ABSTRACT

AZAB, AHMED MONEEB. New System Security Mechanisms for the Cloud Computing Infrastructure. (Under the direction of Dr. Peng Ning.)

Cloud computing is a new computing paradigm, where computing resources are provided as a service to end users. To cut operating costs, hardware virtualization is used so that a single hardware platform is shared among multiple end users. This new paradigm introduces a complex security threat model, where threats can originate from external attackers, malicious end users or even the cloud infrastructure itself. In this dissertation, we introduce three novel security mechanisms that enhance the trustworthiness of cloud computing infrastructures: HIMA, HyperSentry, and SICE.

HIMA, which stands for Hypervisor-based Integrity Measurement Agent, can be used to measure the integrity of guest VMs that run inside the cloud. This measurement can be further used to attest to the integrity of these guest VMs. Unlike other integrity measurement techniques, HIMA provides two essential capabilities: 1) Strong isolation between the measurement agent and the measurement target, and 2) Time of Check to Time of Use (TOCTTOU) consistency, which guarantees the continuity of the integrity evidence beyond the measurement time.

HyperSentry is a novel framework to enable integrity measurement of running hypervisors (or any other highest privileged software). Unlike existing solutions for verifying privileged software, HyperSentry does not introduce a higher privileged software layer below the hypervisor. Instead, it introduces properly isolated software components that 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.

SICE, which stands for Strongly Isolated Computing Environment, is a novel framework to provide hardware-level isolation and protection for sensitive workloads running on x86 platforms in compute clouds. Unlike existing isolation techniques, SICE does not rely on any software component in the host environment (i.e., an OS or a hypervisor). Instead, the security of the isolated environments is guaranteed by a trusted computing base that only includes the hardware, the BIOS, and a very small System Management Mode (SMM) software foundation of about 300 lines of code.
DEDICATION

Dedicated to my wonderful wife, Heba, for so many reasons
ACKNOWLEDGEMENTS

I would like to gratefully and sincerely thank my Ph.D. advisor, Dr. Peng Ning, for his priceless guidance, support, and encouragement. Dr. Ning taught me everything I know about conducting academic research, from identifying research problems all the way to carefully writing and presenting research result. Throughout my Ph.D studies, he provided all the needed resources to support my research. Without Dr. Ning’s commitment to research, eye for detail, and high standards, none of the research in this dissertation could have been done.

I would like to appreciatively thank Dr. Xiaolan (Catherine) Zhang from IBM T.J. Watson Research Center for her invaluable collaboration. Dr. Zhang co-authored all the papers I published during my Ph.D studies. She brought an extremely valuable experience in both academic and industrial research. Both her experience and effort played a great role in enhancing the research in this dissertation.

I would like to thank Dr. Xuxian Jiang, Dr. Chongkyung Kil, Emre C. Sezer, Nathan C. Skalsky, and Zhi Wang who helped shaping my research by being my co-authors. I would also like to thank my committee members, alphabetically, Dr. Xuxian Jiang, Dr. Douglas S. Reeves, and Dr. Yan Solihin for their valuable feedback.

I would like to thank the organizations that funded my research. This research 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 contents of this dissertation do not necessarily reflect the position or the policies of the U.S. Government or IBM corporation.

Last but not least, I would like to express gratitude to my family and friends for their endless support. In particular, my wife, Heba Aly, helped me a lot by proof reading my writing and dealing with the long hours I spent working on my research. My parents, brothers and in-laws provided me with continuous encouragement.
# TABLE OF CONTENTS

List of Tables ...................................................... vii
List of Figures ...................................................... viii

Chapter 1 Introduction ............................................. 1
  1.1 Cloud Computing Security Challenges ......................... 1
  1.2 A Vision for a Trustworthy Cloud Computing ................. 2
  1.3 Building Trust in Cloud Computing Guest VMs ............... 3
  1.4 Verifying the Integrity of the hypervisor ..................... 4
  1.5 Strong Isolation for Security-Sensitive Workloads .......... 5
  1.6 Summary of Contributions .................................. 6

Chapter 2 Background Information ............................... 7
  2.1 Hardware Virtualization .................................... 7
  2.2 Trusted Computing ......................................... 8
  2.3 The System Management Mode (SMM) ......................... 9
  2.4 Intelligent Platform Management Interface (IPMI) .......... 10

Chapter 3 Review of Related Work ............................... 12
  3.1 Integrity Measurement of Guest VMs ......................... 12
  3.2 Integrity Measurement of Privileged Software ............... 14
  3.3 Isolated Execution Environments ............................ 16

Chapter 4 HIMA: A Hypervisor-Based Integrity Measurement Agent .... 19
  4.1 Introduction ................................................. 19
  4.2 Assumptions and Thread Model ............................... 22
  4.3 Overview of HIMA ........................................... 22
  4.4 Active Monitoring of Guest VMs .............................. 23
    4.4.1 Introspecting Event Context and Impact ............... 24
    4.4.2 Semantic-Aware Measurement .......................... 26
  4.5 Guest Memory Protection ................................... 28
    4.5.1 Protecting Guest Memory Using Page Access Permissions .. 29
    4.5.2 Identifying Mapping of Monitored Pages ............... 29
    4.5.3 Handling Remapping of Measured Executable Pages .... 30
    4.5.4 Extending Memory Protection to Physical Frames ....... 32
    4.5.5 Writable-eXecutable (WX) Memory Regions ............. 32
  4.6 Security Analysis .......................................... 32
    4.6.1 Trust in Guest VMs .................................... 33
    4.6.2 Possible Threats and Defenses ......................... 33
  4.7 Experimental Evaluation ................................... 34
    4.7.1 Evaluation Methodology ............................... 35
    4.7.2 Result of Micro-benchmark Evaluation .................. 36
Chapter 5 HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity

5.1 Introduction
  5.1.1 Challenges
  5.1.2 Introducing HyperSentry

5.2 Assumptions, Threat Model, and Security Requirements

5.3 The HyperSentry Framework
  5.3.1 Stealthy Invocation of HyperSentry
  5.3.2 Acquiring Measurement Context
  5.3.3 In-context Integrity Measurement
  5.3.4 Attesting to the Measurement Output
  5.3.5 Security of the HyperSentry Framework

5.4 Verifying the Integrity of Xen – A Case Study

5.5 Implementation and Experimental Evaluation
  5.5.1 End-to-end Execution Time
  5.5.2 Guest Performance Overhead

Chapter 6 SICE: A Hardware-Level Strongly Isolated Computing Environment for Commodity x86 Multi-core Platforms

6.1 Introduction
  6.1.1 Introducing SICE
  6.1.2 SICE Overview
  6.1.3 Summary of Contributions

6.2 Threat Model and Assumptions

6.3 SICE Design
  6.3.1 Time-sharing Mode
  6.3.2 Multi-core Mode
  6.3.3 Attestation and Secure Communication
  6.3.4 Security Analysis

6.4 SICE Prototype: An Isolated VM
  6.4.1 Preparing and Initializing the VM
  6.4.2 Handling VM Exits
  6.4.3 Communication with the Legacy Host
  6.4.4 Using Hardware Peripherals
  6.4.5 Attestation

6.5 Experimental Evaluation
  6.5.1 SICE Execution Time
  6.5.2 Isolated Environment Performance

6.6 Portability to Intel Platforms
Chapter 7 Conclusion and Future Work ........................................ 87
  7.1 Conclusion ................................................................. 87
  7.2 Future Work ............................................................... 88

References ................................................................. 90
LIST OF TABLES

Table 5.1 HyperSentry End-to-end execution time (in ms) . . . . . . . . . . . . . . . 57
Table 6.1 SICE execution time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>HIMA architecture</td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>Example of intercepting concurrent system calls</td>
<td>26</td>
</tr>
<tr>
<td>4.3</td>
<td>Flow diagram of the proposed guest memory protection mechanism</td>
<td>28</td>
</tr>
<tr>
<td>4.4</td>
<td>Data structure for measured hash lists and guest processes</td>
<td>30</td>
</tr>
<tr>
<td>4.5</td>
<td>Computational overhead with UnixBench</td>
<td>36</td>
</tr>
<tr>
<td>4.6</td>
<td>Computational overhead using application benchmark</td>
<td>37</td>
</tr>
<tr>
<td>5.1</td>
<td>HyperSentry’s architecture</td>
<td>44</td>
</tr>
<tr>
<td>5.2</td>
<td>Building trust in SMI handler</td>
<td>44</td>
</tr>
<tr>
<td>5.3</td>
<td>Hardware components involved in remote SMI invocation</td>
<td>46</td>
</tr>
<tr>
<td>5.4</td>
<td>HyperSentry’s SMI handling workflow</td>
<td>48</td>
</tr>
<tr>
<td>5.5</td>
<td>HyperSentry’s attestation process</td>
<td>52</td>
</tr>
<tr>
<td>5.6</td>
<td>Performance impact on guest VMs when HyperSentry is invoked</td>
<td>58</td>
</tr>
<tr>
<td>6.1</td>
<td>SICE Architecture</td>
<td>67</td>
</tr>
<tr>
<td>6.2</td>
<td>Entering SICE from the legacy host</td>
<td>68</td>
</tr>
<tr>
<td>6.3</td>
<td>General multi-core processors architecture</td>
<td>71</td>
</tr>
<tr>
<td>6.4</td>
<td>Double view of memory from different processor cores</td>
<td>73</td>
</tr>
<tr>
<td>6.5</td>
<td>SICE Prototype</td>
<td>78</td>
</tr>
<tr>
<td>6.6</td>
<td>Performance overhead of an isolated VM</td>
<td>84</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Cloud computing is an emerging technology in which computing resources are provided as a service to remote end users. The provided computing resources can be in the form of software applications, computing platforms, and complete infrastructures. The idea behind cloud computing is to deliver these various types of computing resources as services that do not require end users to manage or configure physical systems. At the same time, the cloud computing service providers leverage economies of scale to reduce the cost of the provided services.

Cloud computing service providers, particularly those who provide complete platforms or infrastructures as a service, usually utilize hardware virtualization to further reduce their operating costs. Hardware virtualization introduces the concept of platform multi-tenancy, where a single physical platform is shared among several end users. Each end user is provided with a Virtual Machine (VM) that acts like a real computer with its own Operating System (OS). A privileged software layer, called the hypervisor or Virtual Machine Monitor (VMM), is responsible for allowing these VMs to concurrently share the hardware resources of the physical platform. In this dissertation, we mainly focus on this particular type of cloud computing.

1.1 Cloud Computing Security Challenges

The adoption of virtual cloud computing introduces a complex security threat model to both service providers and end users. A part of this threat model can be represented by traditional security threats, in which an external attacker tries to exploit security vulnerabilities in the software running inside the cloud. This software can be either managed by the cloud service provider or end users. Unfortunately, commodity software systems are not completely secure. As an evidence, software vulnerability databases (e.g., The National Vulnerabilities Database [26]) add thousands of software vulnerability reports every year. This clearly indicates the size of the threat introduced by attackers to any software system (including those
that run inside the cloud). We denote this type of attacks as *external threats*.

The other part of this threat model arises from the architecture of cloud computing, particularly the multi-tenancy of physical hardware platforms by guest VMs that belong to mutually distrusted end users. Although hardware virtualization provides a degree of isolation between guest VMs that share a physical platform, recent attacks [41, 59, 101] and vulnerability reports [1, 2, 4, 82, 83] clearly show that this protection is far from perfect. Moreover, multi-tenancy also exposes side-channels that can be used to leak confidential information [77] even without compromising the virtualization layer.

Thus, a guest VM that runs inside the cloud faces two additional types of threats: *Guest-to-guest threats* that result from sharing the same physical platform with other potentially malicious guest VMs and *cloud-to-guest threats* that arise from the fact that the cloud service provider is granted full access to all running guest VMs. Moreover, the service provider faces *guest-to-cloud threats* that can be represented by attacks originated from a malicious guest VM. Generally, a threat that is originated from the cloud service provider or a guest VM can either result from an initially malicious end user or administrator, or from a legitimate software component that has been compromised by an external attacker.

To summarize, a typical cloud computing environment is subject to four different types of threats: external, guest-to-guest, guest-to-cloud, and cloud-to-guest threats. This complex threat model raises new and fundamental problems of trust and security, which make end users concerned about the loss of direct control over their computing workloads by moving them “to the cloud” [38].

### 1.2 A Vision for a Trustworthy Cloud Computing

This dissertation envisions providing a comprehensive set of security mechanisms to reduce the security threats that face both cloud computing service providers and end users. Achieving this vision would allow end users to utilize the full benefits of cloud computing, while having a guarantee that their computing workloads would preserve their integrity, confidentiality and availability. It would also help the cloud computing service providers to affirm that their resources cannot be misused by malicious or compromised guest VMs.

The complex nature of the cloud computing threat model, discussed in Section 1.1, introduces a set of security challenges that need to be adequately addressed. Several existing security technologies can be incorporated to address these challenges. These technologies include: Intrusion detection, secure operating systems, isolation, malware detection, and access-control mechanisms. Although these existing technologies would provide security to a certain extent, they are still far from perfect. As an evidence, recent vulnerability databases (e.g., The National Vulnerabilities Database [26]) and threat reports (e.g., McAfee Threats Report for Q2
2011 [63]) are adding thousands of new security threats and vulnerabilities every year.

To adequately address the cloud computing security challenges, we focus on a more fundamental problem, which is how to prove to a remote end user that the security measures of the cloud infrastructure are effective and have not been infiltrated. In other words, end user should be able to confirm that computing workload running inside the cloud will consistently behave as expected. Moreover, end users should be able to confirm that the cloud infrastructure will preserve the confidentiality and the integrity of their computing workloads.

We developed a set of novel security mechanisms to measure and verify the integrity of different parts of the cloud computing infrastructure. In the context of this dissertation, measuring the integrity of a system means constructing a cryptographic hash that represents all the software programs and configurations of this system. Thus, the result of this integrity measurement process can be further used to attest that only authorized code runs on a target system. We also harness the recent advances in trusted computing, introduced and developed by the Trusted Computing Group (TCG) [94], to construct a cryptographic proof of the authenticity of the integrity measurement result.

1.3 Building Trust in Cloud Computing Guest VMs

Initially, we focus on verifying the integrity of cloud computing workloads, which are represented by guest VMs that run inside the cloud. The integrity of guest VMs is a major issue for both end users who own this guest VMs and cloud service providers who host it on their virtualized platforms. End users, particularly those who use their VMs to conduct critical services (e.g., hosting an online business), need to affirm that their VMs do not run any malicious software that may tamper with their operations or expose their private information. On the other hand, cloud service providers need to affirm that their hardware resources are not used for a malicious behavior (e.g., spreading malware).

In this regard, we develop HIMA, which stands for Hypervisor-based Integrity Measurement Agent. HIMA uses novel integrity measurement mechanisms that utilize the hardware virtualization architecture widely used by cloud computing. HIMA is a hypervisor-based agent that performs a comprehensive “out-of-the-box” measurement of guest VMs, including both their kernels and user programs.

HIMA provides two capabilities that are essential to the integrity measurement of guest VMs. The first capability is strong isolation between the measurement agent and the target guest VM. In this context, strong isolation means that the measurement technique does not rely on any component or hook in the guest VM, which can be subverted by an attacker. The second capability is Time of Check to Time of Use (TOCTTOU) consistency, which guarantees the continuity of the integrity evidence beyond the measurement time. This capability prevents
attackers from changing a program after it is measured and before it executes. By the time HIMA was developed and published [9], no other then-existing approach had, or could have been easily modified to have both capabilities.

In Chapter 4, we discuss HIMA in detail. We first present HIMA’s architecture and design. Afterward, we present the unique techniques used to achieve the strong isolation and the TOCTTOU consistency that distinguish HIMA from other guest VM monitoring techniques. Finally, we introduce the implementation and experimental evaluation of a HIMA prototype developed using the Xen hypervisor.

