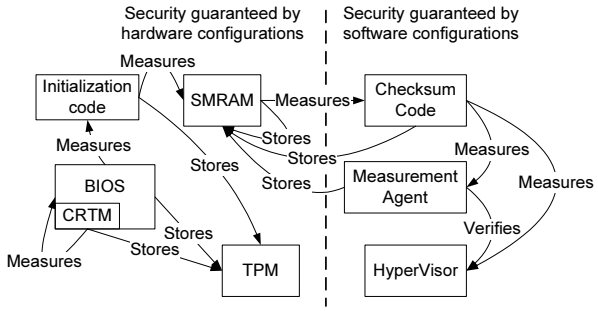

Figure 7: Building the transitive trust chain down from the trusted hardware up to the hypervisor

Measurement Invocation: As mentioned in section 6, the current HyperSentry prototype is implemented using IBM BladeCenter [6]. IBM BladeCenter consists of a chassis that provides shared resources for several thin servers called blades. The chassis is remotely managed by a microcontroller called the Advanced Management Module (AMM), which in turn manages the blades through IPMI and the BMC on each blade. We use the AMM and the IPMI to establish the out-of-band channel that invokes HyperSentry. Remote users connect to the AMM through a SSH connection and send IPMI commands to the BMC to transparently issue an SMI on the target blade.

Current BMC firmware only allows periodic SMI generation on intervals of at least 37.5 seconds. Upon receiving the IPMI command, the first SMI is delayed until one interval elapses. During this delay, the BMC allows the CPU to query about the periodic SMI status. In other words, the hypervisor has a chance to detect if an SMI is scheduled.

To guarantee the stealthiness of the measurement invocation, the BMC's firmware should be updated to either allow immediate SMI generation or to hide the SMI invocation request from the CPU. Our prototype implementation does not include this feature yet. Nevertheless, adding this feature into the BMC would be technically trivial.

Attestation Process: As mentioned in Section 4, the SMI handler should be a part of the server's trusted boot. However, HS21 XM servers are not equipped with a TPM chip. Due to the lack of a TPM, the attestation process is not included in our prototype. A HyperSentry session ends by storing the measurement output in the SMRAM. Nevertheless, it is worth mentioning that static attestation using the TPM, signature key generation and signing are all known techniques that have been implemented previously.