1.4 Verifying the Integrity of the hypervisor

The relatively small code-base of hypervisors and their limited interaction with guest VMs made many security experts assume that hypervisors are both well protected and easily verifiable. For instance, many researchers rely on hypervisors to measure the integrity of guest VMs (e.g., [35], [73], [65], [51], [89], [84]).

Unfortunately, hypervisors did not turn out to be completely secure as they were assumed to be. In fact, the importance of hypervisors to both the functionality and integrity of guest VMs makes hypervisors a valuable target for attacks that target the cloud. As mentioned in Section 1.1, recent attacks [41, 59, 101] and vulnerability reports [1, 2, 4, 82, 83] show that hypervisors are subject to security exploits and can be compromised. 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 the hypervisor integrity at runtime.

To achieve our vision toward a more secure cloud, the second natural step is to verify the integrity of the hypervisor at runtime. Cloud service providers need to verify the integrity of all their hypervisors periodically to make sure that this centerpiece of any virtualized platform is not compromised. At the same time, cloud end users need a proof that the hypervisor that will run their workloads is not compromised.

In this regard, we introduce HyperSentry [10], a framework that supports stealthy in-context integrity measurement of a running hypervisor (or any other highest privileged software). 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 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’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. A key contribution of HyperSentry is the set of novel techniques that 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 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. In Chapter 5, we discuss HyperSentry in detail.

1.5 Strong Isolation for Security-Sensitive Workloads

HIMA and HyperSentry combined cannot solve all the security problems that face cloud computing. HIMA’s active monitoring, which can detect all attacks that target the code sections of the monitored guest VM, cannot detect attacks that target dynamic data. The same limitation is shared with all systems that rely on code measurement for integrity verifications (e.g., [65], [81]). Unfortunately, attacks that target dynamic data are a credible threat. For example, it has been shown that an attacker can compromise an application (or even a system) by simply modifying critical data structures without changing the control flow of the applications [20]. Another type of data attacks, the return oriented attack [17, 19, 43, 87], enables an attacker to compromise the control flow of a program without modifying its code sections.

HyperSentry also has a limitation. Despite the robustness of its integrity measurement framework, it can only be invoked by an external verifier and it cannot provide event-driven monitoring of the hypervisor. Thus, HyperSentry cannot reliably handle transient attacks. In a transient attack, the adversary may cause the harm (e.g., stealing data from the guest memory) and then hide its traces before the following integrity measurement session. This limitation is generic to all integrity verification tools that rely on periodic invocations (e.g., [74], [55], [98], [101]). To the extend of our knowledge, there is no solution that provides an event-driven monitoring of the highest privileged software.

These limitations, which face most of the state-of-the-art security mechanisms, directed us to search for another fundamental solution to provide an elevated protection for security-sensitive workloads that run inside the cloud. Our solution is a framework that is called SICE [11], which stands for Strongly Isolated Computing Environment. SICE is a novel framework to provide hardware-level isolation and protection for sensitive workloads running on x86 platforms. To provide strong isolation, SICE minimizes the code base that is granted full access to the memory...
of running workloads. Thus, we can minimize the exposure to security vulnerabilities that can threaten the isolation provided to these workloads. Moreover, SICE provides this code base with enhanced security protection and the ability to attest to its integrity.

SICE differs from the previous research efforts in providing verifiable and isolated execution. SICE achieves its isolation without relying on any software component controlled by the host environment or shared among mutually distrusted workloads. Instead, SICE harnesses existing hardware and firmware support to prevent untrusted code from accessing the memory assigned to the isolated environment or manipulating its execution.

The SICE philosophy is based on using the isolated environment to run security sensitive workloads provided by the cloud end users. On the other hand, the untrusted environment is used for less sensitive operations, like managing peripheral devices. In a typical execution scenario, a communication channel between the two environments is used so that the host can provide different services (e.g., networking) to the isolated environment. In other words, SICE allows cloud computing infrastructures to keep their basic architecture and services, while eliminating the currently required trust in the host system.

We have implemented a SICE prototype using an AMD x86 hardware platform. Our prototype demonstrates that, subject to a careful security review of the BIOS software and the SMM hardware implementation, current hardware architecture already provides abstractions that can support building strong isolation mechanisms using a very small software foundation of about 300 lines of code. In Chapter 6, we present SICE in detail.

1.6 Summary of Contributions

In the course of our research for new security mechanisms that ensure the trustworthiness of virtual cloud computing, we solved several research problems and technical challenges. The provided solutions make the following high-level contributions:

- We use the hypervisor to achieve strong isolation between the measurement agent and the measurement target, and ensure TOCTTOU consistency for the integrity measurement of user programs running inside guest VMs.

- 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 provide a complete and feasible solution to share hardware resources with an isolated execution environment that does not rely on any host software. Instead, it relies on a TCB that is around an order of magnitude smaller than the state-of-the-art systems.
Chapter 2

Background Information

In this chapter, we give background information on some of the basic technologies used to build our security mechanisms. These technologies include: Hardware virtualization, trusted computing, System Management Mode (SMM), and Intelligent Platform Management Interface (IPMI).

Both hardware virtualization and trusted computing were developed to achieve some desirable security properties. For example, hardware virtualization provides strong isolation, while trusted computing provides remote attestation. We utilize these security properties to achieve our research objectives.

On the other hand, SMM and IPMI were originally designed for system management functions. However, both technologies are represented by system components that are well isolated from the running host (e.g., OS, hypervisor). In Chapters 5 and 6, we utilize this isolation to protect against attacks that break the virtualization layer and compromise the running OS or hypervisor.

2.1 Hardware Virtualization

As mentioned in Chapter 1, hardware virtualization, or simply virtualization, is a key enabling technology for cloud computing. It is a solution to increase hardware utilization by sharing physical platforms among multiple guest VMs. Each guest VM is presented with an abstract instance of the physical machine that is sufficient to boot and run complete OS and applications. These guest VMs act as generic execution contexts or containers for cloud end users.

A software layer called the hypervisor, also known as the virtual machine monitor, is responsible for managing and configuring guest VMs. There are two different types of hypervisors: bare-metal and hosted hypervisors.

Bare-metal (or type 1) hypervisors run directly on the highest privileged level of the hard-
ware platform. Guest VMs thus run on a lower privileged level that is controlled by the hypervisor. Examples of bare-metal hypervisors include VMware ESX server [3] and Xen [5]. Hosted (or type 2) hypervisors are applications that run within a traditional OS, such as Windows or Linux. Examples of hosted hypervisors include KVM [66] and VirtualBox [100]. In both cases, hypervisors control all critical hardware operations (e.g., memory management unit, exception handling) of guest VMs. Hence, the integrity of the hypervisor is a critical aspect of the integrity of all hosted guest VMs.

Using hypervisors to virtualize hardware resources is not a new idea. In fact, the first hypervisor was introduced in the 1960s for IBM’s Z servers [25]. However, modern hardware platforms offer advanced support that yield significant improvements in the efficiency and transparency of hardware virtualization. This support, which is called hardware-assisted virtualization, was introduced by both Intel [47] and AMD [6].

Beside enhancing resource utilization through hardware sharing, virtualization achieved desirable security properties, particularly the isolation between guest VMs and the virtualization layer (i.e., the hypervisor). Many researchers leverage virtualization to provide security services that include intrusion detection [36,51,60,73], intrusion analysis [31,50], kernel protection [76,84] and integrity measurement [9, 65]. Using virtualization to provide security services will be further discussed in Chapter 3.

2.2 Trusted Computing

The Trusted Computing Group (TCG) [94] defines specifications for computing environments that aim to provide a foundation of trust for software running on computing platforms. The core of these specifications is the Trusted Platform Module (TPM) [95], which is a secure co-processor that serves as a hardware root of trust that cannot be tampered by malicious software.

The TPM offers various security facilities that include: generation of cryptographic keys, pseudo-random number generation, sealed storage, and remote attestation. In this dissertation, our main focus is remote attestation, which is proving the integrity of the running OS and applications to remote users.

The first step of remote attestation is integrity measurement. Previous attempts to measure the system integrity using software components (e.g., [34,70,85,86,88]) suffer from the lack of the proper hardware to provide the needed root of trust and isolation. The TPM was introduced to overcome this challenge. The TPM contains a set of Platform Configuration Registers (PCRs) that act as protected storage for hash values that represent the system’s integrity measurement results. A subset of these registers, which are called static PCRs, can not be reset during runtime. Instead, they can only be extended; this means that the new measurement value is
hashed together with the previous value in the PCR. Thus, PCRs would always contain the
history of the integrity measurements until the system is rebooted.

The TPM also plays a significant role in the remote attestation protocol. In a typical remote
attestation scenario, a remote client requests a system to prove its integrity. The client and
the system are called the challenger and attestor, respectively. The attestation process starts
when the challenger requests a proof of integrity from the attestor. The challenger includes a
nonce in its request to guarantee the freshness of the reply. The TPM on the attestor generates
an Attestation Identity Key (AIK) and uses it to cryptographically sign current values of
selected PCR’s along with the received nonce. As mentioned above, the PCR values should be
storing the integrity measurement of the attestor systems. Thus, the challenger verifies both the
authenticity and the freshness of the integrity measurement results using the TPM signature.
Finally, Goldman et al. [39] proposed to extend the PCR with a measurement that represents
the static property of the secure connection endpoint (e.g., SSL public key). Using this new
technique, the remote challenger can guarantee that the TPM signature was generated using
the same machine at the endpoint of the tunnel.

Using the TPM, the TCG introduced “trusted booting” [94] (a.k.a static root of trust
management (SRTM)), where the root of trust starts at a read-only memory called the core
root of trust for measurement (CRTM) at boot time. The CRTM computes the SHA1 hash
(i.e. measurement) of the contents of the BIOS and extends the hash value in one of the TPM’s
static PCRs. The BIOS in turn measures the OS boot loader. This pattern is repeated until the
OS is bootstrapped. As discussed earlier, an adversary cannot reset values of the static PCRs
unless the whole system is rebooted, which allows only the CRTM to start the trust chain.

Beside the SRTM, the TCG also introduces the Dynamic root of trust management (DRTM)
to start the trust chain during system runtime. The DRTM consequently eliminates the BIOS,
including any other system start-up code, from the trust chain. Both Intel [46] and AMD [6]
provide support of the DRTM on their recent hardware platforms.

2.3 The System Management Mode (SMM)

SMM is an x86 operating mode designed to handle system management functions. The CPU
enters the SMM upon receiving a System Management Interrupt (SMI), triggered by either
software or hardware events.

SMM is an independent and protected environment that cannot be tampered by other
software running on the system. When an SMI is invoked, the hardware saves the current
CPU state and switches the context to the SMM. After the SMI finishes, it executes the RSM
instruction to resume the interrupted CPU operation. Inside the SMM, paging is disabled
but the lowest 4GB of physical memory and all I/O ports are accessible. The SMM privilege
is equivalent to the highest privilege level of the protected mode (where the hypervisor runs). Moreover, all interrupts, including the non-maskable ones, are disabled upon entering the SMM. Thus, even the hypervisor cannot interfere with the SMI handler after it is invoked.

Moreover, SMM’s code is stored in a designated, lockable memory called SMRAM. Locking SMRAM prevents all access to it except from within the SMM. As long as the SMRAM is locked, an adversary controlling the system software can neither manipulate the SMI handler nor its running environment.

This escalated privilege level along with its isolation made SMM attractive to malicious users [30]. To avoid its misuse, all BIOS manufactures lock the SMRAM before the OS boots. In our 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. There are two types of SMRAM: the ASeg, which is located at a fixed low address, and the TSeg which is located at a variable base and has a variable size. In our research, we use TSeg due to the flexibility it offers.

Defining the SMRAM location and range differs slightly between AMD and Intel architectures. On Intel platforms, the SMRAM is locked through the memory controller’s D_LCK bit. Once this bit is set, the defined SMRAM range cannot be unlocked or modified until the system is rebooted. Moreover, setting this bit locks the SMRAM range for all CPU cores that use the same memory controller.

On the other hand, the AMD architecture defines the TSeg memory range using two Model Specific Registers (MSRs) that are local to each processor core. The first is SMM_Addr, which specifies the SMRAM base address, while the second is SMM_Mask, which specifies its range. Since MSRs are local to each processor core, all cores have to set their MSRs to provide complete protection of the SMRAM. The SMMLOCK bit in the HWCR MSR can be set to prevent changing the TSeg range. Moreover, the AMD architecture uses a password mechanism to protect the modification of the SMMLOCK bit. When the SMRAM is unlocked, memory writes to the SMM_KEY MSR set the 64-bit password. When the SMRAM is locked, writing the correct password to the same MSR clears the SMMLOCK bit.

2.4 Intelligent Platform Management Interface (IPMI)

IPMI is a server-oriented platform management interface directly implemented in hardware and firmware [45]. 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 micro-controller 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.
In our research, we use IBM BladeCenter [23], which supports IPMI, as our experimental hardware platform. IBM BladeCenter consists of a chassis that provides shared resources for several thin servers called blades. The chassis supplies the blade servers with power, cooling, and networking infrastructure. However, each server is an independent server with its own processors, memory, storage, network controllers, OS, and applications. The chassis is remotely managed by a micro-controller in the chassis called the Advanced Management Module (AMM), which in turn manages the blades through IPMI and the BMC on each blade. A remote user can connect to the AMM through a SSH connection and send IPMI commands to the BMC to transparently issue an SMI on the target blade.

In Chapter 5, we discuss how HyperSentry uses the AMM and the IPMI to establish the out-of-band channel that invokes the integrity measurement of the hypervisor.
Chapter 3

Review of Related Work

In this Chapter, we conduct a literature survey of all related work that target the three security problems identified in Chapter 1. For each problem, we identify the limitations of previous work to motivate the need of fundamental solutions that overcome these limitations.

3.1 Integrity Measurement of Guest VMs

Cloud service providers should be able to attest to the integrity of virtual machines to remote users. As discussed in Section 2.2, remote attestation is a process in which a computer system proves its integrity to a remote party. It plays an important role in improving trustworthiness in distributed computing environments. Essential to remote attestation is the integrity measurement of the attested system; the integrity information allows the remote party to verify the configuration and the current state of the attesting platform (e.g., hardware and software stack), thereby to determine its trustworthiness.

There were several attempts to address the integrity measurement problem. As mentioned in Section 2.2, Sailer et al. introduced the Integrity Measurement Architecture (IMA), which extends the TCG’s trusted booting by measuring all loaded OS components and user programs throughout the lifetime of the system. There are also several research work that proposed extending IMA to enforce the integrity measurement on dynamic system components by either using security policies (e.g., PRIMA [49], UCLinux [61]) or measuring the integrity of dynamic system properties (e.g., Redas [57]). To be able to port all these techniques to guest VM that run inside the cloud, Berger et al. introduced vTPM [13], which virtualizes the hardware TPM. A common property of these techniques is that they all rely on monitoring and measuring components that execute inside the kernel of the same system to be measured.

Unfortunately, these approaches, though providing a certain degree of security, are vulnerable to runtime attacks that target the kernel. These attacks may exploit common OS
vulnerabilities to either manipulate the kernel’s integrity measurement code or critical measurement data. In general, attacks targeting kernel data cannot be easily detected by known kernel code protection methods (e.g., Secvisor [84], NICKLE [76]). A fundamental limitation of these approaches is the lack of strong isolation between the measurement agent and the measurement target.

To overcome this limitation, researchers leveraged virtualization technology to enhance integrity measurement without trusting the OS and application programs of the measurement target. This approach is often referred to as “out-of-the-box” monitoring of guest VMs. Terra [35] uses the hypervisor to measure the integrity of VMs at their boot time. Other research in the same direction (e.g., Livewire [37], VMwatcher [51], Antfarm [52], and XenAccess [72]) proved good results in using the hypervisor to detect malware, or other integrity violations, that run inside guest VMs.

All techniques that rely on the hypervisor to provide security services to guest VMs face one basic challenge: The semantic gap between the view of the guest VM from inside (i.e., as seen by the OS or applications) and the view from outside (i.e., the low-level raw data seen by the hypervisor). For example, instead of seeing kernel data structures that describe running processes and loaded kernel modules, the hypervisor can only see memory pages and CPU registers.

To overcome the semantic-gap, researchers use a technique called memory introspection, where high-level semantic objects (e.g., process control blocks, virtual memory area descriptors) are constructed from the low-level memory viewable to the hypervisor. Guest VM memory introspection has been successfully used in many applications including intrusion detection [36, 51–54,60,72], verification of access control policies [103], and general system monitoring [50,75].

All these systems have two common limitations: 1) The lack of event driven (active) monitoring of the events that occur inside the guest VM, and 2) the lack of runtime memory protection provided for these VMs, as such it is subject to TOCTTOU attacks where the running guest code can differ from the initial measured code. These limitations hurdle the use of these technique to do effective integrity measurement. Some systems tried to overcome the lack of guest VM active monitoring by placing monitoring programs [16] or hooks [73] inside the measurement target and providing memory protection for these hooks. However, similar to IMA, both techniques rely on components inside the guest VMs, thus prone to guest system vulnerabilities.

Patagonix [65] is closest to achieving the goals of hypervisor-based integrity measurement. It utilizes the hypervisor’s control over the Memory Management Unit (MMU) to provide guest memory protection and identify running programs based on the hashes of individual memory pages. However, it lacks the necessary semantic knowledge about the guest VM user processes, which impacts the effectiveness of the integrity measurement. First, Patagonix cannot handle
the on-demand loading of running programs to measure them in their entirety. Second, it is still possible that an attacker can craft malicious programs by reusing or duplicating code pages of legitimate programs, thus bypassing integrity measurement. Finally, Patagonix relies on a large database of trusted hashes of individual pages, which has a negative effect on its scalability. Unfortunately, It is complicated to address these issues.

**Limitations:** To summarize the previous discussion, traditional integrity measurement techniques lack the strong isolation from the monitored target. Although virtualization directly provides the required isolation, current techniques failed to keep the strong isolation guarantees while providing both active monitoring of guest events and TOCTTOU consistency of the measurement results. In Chapter 4, we discuss how HIMA achieves both properties.

### 3.2 Integrity Measurement of Privileged Software

As discussed in Section 2.1, bare-metal hypervisors monopolize the highest privileged software layer of the system, while hosted hypervisors are part of traditional operating systems. In both cases, verifying the integrity of the hypervisor requires verifying the integrity of the highest privileged software layer.

The integrity verification of the highest privileged software 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 integrity for this newly introduced software layer?

To measure the runtime integrity of privileged software, few recent attempts [68,74,98,101,105] 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 completely successful.

Copilot [74] 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 [78] 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 [101] and HyperCheck [98] 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 [48].

Flicker [68, 69] 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.

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 [105]. 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.

Beside integrity measurement, two recently proposed systems aim to overcome potential threats against commodity hypervisors by enforcing their protections. The first is seL4 [58], 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 [99], which aims to extend the hypervisor implementation to enable its self-protection from control-flow hijacking attacks.

**Limitations:** The previous discussion shows that previous approaches to measure the integrity of the highest privileged software lacks two necessary properties. The first is the stealthy invocation of the measurement process, which is necessary to ensure that a compromised system does not have a chance to hide the attack traces upon detecting an up-coming measurement. The second is the ability to do in-context measurement, which is necessary to retrieve all the needed inputs for a successful integrity measurement. In Chapter 5, we present how HyperSentry achieves both properties. Moreover, there are other approaches that aim to enforce the protection of the highest privileged software. HyperSentry complements these approaches 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.

3.3 Isolated Execution Environments

In cloud computing environments, where a physical platform is shared among mutually distrusted end users, isolation becomes a basic security requirement. Traditionally, operating systems are used to provide isolation between the running processes. However, operating systems provide a large Trusted Computing Base (TCB) that is vulnerable to a wide range of attacks. To minimize the TCB, researchers (e.g., [14, 80]) used hypervisors to enable strong isolation between guest VMs. Along the same line of research, several systems (e.g., [21, 27, 104]) use the hypervisor to directly provide memory protection for running workloads without relying on the operating system of the guest VMs. Nevertheless, recent attacks [41, 59, 101] and vulnerability reports [1, 2, 4, 82, 83] show that hypervisors are subject to security exploitation and can be compromised.

There have been a few recent attempts to tackle the problem of isolating sensitive workloads while eliminating the host OS or hypervisor from the TCB of these workloads. These attempts can be divided into two main categories: (1) microhypervisor-based approaches, and (2) hardware-based approaches.

Microhypervisor-based Approaches: These approaches rely on a thin, privileged software layer (i.e., a thin hypervisor) to provide the required isolation for sensitive workloads. Among the notable research efforts in this direction are NOVA [92] and Trustvisor [67].

NOVA proposes to replace current hypervisors with a microhypervisor that is around 9 KLOC in size. Despite having a small TCB compared to commodity hypervisors, NOVA is still responsible for several management tasks (e.g., address space management, interrupt and exception handling, and communication between the running workloads). Thus, its TCB is still relatively complicated.

Trustvisor minimizes the code base of the microhypervisor even further (about 2 KLOC for its core functions), so that it can be used for isolation purposes only. However, Trustvisor is only designed to handle workloads with a small code base and only supports systems with a single processor core. This makes it unsuitable for typical compute clouds.

An effort closely related to these microhypervisor-based approaches is seL4 [58]. As discussed in Section 3.2, seL4 proposes a technique to formally verify a microkernel, which is around 8.7 KLOC, to avoid security vulnerabilities. Although this microkernel can be used for isolation purposes, the formal verification process imposes several restrictions on the microkernel functionality. Thus, it cannot be extended to fully support all the functionalities required from micorhypervisors yet.
**Hardware-based Approaches:** These approaches rely on hardware security extensions to provide isolation for sensitive workloads. The main advantage of these approaches is the enhanced protection for the isolated workloads (compared with software based techniques).

Flicker [68] can also be used to provide isolation for a secure verifiable workload. However, the late launch capability, provided by both Intel [46] and AMD [6], incurs significant overhead (in the magnitude of hundreds of milliseconds) on every context switch to the isolated environment. Flicker also requires that the system dedicate all its resources, including all processor cores, to the trusted environment. Hence, Flicker, or any other system that relies on the late launch capability (e.g., BIND [90]), is not a practical solution to provide isolation for sensitive workloads that run in cloud computing environments.

Lockdown [97] relies on ACPI mode switching to create the isolated environment. However, it requires tens of seconds to switch to the isolated environment, which cripples its ability to work effectively in a production environment.

Another effort, NoHype [56,93], also relies on hardware isolation through assigning dedicated processor cores to each running workload. However, NoHype still relies on a hypervisor to achieve the required protection and manage the hardware resources (e.g., page tables).

Intel is proposing a hardware service called Processor-Measured Application Protection Service (P-MAPS) [79] that offers runtime memory isolation and verification of a protected application. However, there is insufficient detail about the size of its isolated environment and its ability to provide multi-core isolation.

Some other processor architectures provide support for isolated execution. IBM introduced interesting security features in its Cell Broadband Engine (Cell BE) architecture [91], which provides multi-core isolation. ARM also introduces its TrustZone [8] technology, which provides the isolated execution of a verifiable code in a secure world. However, these features are unique to these specific architecture. We need to achieve the same level of isolation on the x86 architecture which is used in the vast majority of cloud computing infrastructures.

Finally, some other approaches proposed to modify the current hardware architecture so that it can support secure isolated execution. For example, SecureME [22], which stands for Secure My Execution, proposed a secure processor substrate that is controlled by a small TCB (in the magnitude of hundreds of lines of code). SecureME can be used as a security service that allows applications to request the processor to protect their address space from other running applications. Although SecureME does not require modifications to commodity operating systems, it requires architectural change to processors and hardware peripherals, which is slow to realize.

**Limitations:** To summarize the previous discussion, all traditional isolated execution techniques that rely on the OS or hypervisor has been subject to attacks. Researchers have been
investigating the elimination of the traditional OS or hypervisor from the TCB of the isolated execution environments. However, there are fundamental drawbacks that face all the proposed techniques. In particular, these drawbacks would prevent the applicability of these techniques in cloud computing environment.

In Chapter 6, we present SICE, which provides a unique TCB that has less than 300 source lines of code to achieve the required isolation for secure workloads in cloud computing environments. The main reason SICE has a very small TCB (compared to microhypervisors like NOVA and Trustvisor) is that SICE TCB functionalities are limited to providing isolation. On the other hand, microhypervisors are still required to provide other functions as a result of occupying the highest privileged level of the system. In a system adopting SICE, all other functions that are not handled by the TCB, whether they are related to system management or other security functions, can be handled by other system components. Eliminating such functions from the TCB required for isolation is one of the desirable properties of SICE.

Finally, it should be noted that all of the above techniques are research prototypes that make several simplifying assumptions about their target platforms. These assumptions may hinder the applicability or even weaken the security guarantees provided by these techniques. For instance, the formal verification introduced by seL4 does not include the firmware or the SMM code, which could negate all seL4 guarantees. Thus, a practical deployment of any of these research prototypes, including the prototype we present in Chapter 6, requires a comprehensive consideration of the target platform that includes the full hardware and firmware specifications.
Chapter 4

HIMA: A Hypervisor-Based Integrity Measurement Agent

Integrity measurement is a key issue in building trust in distributed systems. A good solution to integrity measurement has to provide both strong isolation between the measurement agent and the measurement target and Time of Check to Time of Use (TOCTTOU) consistency (i.e., the consistency between measured version and executed version throughout the lifetime of the target). Unfortunately, none of the previous approaches provide (or can be easily modified to provide) both capabilities. We present HIMA, a hypervisor-based agent that measures the integrity of Virtual Machines (VMs) running on top of the hypervisor, which provides both capabilities identified above. HIMA performs two complementary tasks: (1) active monitoring of critical guest events and (2) guest memory protection. The former guarantees that the integrity measures are refreshed whenever the guest VM memory layout changes (e.g., upon creation of processes), while the latter ensures that integrity measurement of user programs cannot be bypassed without HIMA’s knowledge. We also report the experimental evaluation of a HIMA prototype using both micro-benchmark and application benchmark; the experimental results indicate that HIMA is a practical solution for real world applications.

4.1 Introduction

Remote attestation is a process in which a computer system proves its integrity to a remote party. It plays an important role in improving trustworthiness in distributed computing environments. Essential to remote attestation is the integrity measurement of the attested system; the integrity information allows the remote party to verify the configuration and the current state of the attesting platform (e.g., hardware and software stack), thereby to determine its trustworthiness.
In Section 3.1, we presented an overview of recent attempts on integrity measurement. That review testifies to the difficulty of using virtualization to provide solid system integrity measurement. Indeed, a good solution has to provide both of the following capabilities, which none of the previous approaches have, or can be easily modified to have.

• **Strong isolation between the measurement agent and the target:** Although isolation can be achieved by using virtualization, the measurement technique must not rely on any components or hooks that can be subverted by exploiting a guest vulnerability.

• **Time of Check to Time of Use (TOCTTOU) consistency:** This guarantees the continuity of the integrity evidence beyond the measurement time. This problem arises from the fact that an attacker can change a program after it is measured and before it executes.

We present HIMA, a hypervisor-based agent that measures the integrity of guest VMs running on top of the hypervisor, which provides both capabilities identified above. HIMA is located in the hypervisor, and performs a comprehensive “out-of-the-box” measurement of guest VMs, including both their kernels and user programs. HIMA guarantees the TOCTTOU consistency for (static) integrity measurements of guest user programs as well as the integrity of the measurements, even if the guest kernel is compromised.

HIMA uses two complementary techniques that together guarantee TOCTTOU consistency for integrity measurement: (1) *active monitoring of critical guest events* and (2) *guest memory protection*. The former monitors all events that change guest program’s layout (e.g., creation of processes), so that the measurement can be performed accordingly and ensured to be up-to-date. The latter guarantees capturing any attempt to modify measured programs or to bypass the measurement.

HIMA uses the information it obtained from the guest VMs (through active monitoring) to facilitate the integrity measurement. However, the TOCTTOU consistency does not depend on this information. Instead, it is guaranteed by enforcing the policy that only measured memory pages are executable.

During the development of HIMA, we had to resolve the following challenges:

• **Challenges in active monitoring:** Although intercepting guest events can be straightforward, recognizing their impact on the state of the guest VM is non-trivial. HIMA has to intercept the system call before returning to the calling process, introspect all the needed information and overcome any guest behavior that complicates the monitoring process (e.g., reentrant kernels).

• **Challenges in guest memory protection:** Attackers can either change program binaries after they are measured or corrupt the kernel’s data structures to mislead HIMA.
To overcome these threats, we guarantee that measured portions of the memory will not change and unmeasured programs will not be executable. HIMA manipulates the hypervisor’s control of MMU to solve these problems. Though the principle is straightforward, HIMA has its unique memory protection scheme that benefits from the active monitoring facility while guaranteeing the required semantic awareness about the state of the guest VM.

- **Challenges in semantic-aware measurement:** Semantic awareness is necessary to bridge the gap between the intercepted low-level events and the current state of the system. To produce meaningful measurements, HIMA correctly identifies the context of each event, keep an updated view of the guest state, and overcome any guest behavior that may hinder the measurement process (e.g., the on-demand loading of new program pages).

We implemented a prototype of HIMA using Xen [5], and performed a substantial set of experiments to evaluate its performance using both micro- and application benchmarks. Our micro-benchmark results show a reasonable overhead on monitored guest events (e.g., forking processes), which are infrequent and constitute a small portion of the life-cycle of common applications. Our application benchmark results further show a light impact on application execution.

Our primary contribution in this work is a collection of non-trivial engineering efforts that use the hypervisor to provide strong isolation between the measurement agent and the measurement target, and ensure TOCTTOU consistency for the integrity measurement of user programs running inside guest VMs. Among these efforts, we develop novel techniques to handle the complication involved in active monitoring, such as processing the interception of privileged events with multiple return points and reentrant kernels. We also develop techniques to overcome many of the semantic challenges that face the integrity measurement process, such as on-demand loading. Moreover, we adapt existing guest memory protection mechanisms to ensure that what’s executing is indeed what’s measured. Our solution suite provides a strong and semantic aware binding of measured programs and the executable permission of their code pages. As a result, HIMA guarantees TOCTTOU consistency for integrity measurement of guest VM user programs, even if the guest kernels are compromised.

The rest of this chapter is organized as follows. Section 4.2 discusses the assumptions and threat model. Section 4.3 gives an overview of HIMA. Section 4.4 explains the details of active monitoring in HIMA. Section 4.5 presents the guest memory protection techniques. Section 4.6 gives the security analysis of HIMA. Section 4.7 presents HIMA’s performance evaluation.
4.2 Assumptions and Thread Model

**Assumptions:** We assume a typical virtualized platform consisting of the hardware, a hypervisor, and a management VM. Our measurement targets are the guest VMs running on this platform. We assume the virtualized platform can attest to its own integrity using existing static measurement techniques (e.g., IMA [81]). The same assumption has been adopted in many virtualization-based security architectures [13, 35, 51, 73]. Despite that the guest kernels are measured at load time, we do not trust guest kernels due to potential runtime vulnerabilities. Indeed, even if the kernel code is intact, kernels may misbehave due to compromised kernel data. We also assume that the hardware is equipped with the Non-eXecutable (NX) bit support. Guest VMs should support separating user program’s code and data pages to function properly.

**Threat model:** We consider all attacks that require code injection or modification of running programs. We assume that attackers have full access to guest VMs. In particular, attackers may gain control of the guest kernels and attempt to mislead the underlying integrity measurement. The attackers may attempt to compromise the system and then hide their traces. This raises challenges to our system, which should be valid throughout the lifetime of the guest VMs.

We focus on the measurement of the static portion of the guest VMs. We do not consider the measurement of dynamically generated code, such as executables generated by a Java JIT compiler on an executable heap or self-modifying code. The integrity of dynamic data is also a critical issue for system integrity, which requires orthogonal techniques. We consider these out of scope.

We target the specific objective of achieving TOCTTOU consistency for guest *user programs*. On the other hand, the protection of kernel code has been thoroughly studied in complimentary techniques (e.g., NICKLE [76]). We can safely assume that any of these techniques can be integrated with HIMA to protect guest kernel code. Finally, direct memory modification through DMA is also out of scope as it can be simply prevented by using the IOMMU.

4.3 Overview of HIMA

Located inside the hypervisor, HIMA is properly isolated from the measurement targets. It generates and maintains the measurements (hashes) of the code segments of all kernel components and user programs running inside the guest VMs.

Figure 1 shows the architecture of HIMA. HIMA places hooks inside the hypervisor’s code that handles guest VMs’ privileged events. Thus, HIMA intercepts all these events including: hypervisor service requests (e.g., hypercalls, VMExit), system calls and hardware interrupts.
The generated measurement lists can be stored in the management VM for the convenience of attestation or validation.

HIMA is transparent to guest VMs. It solely relies on the escalated privileges available to generic hypervisors, and can be implemented (or ported) to any hypervisor platform.

We implemented a prototype of HIMA on Xen in X86/64 architecture. Our current implementation is customized to measure para-virtualized Linux guests. Nevertheless, it can be easily modified to support other guest systems (e.g., fully-virtualized guests). In the rest of this chapter, we will use our prototype as an example to clarify the details of our approach.

HIMA utilizes the hypervisor’s control on guest VMs initiation to measure their booted kernels and initial loadable modules. This process is straightforward, and has been studied previously (e.g., [35]). Thus, we do not discuss it in detail. Beyond guest VMs booting, HIMA performs two major tasks to measure the guest VMs’ integrity and ensure TOCTTOU consistency of measured user programs: (1) active monitoring of guest events, and (2) runtime memory protection.

### 4.4 Active Monitoring of Guest VMs

Active monitoring allows HIMA to keep a fresh view of the memory layout of guest VMs. When a guest VM creates, terminates, or modifies its user processes or kernel modules, HIMA intercepts the relevant events and refreshes the integrity measurements. HIMA completes all its measurements before the control jumps to the loaded program to guarantee that no instruction runs inside the system before being measured.

There are several attempts to provide VM active monitoring, including Lares [73], Xenprobes [75], and VMScope [50]. However, both Lares [73] and Xenprobe [75] rely on hooks placed in the guest VMs, which can be bypassed by intelligent attackers, while VMScope [50]
has significant performance overhead due to the dependency on emulation-based VMM.

HIMA performs its active monitoring differently; it is solely based on the intercepted guest events, including *system calls, interrupts*, and *exceptions*. HIMA faces a number of technical challenges that make active monitoring much more complicated than it appears. In the following, we describe these challenges and the key techniques used to overcome them.

### 4.4.1 Introspecting Event Context and Impact

In our virtualization architecture, system calls normally trap into the hypervisor before they are forwarded to the guest kernel. HIMA intercepts all system calls issued by the guest VMs, either issued using `int 0x80` or `syscall` instructions. To achieve a complete active monitoring, HIMA introspects the context of the intercepted calls and their input parameters. Moreover, it traps the calls before they return to inspect their output and assess their impact on the system state.

To introspect a guest VM event context and inputs, HIMA inspects the guest VM’s registers, stack, and heap as soon as the event is trapped. Introspected information includes: (1) The event type (e.g., the system call number stored in the **EAX** register), (2) The current user process identified by the address of the base page table of its address space (the **CR3** register), (3) Event parameters stored in registers, stack or heap and (4) Both instruction and stack pointers of the running process.

As soon as the guest kernel finishes handling the event, HIMA forces it to trap into the hypervisor. For interrupts and exceptions, HIMA replaces the event’s return address stored in kernel’s memory with an illegal address. As soon as the event returns, this replacement causes a protection fault that traps into the hypervisor. Thus, HIMA can perform the necessary introspection and then return to the original return address.

However, we cannot use the same technique for system calls because the guest kernel can potentially rewrite a system call’s return address while handling the call. One example is the *execve* system call, which loads a new program. If this call succeeds, the kernel modifies the system call’s return address so that the execution returns to the new loaded program. On the other hand, if the system call fails, the kernel simply returns the error code to the calling process. It is easy to see that a trapping technique that relies on rewriting the return address will fail when such system call succeeds.

**Handling Events with Multiple Return Points:** HIMA uses debug registers [6, 48] to address the above problem. HIMA places the return address of a system call in one of the debug registers. Consequently, the CPU inspects every instruction fetch till this specific address is fetched from the memory. At this point, a special interrupt is raised and the execution traps into the hypervisor. To overcome the scenario where the guest kernel rewrites the return addresses of
the system call, HIMA utilizes the presence of multiple debug registers per physical platform and the ability of these registers to trap write and read operations involving monitored addresses. HIMA determines the location of the system call’s return address in the kernel stack. It then copies the address of this location to another debug register and set it to cause a debug exception as soon as this address is rewritten. This only happens when the `execve` call succeeds. HIMA then waits for either the write or the execute debug exception to occur in order to introspect the impact and results of the call and clear the debug registers.

Handling Reentrant Kernels: Our use of the debug registers to track the execution of system calls gets complicated when guest kernels support *kernel reentrancy*. Although a system call is a blocking event for user processes, they are not for the kernel. Modern kernels can relay the execution of a system call to a hardware peripheral and schedule another process to run in the meantime [15]. If the new process issues another system call, the call is handled by another kernel thread. In order to support kernel reentrancy, HIMA has to solve the following problems:

- There are a limited number of debug registers per CPU (e.g., 4 debug registers on the x86 platform) [6, 48], which may not be enough to monitor concurrent kernel threads.
- Debug registers use virtual addresses. When a context switch occurs, the contents of the debug registers can be interpreted incorrectly in the new address space.

To overcome these problems, HIMA sets the debug registers only when a monitored execution path is currently running. Whenever a context switch occurs, HIMA either resets the debug registers or sets it with the new values required to monitor the new execution path. This approach provides HIMA with the needed flexibility of monitoring multiple events. At the same time, it minimizes the performance overhead resulting from the use of the debug registers.

To monitor guest context switching, HIMA keeps track of the kernel stack pointer. Multiple execution threads require a separate stack for each thread [15]. Hypervisors control the address of the current kernel stack to correctly forward hardware exceptions and interrupts to the active kernel thread. Upon intercepting this event, HIMA adjusts the debug registers accordingly. In our experimental prototype, kernel context switching is done via the hypercall `do_stack_switch`. Intercepting the hypercall does not violate HIMA’s security guarantees, since it is the only way to do this privileged operation. To modify HIMA to measure a fully virtualized Xen guest, it will need to intercept the proper `VMExit` event.

HIMA keeps a list of monitored threads. Each list entry includes the values of the debug registers, the base address of the thread’s stack and the event type. Whenever a context switch is captured, HIMA compares the stack address of the new thread with the stored addresses of monitored threads. If an entry is found, HIMA sets the debug registers and monitors the event until it either returns or the context changes again.
Figure 4.2: Example of intercepting concurrent system calls (Thick lines denote threads running on CPU)

Figure 4.2 shows an example of handling multiple concurrent guest events. In step 1, a system call is issued by process a. HIMA saves the needed data and sets the debug registers. In step 2, the kernel halts the current thread to schedule process b. This can happen for several reasons (e.g., process a’s system call requires an input from a hardware peripheral). Before context switching, the execution traps into the hypervisor to correctly achieve the privileged operations accompanying the context switching. HIMA resets the debug register accordingly. In step 3, process b issues a system call. HIMA creates a new call entry and sets the debug registers with the new values. In step 4, the kernel finishes process b’s system call. A debug exception traps into the hypervisor to finish its introspection before returning to the user process. In step 5, the kernel resumes process a’s call upon receiving the required input (e.g., an interrupt from the hardware peripheral). (The event that caused the resumption is not shown in the figure.) Again, HIMA gets informed about the context switching and regains the saved debug register values. In step 6, the call finishes and HIMA decides to measure a memory area as a result of this call. At this point, HIMA decides to measure an executable memory based on the outcome of the system call. This measurement process (steps 7 and 8) will be explained in detail in Section 4.4.2.

4.4.2 Semantic-Aware Measurement

HIMA measures the program to be loaded into a guest VM after intercepting the appropriate event. To measure a program, HIMA computes the SHA1 hash of its code and initial data segment as they get loaded into the memory. HIMA handles several situations: (1) loading kernel modules, (2) loading user programs, and (3) cloning (or forking) processes from existing ones. It is straightforward to measure kernel modules, since each kernel module is copied into the guest kernel entirely as a block of memory. Cloning or forking new processes from existing ones relies on programs already mapped into the guest VMs’ memory; we present its treatment later in Section 4.5. In the following, we focus on the second case, the semantic-aware measurement of loaded user programs.
When measuring a program, HIMA collects semantically rich information to produce meaningful measures that can be potentially used in further attestation. HIMA identifies the program’s memory location and its ID (e.g., file name), and calculates the sequential hash of the whole program to identify its exact version and hence verify its integrity.

**Identifying program’s memory map:** HIMA measures programs based on their memory mapped binaries. In order to identify a program’s memory map, HIMA intercepts both `mmap` and `execve` system calls. After an `execve` call succeeds, HIMA reads the guest memory to introspect the needed information. HIMA begins introspection based on the address of the current guest kernel stack. The bottom of the stack points to the current process descriptor. The descriptor includes start and end addresses of both the code and the data segments of the new process. It also includes the address range of any pre-loaded executable libraries (e.g., the dynamic linker). On the other hand, the return of an `mmap` system call denotes the address of the mapped memory area while the area’s size is one of the system call’s input parameters.

**Handling on-demand loading:** After mapping new binaries, kernels follow an on-demand loading scheme that reads the required binaries page by page whenever a page needs to be read or executed. This behavior contradicts our intentions of calculating the hash of the entire application sequentially.

To overcome this problem, HIMA forces guest kernels to change their behavior and load the entire program before it runs. Although this may incur some performance overhead, it ensures that HIMA records the hashes of the loaded binaries before their executions to avoid any impact that may obfuscate the measurement process. To force guest kernels to load all the binaries before hand, HIMA blocks the execution of the user process and sends a series of page faults to the guest kernel. Each fault triggers a read of one page in the memory area to be measured. The return address of each exception points to a non-executable memory area to force the kernel to trap back into the hypervisor after loading the page. HIMA calculates the SHA1 hash after the entire memory area is loaded.

This process is shown in steps 6–8 in Figure 4.2. As soon as HIMA captures the system call return in step 6, it generates a page fault to load the first page. In step 7, after loading the page, a protection exception occurs as a result of the illegal return address and the execution traps into the hypervisor again. The process is repeated till all pages are loaded. Finally, in step 8, HIMA measures the memory area, restores original kernel output, and returns to the user process.

**Coupling measured programs with their IDs:** After measuring a program, HIMA adds a new entry to the measurement list. Beside the hash value, the entry includes some other
information (e.g., the program’s file name). The program’s user process and address range are also stored to help enforce guest memory protection, though the success of this enforcement does not depend on the correctness of these parameters.

4.5 Guest Memory Protection

Active monitoring is not sufficient to ensure TOCTTOU consistency of user programs. A runtime vulnerability in the guest kernel or user programs can be used to either bypass the monitored events and skip the measurement process or modify the measured programs before they execute. Thus, HIMA has to further guarantee that both (1) the guest VMs are only able to execute measured user programs and (2) the measured user programs cannot be modified without HIMA’s knowledge.

HIMA achieves the above guarantees through guest memory protection. Specifically, HIMA utilizes the hypervisor’s control of the MMU to enforce its guarantees based on memory pages’ access permissions. Thus HIMA enforces TOCTTOU consistency of user programs in complete isolation of the guest VMs. However, we have to resolve a number of technical issues, including (1) identifying memory mappings of monitored pages, (2) handling remapping of measured executable pages (e.g., due to fork or clone system calls), and (3) affirming that our protection extends to actual physical memory.

Figure 4.3 shows the overall flow of the guest protection mechanism in HIMA. In the following, we first present the basic idea of this mechanism, and then discuss how HIMA overcomes the above technical issues.
4.5.1 Protecting Guest Memory Using Page Access Permissions

To ensure that only measured guest programs can be executed, HIMA utilizes the NX-bit page protection flag [6, 48] (available on most hardware platforms). Executing an instruction from a page protected by this flag will cause an exception that traps into the hypervisor (and thus HIMA).

HIMA traps all guest page table updates to match the address of any executable page with those of measured programs. This step validates the information HIMA introspects from the guest kernel and eliminates the need to trust its data structures. For instance, if a program’s start and end addresses are manipulated in the process descriptor to hide some malicious pages, HIMA will detect the executable mappings of these pages and invalidate the measurement.

Moreover, to prevent programs from changing after being measured, HIMA marks all measured executable pages as non-writable. If an attacker tries to write to a measured page, an exception will be raised and traps into HIMA. An attacker may try to make a measured page writable; however, HIMA disallows a page to be concurrently executable and writable.

4.5.2 Identifying Mapping of Monitored Pages

HIMA needs to differentiate between legitimate mapping of monitored pages and any other malicious page mapping. Specifically, HIMA needs to link any mapped page to one of the monitored memory areas.

HIMA identifies a mapped page by both its address space and virtual address. To identify the address space and consequently the user process, HIMA uses the address of the highest level page table pointed to by the CR3 register. Note that identifying the virtual address of a mapped page depends on the used virtualization architecture. In our prototype using para-virtualized guest VMs, the virtual address was passed as a parameter of the update_va_mapping hypercall, which is the guest VM’s only gateway to update the machine’s page tables. In other virtualization settings like fully virtualized Xen guest VMs, identifying the virtual address will rely on the implementation details of the employed memory mapping mechanism (e.g., shadow vs. nested page tables).

To verify that a mapped page, identified by its virtual address and address space, belongs to one of the monitored memory areas, HIMA adds a few fields to the entries of the list that tracks the currently monitored events (refer to Section 4.4). These fields identify the address space and address range of each memory area being measured. HIMA just needs to match the mapped page with one of these monitored memory areas.
### 4.5.3 Handling Remapping of Measured Executable Pages

During normal operations, some of the pages that belong to measured programs may be legitimately remapped. This behavior results from one of the following cases:

1. Pages removed from the physical memory due to memory shortage and getting reloaded, from the file or swap memory, due to an access demand. These pages may be remapped to new physical frames.

2. Pages that belong to a newly forked process. These pages are usually mapped to the existing physical frames of the corresponding pages in the parent process. However, they can be remapped to new physical frames if the parent process exited or the frames were swapped out.

To verify that a page remapping is legitimate, HIMA needs to identify that (1) the page actually belongs to a valid measured memory area and (2) the page content has not changed if it was moved out of the memory.

A naive approach to solving this problem would be to re-measure the hash of the whole memory area by forcing the guest kernel to load all pages as soon as an executable page is remapped. However, this will cause significant performance overhead and may result in memory shortage due to reloading multiple swapped pages whenever a single page is requested.

To overcome this problem, HIMA stores the hash image of each of the pages that form a measured memory area. Whenever a page is reloaded, HIMA identifies its offset relative to the measured area and then compares its hash with the stored one. We describe this solution in detail next.
Keeping track of user processes: To validate that a mapped page belongs to a legitimate executable memory area, HIMA keeps track of the memory layout of all the processes running in the guest VM. Figure 4.4 shows the structure of the information HIMA keeps about running processes. Each process is uniquely identified by its address space and contains a list of all the measured memory areas in that address space.

HIMA monitors system calls responsible for process creation (e.g., `fork`, `clone`) to add new processes to the list. When a process executes a new program, HIMA updates the list of its measured areas based on the new memory layout. As shown in Figure 4.4, each of the entries that represent a memory area points to the corresponding entry in the measurement list which points to another list of the hashes and physical frame numbers of the pages that constitute the measured program.

Validating the integrity of remapped pages: Upon remapping an executable page, HIMA first validates that it belongs to one of the measured areas in its address space. The next step is validating the integrity of the page content. To minimize hash operations, HIMA exploits the fact that guest VMs reuse the same physical frames, which have been measured and protected, among multiple processes. As shown in Figure 4.4, HIMA keeps the physical frame number of each measured page. HIMA determines the offset of a remapped page in its measured memory area to retrieve the stored page hash and the physical frame number. If the page is mapped to a physical frame that was measured and never been rewritten (explained next), HIMA can safely skip calculating the page’s hash. This scenario is likely upon remapping pages to a new address space of a forked process. On the other hand, if the physical frame has been rewritten or the page is mapped to a new frame, HIMA calculates the page hash and compares it with the stored value. This scenario is likely upon reloading a swapped-out page. To determine if the guest VM has rewritten a measured frame, HIMA keeps track of all the measured frame numbers in a separate list. When any of these frames is swapped out, HIMA removes it from the list. The same list is also used for extending memory protection to actual physical frames as discussed in Section 4.5.4.

Reducing performance overhead through caching page hashes: The same technique is used to minimize forcing guest kernels to bypass on-demand loading. HIMA keeps the hash lists of all the measured programs. If a measured program is reloaded into a new process, HIMA can validate the program’s integrity based on the pre-calculated hashes of its individual pages rather than forcing the kernel to load the whole program.

Storage overhead: Despite the performance enhancement, this technique requires HIMA to store hash lists for all measured programs. However, our evaluation (Section 4.7) shows that
the storage overhead is within acceptable ranges and can be accommodated by increasing the hypervisor’s heap.

4.5.4 Extending Memory Protection to Physical Frames

As mentioned earlier, HIMA provides memory protection based on page permissions. However, these permissions apply to virtual pages rather than the actual physical frames. A physical frame can be referenced by multiple virtual mappings; this allows potential threats such as rewriting a measured page by remapping it to another writable virtual address. HIMA’s operations include two extra steps to protect the physical frames of measured pages.

1. HIMA leverages the stored frame map inside the hypervisor to obtain information about the physical frames of measured pages. Our Xen prototype uses a flag that indicates if the frame has an existing writable mapping.

2. As described in Section 4.5.3, HIMA keeps a list of all measured physical frames. The entries are also coupled with their virtual addresses. Whenever a measured frame is mapped as writable, HIMA makes sure that it does not belong to its protected pages list.

4.5.5 Writable-eXecutable (WX) Memory Regions

To avoid security exploits, kernels should allow user memory to be either executable for code or writable for data, but not both. As mentioned previously, HIMA relies on the enforcement of such W⊕X mapping to provide the protection of the guest user memory.

Though most kernels support W⊕X mapping, there are a few exceptions. For example, Linux kernel v2.6.18 used in our prototype maps shared writable pages as executable. Fortunately, there are orthogonal techniques that can patch such systems to enforce the W⊕X mapping (e.g., PaX [71]).

On the other hand, this protection cannot be enforced on all applications. Some applications rely on WX pages to function correctly. For instance, Java converts input code into executable binaries in the heap. The lack of support for such applications is a limitation of the current version of HIMA.

4.6 Security Analysis

HIMA guarantees the consistency between its measurement lists for user programs and the actual state of the guest VMs. This consistency is based on both the initial trust in the virtualized platform and the protection mechanisms that ensure TOCTTOU consistency of user programs. In this section, we first reason about the trust in guest VMs that HIMA helps
achieve, and then discuss how HIMA prevents potential attacks from violating the TOCTTOU consistency of user programs.

4.6.1 Trust in Guest VMs

HIMA starts the establishment of trust in guest VMs from the initial trust in the hypervisor and the management VM. As part of our assumptions, this can be easily achieved with existing approaches (e.g., IMA [81]). HIMA builds up the trust in a guest VM by verifying the integrity of the boot-up sequence and any programs loaded in the guest VM. Specifically, HIMA measures the base kernel (possibly with the initial loadable kernel modules) and the initial user processes (e.g., the init process in Linux) after retrieving the binaries from the guest VM’s image file. This process guarantees that once a guest VM is booted, its entire content is measured and trusted.

After booting up a guest VM, HIMA monitors all the critical events that may change its program layout, including loading and removing kernel modules (e.g., init_module system call), creation and termination of user processes (e.g., execve, fork, clone, kill system calls), and loading and unloading of libraries (e.g., mmap). It is worth mentioning here that HIMA’s active monitoring can be extended to monitor any system call. In other words, HIMA can be adapted to handle any system functionalities that are not included in our prototype or any future kernel functionalities. Moreover, HIMA exploits hardware based memory protection to intercept all attempts to modify already mapped user programs and prevent unmeasured user programs from being executed. Together, these mechanisms ensure that the integrity measurement is up-to-date. In other words, HIMA guarantees the TOCTTOU consistency of the measures of all guest user programs.

4.6.2 Possible Threats and Defenses

In the following, we discuss the resilience of HIMA against potential threats that aim to inject malicious code into a monitored guest VM. By examining all the possible paths to such attacks, we find that they fall into four broad categories: (1) changing the program layout of the guest, (2) modifying the content of the measured memory, (3) executing unauthorized code, and (4) changing the guest kernel’s code segment.

Security guarantee for user programs: HIMA explicitly binds the measurement and the executable permission of a page together. That is, the unmeasured code pages will not have the executable permission. As a result, any attempt to maliciously change the program memory layout will not result in executable pages. Moreover, any attempt to modify a measured page will fail due to the non-writable permission placed on the page by HIMA. Finally, any attempt to
execute unauthorized code will also fail, since the code pages will not have the right executable permissions. In other words, all attempts in the first three classes of attacks will fail due to HIMA’s guest memory protection.

Note that even if a runtime data vulnerability is exploited to compromise the guest kernel, the above argument still holds, because the compromised kernel does not have control over the guest memory protection mechanism enforced by HIMA. Thus, HIMA will not allow such attacks to change any measured user programs or load new ones without its knowledge (i.e., TOCTTOU consistency). Although HIMA extracts some semantic information from the guest kernel (e.g., file name, program address range), such information is only used to facilitate the measurement process and any future attestation. As mentioned earlier, the guest memory protection does not rely on such information.

The above argument also applies to the execution of scripts. An attacker may attempt to exploit a legitimate interpreter to perform malicious actions by supplying compromised script files. HIMA can be easily extended to measure the contents of input files as they are read by the kernel. In this case, HIMA needs a specific input to identify these files and enforce the required security guarantees.

**Security guarantee for guest kernels:** Almost all major OS kernels need mixed code and data pages to function correctly. The behavior clearly contradicts with HIMA’s memory protection mechanism, which requires $W\oplus X$ mapping to protect the kernel code segment. Fortunately, as mentioned in section 4.2, there are existing mechanisms that resolve this problem (e.g., NICKLE [76]). (Note that these mechanisms can separate the kernel code from kernel data and prevent kernel code modifications, but they cannot protect the kernel data, which can still be modified if an attack can exploit some kernel vulnerabilities.) With the presence of such a mechanism, HIMA’s memory protection mechanism would right away extend to include kernel code injection or modification.

**Limitations:** HIMA does not handle applications that rely on writable and executable memory pages (e.g., Java, self-modifying code), as discussed earlier. There are also threats that HIMA is not intended to handle, for example, attacks that exploit data vulnerabilities either in user programs or the guest kernel. The current version of HIMA only focuses on the integrity of code running inside the guest VMs.

### 4.7 Experimental Evaluation

We perform an extensive set of experiments to evaluate the performance of HIMA. Our experimental platform is a DELL OptiPlex 755 PC, which has a 3.0 GHz Intel dual-core processor
with 4 GB RAM. Our HIMA prototype uses Xen version 3.2.1 as the base hypervisor. The management VM (i.e., Dom0 in Xen’s terminology) runs 64-bit Fedora 8 Linux, and the guest VM that HIMA measures runs para-virtualized 32-bit Fedora 8 Linux. In all experiments, Xen allocates one virtual CPU and 512 MB RAM to the guest VM. Both the management VM and the guest VM use 2.6.18 Linux kernel.

4.7.1 Evaluation Methodology

Our goal is to assess the performance overhead introduced by HIMA, including both computational and storage overheads. We use both a micro-benchmark and an application benchmark. The former is used to understand the direct computational overhead introduced by HIMA on the guest kernel (the impact of intercepting the privileged operations), while the latter is to determine the overall overhead on applications.

**Micro-benchmark:** We use UnixBench [96] in our micro-benchmark evaluation. HIMA has three main operations that add overhead to guest kernels: (1) intercepting certain events, (2) verifying the integrity of mapped pages, and (3) measuring executable programs. The first step includes intercepting the `execve`, `mmap`, `fork`, `clone`, `vfork`, `open`, `create`, `dup` and all versions of the `exit` and `kill` system calls. HIMA also intercepts all the interrupts and exceptions generated by the hardware, particularly page faults, debug exceptions, and protection faults. The last two steps rely on calculating the SHA-1 hash, which is relatively expensive. The number of hash operations can be reduced significantly through caching the hash values of individual pages. To evaluate the effectiveness of caching, we evaluate two different versions of HIMA: with and without caching.

**Application benchmark:** We hypothesize that HIMA will introduce an overhead on guest applications only when they require a monitored operation. To validate this hypothesis, we choose a diverse set of five applications: two file compression applications, a web server, a download manager and a kernel compilations benchmark. The file compressing applications (gzip and bzip2) evaluate the performance incurred by extensive I/O operations, with the latter being more computationally intensive. Both programs are evaluated based on the time they use to compress an 11 MB file. The web server (apache) relies heavily on both threading and I/O operations. We compare the server throughput in delivering a 1.2 KB file using ApacheBench v2.3 with 50 concurrent requests. To eliminate networking overhead, we ran the benchmark on the same guest VM that runs the web server. To verify that HIMA does not incur any overhead on network operations, we use the download manager prozilla to download a 600KB file from a local server. We ran this experiment over a weekend in order to avoid any overhead added by the network. Finally, to evaluate the performance of applications that are computationally...
extensive and rely on threading and file operations at the same time, we use the Linux kernel compile benchmark \texttt{kcbench}.

**Storage overhead:** We estimate the amount of heap memory needed by HIMA to store the necessary guest state and cached hashes. We run all our benchmarks, both the micro- and application benchmarks, on a single running instance of the guest and calculate the maximum amount of memory required by HIMA.

### 4.7.2 Result of Micro-benchmark Evaluation

Figure 4.5 shows the result of micro-benchmark evaluation. HIMA introduces less than 5% overhead in all test cases except for \texttt{execl}, process creation, and shell script tests. These tests measure the ability of the machine to do extensive computation and I/O operations that do not include monitored events other than the one-time loading of the test programs.

The \texttt{execl} test exposes HIMA’s overhead for monitoring and measuring program-loading events. The test consists of a loop that runs a dummy program. When caching is enabled, HIMA introduces a 31% overhead. It includes a one-time measurement of the whole program (and the needed libraries) and page-level hash comparisons with cached values. When caching is disabled, HIMA remeasures the test program every time, and the overhead jumps to 75%.

Another test that incurs a large overhead is process creation through forking. Forking is a monitored operation that results in newly mapped pages to be verified. When caching is enabled, HIMA introduces 21% overhead. Disabling caching does not add much overhead, because the forking process stays in memory till the child loads, and these processes usually share the same physical frames.

Finally, the shell script test represents HIMA’s worst case scenario. It consists of both loading new programs (\texttt{bash shell}) and forking processes to run concurrent tests. Moreover,
the shell needs more libraries than the simple `exec1` test. With caching, HIMA’s overhead is 43%. The overhead rises dramatically to 99% when caching is disabled due to frequent measurements of the shell program and needed libraries.

### 4.7.3 Result of Application Benchmark Evaluation

Figure 4.6 shows the application benchmark results. For `gzip` and `bzip2`, HIMA introduces 1.25% and 0.17% overhead, respectively. As expected, `bzip2` introduces less overhead due to its computational intensive nature. The `apache` server introduces 4.64% overhead. Based on the micro-benchmark results, we can deduce that this overhead is mainly due to forking multiple processes to handle concurrent requests. The overhead by `prozilla` is negligible as the test’s bottleneck is the network operations rather than any of the monitored events. This test verifies that HIMA does not introduce much overhead on networking operations.

Although kernel compiling is not a frequently used application, we choose it to represent the worst case scenario. The operations of `kcbench`, rely on a huge number of concurrent processes. These processes run different programs to compile, link and manage kernel code files (e.g. `gcc`, `cc1`, `rm`). Consequently, HIMA monitors the creation of each of these processes and validates the integrity of all programs that run inside them. As the lifetimes of these processes are relatively short, the monitored operations represent a significant portion of their life cycle. Hence, this test shows a relatively higher overhead of 17.48%.

### 4.7.4 Storage Overhead

We monitored the memory required by HIMA to run all the evaluation programs. The maximum memory needed during the evaluation process was around 2.6 MB. Almost half of this memory is consumed by storing page hashes. Considering the fact that Xen’s heap size is configurable and physical memory is a relatively abundant resource on today’s computing platforms, we believe that the memory demand can be accommodated easily. The space used for caching...
page hashes could be more controversial, since this space may continuously grow and cause memory starvation if the guest VM operates for a long period of time. Fortunately, HIMA can take advantage of any cache management technique to efficiently use a pre-allocated amount of memory.
Chapter 5

HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity

This chapter 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.
5.1 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 [35], Lares [73], HIMA [9], vTPM [13] and SIM [89]).

Unfortunately, hypervisors did not turn out to be completely secure. A perfect example is Xen [5], which is a popular hypervisor used in Amazon’s Elastic Compute Cloud (EC2) [7]. Recent attacks showed that Xen’s code and data can be modified at runtime to allow a backdoor functionality [101]. 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 [83].

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 [82].

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.

5.1.1 Challenges

In Section 3.2, we reviewed recent attempts to measure the integrity of the highest privileged software. That 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.

5.1.2 Introducing HyperSentry

HyperSentry is 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. We use Intelligent Platform Management Interface (IPMI) [45], 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) [44]).

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 com-
modesty 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 [23]. To validate HyperSentry, we implement an integrity measurement agent to verify the integrity of the Xen hypervisor [5]. 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%.

The rest of this chapter is organized as follows. Section 5.2 discusses our assumptions, threat model, and security requirements. Section 5.3 presents HyperSentry in detail. Section 5.4 discusses a case study of HyperSentry using Xen as the measurement target. Section 5.5 presents the implementation and experimental evaluation of our HyperSentry prototype.

5.2 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 [94] trusted boot hardware (i.e., BIOS with Core Root of Trust for Measurement (CRTM) and Trusted Platform Module (TPM) [95]). 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 [29, 102]. Fortunately, such attacks can be prevented using proper hardware configurations (e.g., System Management Range Register (SMRR) [48]).

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 5.4), 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.

5.3 The HyperSentry Framework

Figure 5.1 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 5.2. The CRTM, which is a part of the BIOS, measures itself and the code to be executed next. This process continues until 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 5.2: Building trust in SMI handler](image)

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 HyperSentry 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.

5.3.1 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 5.3 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.

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) [24] or Intel AMT’s Manageability Engine (ME) [44]) 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
Figure 5.3: Hardware components involved in remote SMI invocation

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 DLCK 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.
5.3.2 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 [6].

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 [48].

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 [6]. 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.

As illustrated in Figure 5.4, 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).

5.3.3 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., [29], [102]), 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 tables, 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 5.3, 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 5.4, 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.

5.3.4 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 5.3.1, 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 $K^{-1}_{smm}$ 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 $K^{-1}_{smm}$ so that it cannot be accessed by the hypervisor.

![HyperSentry's attestation process](image)

**Figure 5.5: HyperSentry’s attestation process**

Figure 5.5 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 different signed values. The first is the static attestation output, signed by the TPM private attestation integrity key ($K^{-1}_{AIK}$). 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 ($K^{-1}_{smm}$). The remote user accepts the measurement output only if the private key ($K^{-1}_{smm}$) 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.

### 5.3.5 Security of the HyperSentry Framework

In this section, we discuss how HyperSentry meets all the security requirements presented in Section 5.2.

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 5.3.1. 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 5.3.1, 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 5.3.3. 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 5.3.4. 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., [74], [55]). 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 cause 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.

5.4 Verifying the Integrity of Xen – A Case Study

In this section, we present a case study of HyperSentry using Xen [5]. 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 5.3.3.

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.

In terms of the techniques used to verify the integrity of Xen, we benefit from the work done in [55] to detect persistent kernel control flow attacks. Other work in this direction (e.g., [12,28]) can be used to enhance the measurement agent.

**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 [55] 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., \texttt{SYSENTER_EIP}).

To verify indirect function calls, we use CodeSurfer \cite{40}, 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 \cite{55}, which can be misled by ambiguous type definitions (e.g., \texttt{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 5.2, HyperSentry can only discover persistent changes in the hypervisor. More research is needed to address such integrity violations.

\textbf{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 \texttt{page_list}. Each instance of this list is a member of another list called \texttt{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 \texttt{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.

5.5 Implementation and Experimental Evaluation

We implemented a HyperSentry prototype on an IBM BladeCenter H chassis with HS21 XM
blade servers [23]. 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 5.3 and 5.4.

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 opera-
tion. 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.

5.5.1 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 trig-
nering 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 manage-
ment 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 5.1 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>
<thead>
<tr>
<th>Operation</th>
<th>Average Time</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent Invocation</td>
<td>2.78</td>
<td>0.0616</td>
</tr>
<tr>
<td>Checksum Code</td>
<td>8.82</td>
<td>0.0007</td>
</tr>
<tr>
<td>Retrieve Hypervisor Checksum</td>
<td>0.85</td>
<td>0.0010</td>
</tr>
<tr>
<td>Verify Data (Dom0 and 1 VM)</td>
<td>18.84</td>
<td>0.0162</td>
</tr>
<tr>
<td>Verify Data (Dom0 and 2 VMs)</td>
<td>21.39</td>
<td>0.0288</td>
</tr>
<tr>
<td>Finalize Measurement Session</td>
<td>1.15</td>
<td>0.1132</td>
</tr>
<tr>
<td>Total (Dom0 and 2 VMs)</td>
<td>35.04</td>
<td>0.1648</td>
</tr>
</tbody>
</table>

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.

5.5.2 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 [96]. 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.

The results of our experiment are shown in Figure 5.6. 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 5.6, 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.
Chapter 6

SICE: A Hardware-Level Strongly Isolated Computing Environment for Commodity x86 Multi-core Platforms

The adoption of cloud computing has forced users to trust the ability of hypervisors to provide isolation and protection for sensitive workloads. Unfortunately, there have been numerous exploits against hypervisors and as the code base of hypervisors grows the situation will only get worse.

SICE is a novel framework to provide hardware-level isolation and protection for sensitive workloads running on x86 platforms in compute clouds. Unlike existing isolation techniques, SICE does not rely on any software component in the host environment (i.e., an OS or a hypervisor). Instead, the security of the isolated environments is guaranteed by a trusted computing base that only includes the hardware, the BIOS, and the System Management Mode (SMM). SICE provides fast context switching to and from an isolated environment, allowing isolated workloads to time-share the physical platform with untrusted workloads. Moreover, SICE supports a large range (up to 4GB) of isolated memory. Finally, the most unique feature of SICE is the use of multi-core processors to allow the isolated environments to run concurrently and yet securely beside the untrusted host.

We have implemented a SICE prototype using an AMD x86 hardware platform. Our experiments show that SICE performs fast context switching (67 µs) to and from the isolated environment and that it imposes a reasonable overhead (3% on all but one benchmark) on the operation of an isolated Linux virtual machine. Our prototype demonstrates that, subject to a...
careful security review of the BIOS software and the SMM hardware implementation, current hardware architecture already provides abstractions that can support building strong isolation mechanisms using a very small SMM software foundation of about 300 lines of code.

6.1 Introduction

As discussed in Chapter 1, a need emerges for a solution that provides strong isolation to workloads running in the cloud, yet still allowing hardware sharing to reduce the operating costs. To provide strong isolation, we need to minimize the code base that is granted full access to the memory of running workloads. Thus, we can minimize the exposure to security vulnerabilities that can evade the isolation provided to these workloads. Moreover, this code base should be provided with enhanced security protection and the ability to attest to its integrity.

To achieve this objective, we introduce a prototype system that provides a strongly isolated execution environment, which relies on a trusted computing base (TCB) composed of the hardware, the BIOS, and the System Management Mode (SMM).

Our prototype represents a test system that aims to explore the capability of current hardware platforms in providing more secure isolated environments. We demonstrate that current hardware architecture already provides abstractions that can support strong isolation. Moreover, we show that building strong isolation mechanisms on top of those abstractions requires a very small software foundation of about 300 lines of code (LOC), which tremendously reduces the TCB size compared with previous techniques.

Since the SMM was neither designed nor implemented with high-assurance security mechanisms in mind, detailed security reviews by both CPU and platform vendors would be necessary to verify that current SMM implementations are properly done to support such strong isolation guarantees provided by our prototype in practice.

6.1.1 Introducing SICE

In this Chapter, we present SICE, which stands for Strongly Isolated Computing Environment, a framework that provides a hardware-level isolated execution environment for x86 hardware platforms. SICE’s main objective is to minimize the TCB required to create an isolated execution environment on commodity x86 platforms. This isolated environment can be used to host a security sensitive workload.

SICE achieves this objective by relying on a TCB that is only composed of the hardware, the BIOS, and the SMM. The TCB, which is fundamentally different from the previous research discussed in Section 3.3, gives SICE principal advantages over both microhypervisors and hardware-based isolation techniques. We summarize these advantages below.
Smaller Attack Surface: SICE utilizes the hardware protection provided by commodity x86 processors for the SMM and the memory that hosts its code, which is called System Management RAM (SMRAM). There are two fundamental differences between the SMM and microhypervisors.

First, the SMM can only be triggered by a single interface, which is to invoke a System Management Interrupt (SMI). In SICE, the SMI handler is required to execute one of only four functions upon receiving an SMI, which are to create, enter, exit and terminate an isolated environment. Implementing these functions requires the system to run briefly in the SMM. Moreover, the SMI handler is not required to handle any other interrupts because all interrupts are disabled upon entering the SMM mode. The SMI handler is also not responsible for managing the communication between running workloads. On the other hand, microhypervisors reside at the system’s highest privileged level. Thus, they have to handle all system events (e.g., hypercalls, interrupts and exceptions). They are also required to manage the communication channels (e.g., shared memory pages) between different workloads.

Second, SICE uses the SMRAM to provide the needed memory isolation. After the SMRAM is initialized by the BIOS, it can be locked so that no software can access its contents except for the SMM code. The SMM code can manage the SMRAM through modifying only two registers, which has a huge impact on decreasing the size of the TCB. In contrast, microhypervisors that rely on hardware virtualization have to manage a different set of page tables for every isolated environment to provide memory protection.

To sum up, SICE’s SMM code base is better protected and less complicated than any microhypervisor. Moreover, this code does not provide any functionality other than the required isolation, which results in a very small code base. For instance, our prototype SMI handler consists of around 300 LOC (excluding cryptographic libraries). This is around an order of magnitude less than current microhypervisors (e.g., Trustvisor and NOVA). As discussed in [58], this is a significant difference when it comes to verification cost. For instance, using the industry rules-of-thumb of $10K per LOC for common criteria certification as a guideline for the cost of code verification, the cost for verifying 500LOC is $5M versus $100M for 10 KLOC.

Compatibility with Existing Software Systems: Unlike isolation techniques that monopolize the highest privileged execution level of the target platform (e.g., Trustvisor), SICE does not exclude running legacy workloads (e.g., a hypervisor with multiple VMs) on the same physical platform. In other words, a platform using SICE can offer isolated environments, and at the same time accommodate legacy virtualization software.

Feasible Hardware-based Isolation: SICE uses existing hardware features to provide the required isolation. We have successfully implemented a SICE prototype using a commodity
AMD processor. Thus, it does not require fundamental changes to current hardware architecture.

Moreover, our performance evaluation shows that SICE performs a secure context switching with an isolated environment that is four orders of magnitude faster than systems that rely on the late launch capability (e.g., Flicker [68]). It also uses multi-core processors to allow isolated workloads to run in parallel to a legacy hypervisor or OS. Thus, SICE avoids the two main drawbacks of using late launch, which are the high performance overhead, and the dedication of all system resources to only one isolated workload.

6.1.2 SICE Overview

SICE introduces novel techniques that allow the isolated workloads to run in parallel with a host OS or hypervisor. For convenience, we refer to the OS or hypervisor along with all the software running on it as the legacy host, a strongly isolated computing environment, which supports an isolated workload, as an isolated environment, and the code that manages the isolation between these environments as the SMI handler, or simply SICE.

The SMI handler, which represents the TCB of the isolated environments, resides inside the SMRAM. It is the only part of our framework that executes in the SMM. The SMI handler is responsible for two main tasks: 1) maintaining the memory isolation of the isolated environments, 2) securely initializing the isolated environments and attesting to their integrity. In SICE, these tasks require the SMI handler to run for a very short time.

An isolated environment is composed of two components: an isolated workload and a security manager. The isolated workload is a user-provided system that runs in the isolated environment. It can be any software, ranging from a single program (e.g., a program that manages secret keys) to a complete VM (e.g., a VM that runs a web server).

The security manager is a thin software layer that has limited functionalities such as handling exceptions and managing page tables. It is mainly responsible for confining the isolated workload. Due to the commodity hardware limitation on SMRAM size, SICE’s unique hardware-level isolation is not used to protect the legacy host from the isolated environments. Thus, SICE uses the security manager to prevent the isolated workload from accessing the memory of the legacy host. A separate copy of the security manager is generated by SICE for every isolated workload running on the system. Both the security manager and the isolated workload run after the system returns from the SMM.

Though SICE requires the legacy host to trust the security managers, it does not weaken the hardware-level isolation provided to the isolated workloads. SICE uses SMM to protect isolated environments from the legacy host. Even if a malicious workload (in one isolated environment) compromises its own security manager and consequently the legacy host, it will not be able to
compromise any other isolated environment running on the same platform.

SICE’s protection of the legacy host may appear to be equivalent to those provided by microhypervisor-based approaches such as NOVA. Indeed, the security manager, which is a thin privileged software layer, is similar to a microhypervisor in terms of its required tasks and code size. However, SICE also provides hardware-level, stronger protection for the isolated environments, which are not available in microhypervisor-based approaches.

SICE provides two operating modes: *time-sharing mode* and *multi-core mode*. In both modes, the SICE philosophy is based on using the isolated environments to run security sensitive workloads, while the legacy host is used for running less sensitive workloads and managing hardware peripherals. In a typical execution scenario, a communication channel is used so that the legacy host provides hardware services (e.g., networking) to the isolated environments. The communication channel can be established using a shared memory outside of the memory range protected by SICE. Communication channels are not controlled or managed by the SMI handler. Thus, the code responsible for managing the communication is not a part of the TCB.

In the time-sharing mode, time multiplexing is used to share the hardware platform between the isolated environments and the legacy host. During the environment switching, SICE guarantees a fresh start of the processor, complete memory isolation between these environments, and a timely switching that does not largely impact the performance.

In the multi-core mode, SICE assigns one or more processor cores to each isolated environment, while the other cores are used to run the legacy host. SICE guarantees isolation of both the processor cores and the memory dedicated to each of the concurrently running environments.

In both modes, SICE attests to the integrity of each isolated environment to remote users, while avoids revealing sensitive information about the workload to the legacy host.

The current SICE design relies on hardware features provided by AMD processors. Some of these features are not currently supported by Intel processors, particularly the ability to resize the SMRAM at runtime and defining a separate SMRAM for each processor core. Proposing alternative techniques to implement SICE on Intel platforms will require further research. We discuss these issues in Appendix 6.6.

We implement a prototype of SICE on an IBM LS22 blade server that uses AMD processors. We use our SICE prototype to run a complete Linux VM in the isolated environment. Our experimental evaluation shows that the time required to enter and exit an isolated environment using SICE is around 67 $\mu$s. We conduct experiments to evaluate the performance of the isolated VM. In the multi-core mode, SICE incurs a low (under 3%) overhead on VM operations that do not require frequent communication with the legacy host. The time-sharing mode shows a higher overhead (around 10%) due to the time for the context switching. A test dedicated to measure the performance of network emulation shows a higher overhead. However, this
overhead is expected to decrease using an optimized implementation of network emulation for isolated VMs.

6.1.3 Summary of Contributions

We make several technical contributions in this chapter:

- We provide a complete and feasible solution to share hardware resources with an isolated execution environment that does not rely on any host software. Instead, it relies on a TCB that is around an order of magnitude smaller than the state-of-the-art systems.
- We provide a novel technique that allows concurrent execution of the isolated environments with the untrusted host environment.
- We provide attestation to the integrity of the isolated workloads without revealing sensitive information about the workloads to untrusted host software.
- We implement a prototype of SICE on an AMD platform. We use this prototype to evaluate the performance overhead introduced by SICE.

The rest of this chapter is organized as follows. Section 6.2 discusses our assumptions and threat model. Section 6.3 presents SICE in detail. Section 6.4 presents our prototype implementation. Section 6.5 presents our experimental evaluation. Finally, Section 6.6 discusses SICE’s portability to Intel platforms.

6.2 Threat Model and Assumptions

Threat Model: SICE aims at defending against all malicious activities by software running in the legacy host that are targeted at compromising the isolation offered to the isolated environment (e.g., malicious activity that result from exploiting a vulnerability in the hypervisor). Specifically, SICE protects the isolated environment from all types of unauthorized memory accesses or any modification to its execution environment. The protection starts from the moment the isolated environment is initialized by SICE. Upon initialization, the initial image of the isolated workload, which is loaded by the legacy host, is measured so that SICE can further attest to its integrity.

We consider the following attacks out of the scope of this chapter: Attacks aimed at the availability of SICE (e.g., denying its network access), and those that directly exploit vulnerabilities of the isolated environment through legitimate communication channels. Such attacks are not specific to the isolation mechanism, and should be addressed by other techniques such as keep-alive messages and patching the vulnerabilities. Moreover, SICE is not responsible for securing the hardware services provided by the host. Thus, the isolated workload should use other techniques to achieve this objective (e.g., encrypting its network traffic).
Side-channel attacks are also out of the scope of this chapter. Adopting SICE prevents cache side-channels because the hardware automatically clears the cache upon entering and exiting from the SMM. However, these attacks in general are not unique to our approach and require further research.

**Assumptions:** We assume that our platform is physically secure (e.g., locked in a server room) so that an adversary cannot launch any hardware attack. Moreover, we assume that the target platform is equipped with trusted computing hardware, including the Core Root of Trust Measurement (CRTM) and Trusted Platform Module (TPM) [95]. This allows the attestation to the integrity of key software components (e.g., the SMI handler).

We assume that the SMM is properly isolated from other software running on the system and the hardware provides the SMRAM with proper isolation from all unauthorized memory accesses (e.g., cache poisoning, Direct Memory Access (DMA)). Recent incidents showed that attackers were able to subvert the SMRAM using cache poisoning [29, 102]. Fortunately, such attacks cannot be mounted on AMD platforms due to its SMRAM cache protection and can be easily defeated on Intel platforms using proper setting of the System Management Range Register (SMRR) [48].

### 6.3 SICE Design

In this section, we present the design of SICE. The objective is to enable hardware-level strongly isolated computing environments that run in parallel with the legacy host on the same hardware platform.

**Implementation requirements:** To implement SICE, the SMI handler needs to be modified to include SICE’s code. Recently, most hardware vendors use the BIOS to lock the SMRAM to prevent potential SMM misuse. Thus, SICE requires hardware vendors to allow adding its code to the SMI handler before locking the SMRAM.

The legacy host (e.g., the hypervisor) is required to add an interface that invokes an SMI to trigger SICE. Hardware management functions provided by the legacy host (e.g., hardware device drivers) should be modified to use this interface to provide its services to the isolated environment. Note that the legacy host may refuse to invoke an isolated environment or deny it some services, thus threatening its availability. However, these attacks are easily detectable by SICE and its remote users (e.g., from the lack of response of the isolated workload), and can then be thwarted by replacing the faulty legacy host.

66
**SICE Architecture:** Figure 6.1 shows the architecture of SICE. It consists of three components: The SMI handler, the security manager, and the isolated workload. The security manager and the isolated workload constitute the isolated environment. SICE enables the isolated environment and the legacy host to share the physical platform’s resources.

The security manager is a thin software layer that confines the isolated workload. It has limited functionalities such as handling exceptions and managing page tables. The initial image of the security manager should be loaded to the SMRAM, along with the SMI handler, upon the system initialization. Thus, we assume trust in both the SMI handler and the security manager based on the system’s trusted boot, which can be attested to later using the TPM.

The isolated workload is a user-provided system that runs in the isolated environment. It can be any software, ranging from a single program to a complete VM. The only restrictions on this software are that it does not use more than 4GB of memory (due to SMRAM constraints) and that it does not manipulate hardware peripherals directly.

When the isolated environment is not actively running, its code and data are securely stored in the SMRAM (as shown in Figure 6.1). However, the SMM is a limited execution mode that is not suitable for running the isolated environment. For instance, code execution inside the SMM is slower than the protected mode [10]. Moreover, some hardware functionalities (e.g., starting a hardware assisted VM) are not supported in the SMM. Thus, SICE only uses the SMM to prepare and enter the isolated environment. SICE uses novel techniques to move the security manager and the isolated workload out of the SMRAM after the isolated environment is initialized.

In the following, we discuss SICE in a *time-sharing mode*, where the legacy host and the isolated environment time share the physical platform. We then present the *multi-core mode*, where the legacy host and the isolated environment run in parallel using multi-core processors.
6.3.1 Time-sharing Mode

In the time-sharing mode, SICE provides two important features: (1) fast context switch between the legacy host and the isolated environment (in the magnitude of few tens of microseconds), and (2) large protected memory range for the isolated workload (up to 4GB). The fast context switch allows the isolated workload to receive its input data and send its output to and from the legacy host without posing significant overhead on the system performance. The large protected memory range enables the isolated environment to securely keep its state across the context switches.

Initializing the Isolated Environment: Both the security manager and the SMI handler are initialized when the physical platform is booted. When an isolated workload is ready to be started, the legacy host first loads the initial image of this workload to a specific memory range. Then, it triggers SICE using an SMI. The SMI handler then measures the initial image of the isolated workload and copies it to the protected SMRAM. Details on measuring and attesting to the isolated environment are discussed in Section 6.3.3.

Entering the Isolated Environment: Figure 6.2 shows the process of entering the isolated environment. Whenever the legacy host requires the isolated workload to run, it triggers another SMI. The SMI then switches the processor's execution environment to the SMM and the execution jumps to the SMI handler. The SMI handler then prepares the isolated environment by changing the saved processor state so that the security manager, instead of the legacy host, runs after the processor returns from the SMM. The SMI handler also stores the processor state.
of the legacy host so that it can resume execution after the isolated environment finishes its execution.

In the time-sharing mode, SICE gives the isolated environment full control of the physical platform as soon as it runs. Thus, it changes the processor state (e.g., interrupt descriptors, system call handlers) so that only the security manager will have control after the SMI handler returns.

Unfortunately, the SMM cannot change most of the critical processor states (e.g., the interrupt descriptor table (IDT) register and the CR3 register). To overcome this challenge, the SMI handler relocates the code pointed to by the current processor state through modifying the page tables. Since the CR3 register is not writable, the SMI handler modifies the first level page table pointed to by the CR3 so that it directly points to the security manager. It also flushes the Translation Lookaside Buffer (TLB) to avoid potential race conditions with the cached page tables.

Before the SMI handler returns, we need to ensure that both the security manager and the isolated workload are accessible to the processor after it returns from the SMM. As shown in Figure 6.1, both the security manager and the isolated workload are stored in the SMRAM before the isolated environment is entered. These two components need to be moved out of the SMRAM so that they can run after the processor returns from the SMM.

A straightforward solution is to copy the security manager and the isolated workload out of the SMRAM to an unprotected memory. However, this will introduce an unacceptable performance overhead.

To address this problem, we use the ability of AMD processors to resize the SMRAM during the system runtime. This feature, discussed in Section 2.3, relies on a password-protected mechanism to clear the SMM lock so that the `SMM.Addr` and the `SMM.Mask` registers can be updated.

Upon the initialization of the system, SICE generates a 64-bit random number, sets it as the password for locking/unlocking the SMRAM, and stores it inside the SMRAM. Before entering the isolated environment, the SMI handler uses this password to unlock the SMRAM and modify its protection range (by modifying the `SMM.Addr` and the `SMM.Mask` registers) to exclude both the security manager and the isolated workload. Thus, they can be accessed by the processor after the SMI handler returns.

As shown in Figure 6.2, the SMRAM protection remains for the SMI handler while the isolated workload runs. This guarantees that the SMI handler (including the SMRAM password) can never be accessed by any other software running on the system, including the isolated workload.

A subtle issue needs to be clarified. To guarantee full control over the system, the isolated environment should run in the highest privileged level after the SMI returns. However, this
requires the legacy host to trigger the SMI from within the highest privileged level. In other words, the SMI cannot be triggered by a guest VM. Otherwise, the isolated environment will return to the context of a guest VM controlled by the legacy host.

To verify that the SMI is triggered by the highest privileged level, SICE requires the legacy host to disable virtualization by clearing the $\text{SVME}$ bit in the $\text{EFER}$ register before triggering the SMI. The SMI handler verifies that this bit is clear. Otherwise, it will not enter the isolated environment and keep the SMRAM protection for its memory.

**Managing the Isolated Workload:** Since the isolated workload is provided by the user, it is not trusted by SICE. Hence, it should not be allowed to tamper with the hardware configuration or access memory regions that belong to the security manager or legacy host.

Unfortunately, commodity hardware architecture cannot assign more than 4GB of memory to the SMRAM. Thus, using the SMRAM to protect the legacy host memory from the isolated workload will not be a feasible solution due to its restriction on the memory capacity of the legacy host.

To address this challenge, the security manager plays the role of a hypervisor and runs the isolated workload in the context of a guest VM. Therefore, the isolated workload’s execution environment is restricted so that it cannot execute privileged instructions. Moreover, the security manager crafts the page tables of the isolated workload so that it can access a limited specific range of physical memory. Thus, the isolated workload cannot access any memory range that belongs to either the security manager or the legacy host.

**Exiting the Isolated Environment:** To allow the legacy host and the isolated environment to efficiently share the hardware resources, SICE provides a technique to securely and properly exit the isolated environment and return the execution back to the legacy host.

To return back to the legacy host, the isolated environment triggers an SMI. The SMI handler then: (1) Clears all general purpose registers, (2) flushes all cache levels, (3) uses the SMM password to change the SMRAM range so that it covers both the security manager and the isolated workload, and (4) restores all changes done to the legacy host page tables before the isolated environment was initialized.

The first two steps ensures that no sensitive data is leaked to the legacy host. The third step ensures the protection of the isolated environment’s memory. Finally, the last step ensures that the legacy host resumes its operations correctly.

**Terminating the Isolated Workload:** When the isolated workload finishes execution, it needs to be securely terminated so that no confidential information are leaked to the legacy host. Thus, isolated workload termination requests must be forwarded to the SMI handler,
which securely erases the memory belonging to the isolated workload and then removes it from the protected SMRAM.

6.3.2 Multi-core Mode

The SICE multi-core mode enables the legacy host and the isolated environments to run in parallel on different processor cores to better utilize the hardware resources.

Before presenting our solution, we first give some background information on multi-core processors. As shown in Figure 6.3, multi-core processors are equipped with one or more processing nodes. Each node has its own memory control hub (north bridge) and one or more processor cores. North bridge configuration registers can be accessed by any core on any node. On the other hand, each processor core has its own general-purpose registers, MSRs, APIC, and two levels of cache. These components can only be accessed by their designated core. Among these core-dependent components are the $\text{SMM\_Addr}$ and $\text{SMM\_Mask}$ MSRs that define the SMRAM memory range. Thus, each processor core can have its own protected SMRAM memory range that is independent from the SMRAM memory range defined on other processor cores that share the same node. SICE utilizes this architecture to provide the necessary protection for the isolated environment in the multi-core mode.

For simplicity, we assume a platform with two cores in our discussion: the host core that runs the legacy host, and the isolated core that runs the isolated environment. However, the same techniques can be used to assign more than one core to either the legacy host or the isolated environment. In general, two processor cores that belong to different execution environments can be either co-located on the same processing node or located among different nodes.

In the multi-core mode, SICE initializes the isolated environment in the same way as the time-sharing mode (i.e., by triggering an SMI in the legacy host). The SMI switches the
The SMI handler performs a different task on each processor core. On the isolated core, the SMI handler prepares the isolated environment and returns to the security manager using the techniques described in Section 6.3.1. Meanwhile, on the host core, the SMI handler returns directly to the legacy host so that it can resume its operations at the same time.

Since both the legacy host and the isolated environment run in parallel on different processor cores, no context switching between these environments is needed.

To ensure the isolation between the legacy host and the isolated environments, SICE prevents the legacy host from affecting the execution of the isolated environment, and the isolated workload (excluding the security manager, which needs to be trusted by the legacy host) from affecting the execution of the legacy host. In particular, we need to prevent the interference through explicit inter-core communication as well as those through modifying each other’s memory.

**Handling Inter-core Communication:** Communication between processor cores, which is done through Inter-Processor Interrupts (IPIs), can modify the execution environment on the recipient code by changing the execution path. This can pose a risk on the integrity of the isolated environment. Therefore, all IPIs should either be blocked or securely handled by the recipient core.

There are two types of IPIs: (1) Maskable IPIs that can be blocked by the recipient core’s Advanced Programmable Interrupt Controller (APIC), and (2) non-maskable IPIs (e.g., NMI, SMI, startup, and INIT) that cannot be blocked.

Upon initializing the isolated environment, the SMI handler disables all maskable interrupts on the isolated core so that the host core cannot interfere with the isolated core. The security manager, which runs as a thin hypervisor on the isolated core, keeps these interrupts disabled.

On the other hand, non-maskable interrupts cannot be blocked. Moreover, a specific non-maskable interrupt, the startup interrupt, resets the processor core to start execution at a low physical memory address. Intuitively, this address can be modified by the host core to alter the course of execution on the isolated core.

To overcome this problem, SICE relies on the Global Interrupt Flag (GIF) introduced by AMD. When the GIF is clear, all interrupts, including non-maskable ones, are ignored or held pending. The security manager thus clears the GIF of the isolated core when the isolated environment is entered, and sets it only after the isolated environment completes its execution and all memory protection measures are taken.

The security manager then runs the isolated workload in the guest VM mode. Upon entering the VM mode, both global interrupts and maskable interrupts are re-enabled. Hence, the isolated workload can receive all interrupts including both IPIs and local processor interrupts.
Figure 6.4: Double view of memory from different processor cores. From the host core’s view, the isolated environment is part of the SMRAM. From the isolated workload’s view, the SMI handler is protected by the SMRAM and the legacy host is protected by the security manager via hardware assisted paging.

However, the received interrupts will cause a VM exit and jump to the security manager rather than modifying the execution environment.

Global interrupts are automatically disabled again before exiting the VM mode and jumping to the security manager. The security manager identifies the reason that causes the VM exit. If it is an interrupt that aims to modify the execution environment (e.g., INIT interrupt), then it indicates a malicious activity from the host core and all memory protection measures are taken. However, if the VM exit is caused by a local interrupt or an IPI, the interrupt is forwarded back to the isolated workload. This architecture allows the isolated workload to use IPIs as method of signaling, which is required to build a communication channel with the legacy host. This channel can be used to provide hardware services offered by the legacy host to the isolated workload.

On the other hand, SICE relies on hardware virtualization to prevent the isolated workload from perturbing the legacy host. The isolated workload, which runs in a guest VM, is not allowed to directly access the APIC to send any interrupts that can perturb the execution of the host core.

Memory Isolation: To prevent the legacy host from accessing the isolated environment’s memory, SICE relies on a novel protection called the memory double-view technique, shown in Figure 6.4. This technique relies on the fact that AMD processor’s SMRAM is defined based on the core-dependent MSRs. Thus, different cores can view the SMRAM differently depending on the values of their own SMM.Addr and SMM.Mask registers.

As shown in Figure 6.4, each processor core has its own view of the physical memory. From the host core’s perspective, the isolated environment uses a physical memory that lies in the SMRAM memory range. Hence, the legacy host cannot access such memory due to the SMRAM protection. From the isolated core’s perspective, the isolated environment lies within
a normal memory region that is not part of the SMRAM. Hence, the isolated environment can run normally on the isolated core.

SICE also prevents the isolated workload from accessing the legacy host memory. The security manager, which plays the role of a hypervisor, restricts the isolated workload to its assigned physical memory. Nevertheless, a shared memory, which does not fall in the host core’s SMRAM, is mapped to allow communication between the two environments.

6.3.3 Attestation and Secure Communication

Cloud computing users are required to trust the environment that runs their workloads. In SICE, attesting the integrity of the isolated environment is complicated by the fact that the legacy host has full access to all hardware peripherals and communication channels. In this section we provide a three-step solution to address this challenge.

Attesting to Integrity of SICE: SICE requires the platform to use standard trusted boot [94]. After a typical trusted boot, the TPM securely stores the measurement of the boot process, which includes the SMI handler, the code image of the security manager and the code that loads them and locks the SMRAM. The measurement is stored in TPM special registers that cannot be erased by malicious software.

The TPM can further use its private Attestation Identity Key (AIK) to attest to the integrity of this measurement to remote users. Unlike other system software that is continuously interacting with potential attackers, the SMI handler cannot be modified by any code running on the system and the trust in it can be maintained.

Attesting to Integrity of Isolated Workload: In turn, the SMI handler attests to the integrity of the isolated workload. It measures the isolated workload before it is first invoked and stores the measurement in the secure SMRAM.

To enable the attestation, we adopt an approach we previously used in HyperSentry [10]. Specifically, SICE generates a public/private key pair during system boot. The private key is securely stored in the SMRAM, and the public key is securely stored inside the TPM.

To attest to the integrity of the isolated workload, the SMI handler signs the measurement of the initial workload image using its private key. The remote user accepts the measurement output only if the private key matches the public key stored in, and attested to by, the TPM.

Secure Communication with the Isolated Environment: SICE allows the establishment of a secure communication channel between the isolated workload and its remote owner using standard cryptographic techniques. To enable the remote owner to authenticate the isolated workload, SICE provides a public/private key pair for the isolated workload. The public key is
sent to the user as a part of the workload attestation evidence signed by the SMI handler. The private key is directly provided to the isolated workload. Thus, it is never exposed to potentially malicious code running inside the legacy host. On the other hand, the isolated workload should include the public key of its remote owner as a part of its initial image.

With both the security manager and the remote owner being able to validate each other’s public keys, we can easily modify, for example, SSL for secure communication without leaking information to the (untrusted) legacy host.

6.3.4 Security Analysis

Now we discuss the security of the isolated environment, including its confidentiality, integrity, availability, and TCB. We also discuss the security implication of using the SMM.

Confidentiality: To protect the confidentiality of the isolated environment, SICE prevents potential attackers from accessing its memory. SICE uses the SMRAM to achieve this objective. The hardware protects the SMRAM from access requests made by both the CPU and direct memory access (DMA) capable devices.

In the time-sharing mode, SICE modifies the SMRAM memory range according to the running environment. Whenever the legacy host is running, SICE extends the SMRAM to include the memory of the isolated environment. Thus, this memory is protected from all memory requests.

When the isolated environment is triggered, SICE takes two security measures to guarantee its confidentiality. First, SICE assures that the isolated environment fully controls physical platform. Hence, the legacy host is not active to threaten the isolated environment, given that modern processors do not use cached memory upon entering the SMM to avoid cache poisoning attacks [6,48]. Second, SICE uses hardware DMA exclusion (e.g., AMD’s DMA exclusion vector (DEV) [6] and Intel’s VT-d [47]) to prevent DMA capable devices, which could be maliciously programmed, from accessing the protected memory.

In the multi-core mode, SICE’s memory double-view technique provides the needed protection. According to architecture manuals, memory access requests made from a processor core are checked against its own SMRAM before being routed. Hence, the host core will not be able to access the protected memory that falls within its own SMRAM range. The same is true for requests to update or retrieve cache entries. Hence, cache poisoning is not possible.

In the multi-core mode, SICE cannot rely on hardware DMA exclusion because its control registers exist in the memory control hub, which is accessible by all processor cores. Thus, SICE relies on the SMRAM protection to prevent DMA access to the isolated environment’s memory. Typically, the memory control hub does not allow DMA access to the SMRAM. However, the AMD hardware architecture manual does not precisely define which processor core’s SMRAM
range is used to prevent DMA access, particularly when different SMRAM ranges are defined on different cores. Thus, implementing SICE on a specific platform requires verifying which SMRAM range is protected from DMA. The specific processor core(s) that defines this range is the same one that should be used as the untrusted host core. As shown in figure 6.4, the host core’s SMRAM memory range includes the whole memory range used by the running, one or more, isolated environments.

A final threat to the confidentiality of the isolated environment is brute force attacks against the SMM password. For each processor core, SICE generates a 64-bit random value to be used as the SMM password. Thus, it is computationally infeasible to break this password. The random password generation can be done using the TPM.

**Integrity:** In the time-sharing mode, the legacy host cannot threaten the integrity of the isolated environment because they do not run concurrently. In the multi-core mode, the host core cannot access the registers, MSRs and APIC of the isolated core. However, the host core can still modify system-wide configurations that rely on shared resources (e.g., the memory control hub and the IO control hub). Next, we prove that such configurations can only perturb the isolated environment, for example, by rebooting the system, without threatening its confidentiality or integrity.

According to the Advanced Configuration and Power Interface (ACPI) specification [42], processor cores keep their state as long as the processor runs in the S0 or S1 power modes. Thus, the isolated core will keep running the isolated environment as long as either S0 or S1 state is maintained. On the other hand, all other ACPI states will cause the processor core to lose its state and resume execution from the non-volatile memory (i.e., the BIOS). Since the BIOS belongs to the trusted computing base, changing the ACPI state will be detected by SICE. In all cases, the ACPI implementation of the target platform needs to be carefully reviewed to securely implement SICE.

As presented in Section 6.3.2, attacks that use IPI between processor cores are thwarted using APIC setting and the GIF of the processor core that runs the isolated environment.

**Availability:** SICE does not provide protection against attacks that target the availability of the isolated environment. Example attacks include perturbing the isolated environment through system reboots or denying it network access. However, this type of attacks is easily detectable by SICE and its remote users (e.g., from the lack of response of the isolated environment), and can be easily thwarted by removing the malicious code from the legacy host.

**The TCB of SICE:** SICE aims to minimize and enhance the protection of the TCB of the isolated environment so as to maximize its security.
The TCB of the isolated environment consists of the hardware, the BIOS, and the SMM. Using the SMM gives SICE two main advantages over microhypervisor-based isolation. First, SICE TCB enjoys the hardware protection provided for the SMRAM. Second, the SMM’s attack surface is much smaller than that of microhypervisors. Thus, The TCB of the isolated environment is better protected and less complicated than that of microhypervisors. Section 6.1.1 compares the TCB of SICE with that of microhypervisors in detail.

Trusting the BIOS is required to start the trust chain based on the static root of trust management (SRTM) technique. However, recent incidents [62] show that the SRTM can be compromised. Nevertheless, SICE can adopt the Dynamic root of trust management (DRTM) to start the trust chain, which consequently eliminates the BIOS from the TCB. SICE can use the DRTM to invoke a trusted code that securely initializes the SMRAM, given that it will not be locked by the BIOS. A similar technique was used by Trustvisor [67] to initialize its TCB.

SICE requires the legacy host to trust the security manager(s). However, this does not make the security guarantee provided to the legacy host weaker than microhypervisor-based approaches such as NOVA, which assume trust in a thin hypervisor with typical duties. Indeed, SICE and the microhypervisor-based approaches provide similar security guarantees for the legacy host, while SICE additionally also provides a stronger protection for the isolated environments, as discussed in Section 6.1.2.

**SMM Security:** Despite being an integral part of the TCB of SICE, the SMM was not designed to provide security services. It was originally designed to create a shadow environment to run hardware management tasks. In the following, we discuss some of the important issues that need to be considered before deploying SICE, or any other system that relies on the SMM, in a production environment.

*Legacy SMM Tasks:* Intuitively, the TCB of SICE includes any code that exists in the SMRAM. To keep the TCB minimal, legacy system management tasks need to be eliminated from the SMM. This does not mean that SICE needs to completely eliminate them from the system. Instead, SICE can simply forward specific SMIs to (possibly relocated) legacy SMM code after taking the required memory protection measures.

*Previous SMM Attacks:* As mentioned in Section 6.2, there have been attempts to subvert the SMRAM using cache poisoning [29,102]. Although these attacks can be easily prevented by using proper hardware configuration, it shows that hardware vendors should undergo a rigorous review of the SMM security to avoid any further security problems.

It is worth mentioning that the implication of SMM attacks is beyond the systems that rely on the SMM for security. It is shown in [32] that a subverted SMM can be used to host a stealthy rootkit that undermines the security of any system. In general, trusting the underlying hardware, including the SMM, is imposed on all software systems.
Hardware Vendors Cooperation: As mentioned in Section 6.3, implementing SICE requires hardware vendors to allow adding its code to the SMI handler before locking the SMRAM. In fact, we advocate that SICE should be entirely implemented by hardware vendors. In this case, the system BIOS will be responsible for initializing SICE upon booting the system. This guarantees that SICE will always be compatible with the specification of the underlying hardware.

6.4 SICE Prototype: An Isolated VM

In this section, we present a SICE prototype implemented on an IBM LS22 blade server, which is equipped with two 2.7GHz AMD Opteron 10h family quad-core processors. We use an Ubuntu 9.10 Linux as the OS. In our prototype, we replace the original SMI handler with SICE’s SMI handler. Implementing the core functions of SICE’s SMI handler, which are to prepare and enter the isolated environment, requires around 300 LOC (excluding cryptographic libraries).

As mentioned in Section 6.1.2, the isolated environment can support running any software, ranging from a single program to a complete VM. To run a single program inside SICE, it needs to be instrumented so that it directly runs on top of the security manager. The required instrumentation is similar to that required by previous research on running isolated security sensitive code (e.g., Flicker [68], Trustvisor [67]).

In our prototype, we use SICE to run a Linux VM, rather than running an instrumented single program. This is to demonstrate the flexibility provided to the isolated environment. Running a whole VM requires the ability to support a diverse set of applications, frequent context switching, and a large range of protected memory.

Figure 6.5 shows our SICE prototype. The security manager plays the role of the hypervisor in the isolated environment. It uses hardware assisted virtualization to manage and run the VM as the isolated workload. All hardware management functions are delegated to the legacy
host, specifically, a modified version of Qemu/KVM [66] running in the legacy host. Qemu is a program that emulates hardware peripherals. KVM is a kernel module that manages other VM operations like scheduling and memory management.

We modified KVM so that it uses a shared memory to send the required VM configurations to the security manager. The same shared memory is used by the security manager to pass the information required to request hardware services from the legacy host. This architecture is mainly chosen to simplify the implementation efforts. In our prototype, the security manager is composed of around 2.1 KLOC, which is comparable to the size of current microhypervisors.

Our implementation provides both network interface and serial port emulation using Qemu. Networking is necessary to allow the VM to communicate with remote users, while the serial port is used as a console for debugging.

Our prototype does not support graphic display emulation, which is not necessary for cloud computing applications. Moreover, we do not support disk drive emulation because our experiments do not need a permanent storage.

Supporting a disk drive emulation is straightforward based on the technique used to implement the network interface emulation. However, the main question is how to secure the disk access from a compromised legacy host. The answer is dependent on the isolated workload rather than the isolation mechanism. For instance, security sensitive applications can either use full disk encryption (similar to Bitlocker [33]), or selectively encrypt secret files only (e.g., a file that contains user passwords). Other applications only need to keep secret data in memory without a permanent storage (e.g., web servers that process online purchases without storing customer credit card numbers).

In the following, we present more details about our prototype implementation.

6.4.1 Preparing and Initializing the VM

As discussed in Section 6.3.1, the legacy host provides SICE with the initial VM image, composed of the VM kernel and the initial ram disk. The SMI handler copies this image to the SMRAM. KVM also places the required VM configuration parameters (e.g., RAM size) in the shared memory then triggers an SMI to initialize the isolated workload. As mentioned in Section 6.3.3, the SMI handler measures the initial VM image to attest to its integrity to the remote owner of this VM. The integrity evidence should be extended to include the VM configuration parameters passed by KVM. Some of these parameters (e.g., the VM execution entry point, initial register values) are critical to its integrity.

The security manager prepares the VM page tables and its virtual machine control block (VMCB), based on the provided configuration parameters. The page tables are crafted using AMD’s nested page tables (NPT) [6], which adds another layer to the virtual-to-physical
memory translation.

6.4.2 Handling VM Exits

Certain events force the VM to exit its operations (e.g., external interrupts, page faults). These events need to be handled by the hypervisor.

We forward most of the VM exit events to KVM in the legacy host. Nevertheless, the security manager handles some specific VM exits that do not require much computation to avoid unnecessary context switching. Among these VM exit events are control register accesses and requests to execute privileged instructions like CPUID or INVD.

Other VM exits (e.g., writing to an IO port, accessing an IO memory) are directly forwarded to the legacy host. To preserve the VM confidentiality, the security manager should only send information that is necessary for handling the VM exits (e.g., VM exit reason and error code). This information is sent to the KVM in the legacy host through an established communication channel between the isolated environment and the legacy host.

6.4.3 Communication with the Legacy Host

Establishing a communication channel with the legacy host is not managed by the SMI handler. Instead, it is directly managed by the security manager and the legacy host. As discussed in Section 6.3, a shared memory that is outside the SMRAM protection is used to establish the communication between the legacy host and the isolated workload.

Signaling is required between the two environments to send notifications that data is placed into the shared memory. In the time-sharing mode, we use the context switching as the method of signaling. In the multi-core mode, IPIs between the host and the isolated cores are used for signaling, as discussed in Section 6.3.2.

6.4.4 Using Hardware Peripherals

In our prototype, hardware peripherals in the isolated workload are emulated using Qemu in the legacy host. There are three main methods to control a hardware peripheral: IO ports, IO memory, and Direct Memory Access (DMA). When the VM accesses an IO port or an IO memory, a VM exit occurs and the control is transferred to the host to emulate the hardware access.

DMA requests work differently because DMA is supposed to directly read or write physical memory. In legacy VMs, Qemu is granted full access to the VM physical memory to emulate DMA accesses. However, since Qemu is located in the legacy host, SICE prevents it from directly accessing the memory of the isolated environment. Thus, our prototype modifies Qemu to send a request to the security manager with the address, size, and type (read or write) of
the emulated DMA. The security manager in turn copies the required memory between the VM memory and the shared memory. To preserve the confidentiality of the isolated workload, the security manager only allows Qemu to retrieve or modify VM memory areas assigned to DMA operations.

In our prototype, the security manager handles DMA access to simplify our implementation. Nevertheless, this operations should be directly handled by the isolated VM by allowing it to directly access a part of the shared memory, which will reduce the tasks required from the security manager, and consequently reduce its code size.

6.4.5 Attestation

As mentioned in Section 6.3.3, the SMI handler should be a part of a trusted boot process. However, the LS22 servers, used for our implementation prototype, are not equipped with a TPM. Due to the lack of a TPM, the attestation process is not included in our prototype. 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.

It is worth mentioning here that implementing these cryptographic operations may increase the size of the code base of SICE. For instance, a typical SHA1 library is around 120 LOC. Other cryptographic functions (e.g., generating an RSA signature) can be done using the TPM.

6.5 Experimental Evaluation

We perform a set of experiments to evaluate the performance of SICE. There are two anticipated sources of performance overhead associated with SICE. The first is the direct overhead resulting from entering/exiting the isolated environment. The second is the indirect overhead that results from the cache and TLB flushing required for SICE operations. On the other hand, running the isolated environment outside the SMM avoids any execution slow-down and is not anticipated to cause any performance overhead, compared to running the same workload without SICE.

In the rest of this section, we present a measurement of the anticipated overhead. First, we measure the execution time needed for a full context switching to and from the isolated environment. Second, we use the SICE prototype (See Section 6.4) to compare the performance of the isolated guest VM with the same VM running without SICE isolation.

6.5.1 SICE Execution Time

We measure the execution time needed to perform each of the four major steps of entering and exiting the isolated environment. The measured steps include: (1) triggering the SMI,
(2) preparing the isolated environment by the SMI handler, (3) entering the isolated environment, and (4) returning to the legacy host. These measures are obtained from the average of 100 runs.

To precisely measure the end-to-end execution time of each step, we use the \texttt{RDTSC} instruction to read the processor’s Time Stamp Counter (TSC). We then convert cycles to microseconds based on the TSC speed (2.7GHz in our experimental platform).

\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|}
\hline
Operation & Time (in $\mu s$) & Std. Dev. \\ \hline
Triggering an SMI & 6.8 & 0.074 \\ \hline
Preparing the isolated env. & 20.7 & 0.155 \\ \hline
Entering the isolated env. & 9.3 & 0.233 \\ \hline
Exiting the isolated env. & 30.1 & 0.644 \\ \hline
Total ($\approx$) & 67 & \ \hline
\end{tabular}
\caption{SICE execution time}
\end{table}

Table 6.1 shows the experimental results for the time-sharing mode. Triggering an SMI and switching the processor context from the legacy host to the SMI handler needs an average of 6.8 $\mu s$. The SMI is invoked by the local APIC of the processor core by sending an IPI to all cores on the system.

The next step, which is to prepare the isolated environment, needs an average of 20.7 $\mu s$. The latency of this step is relatively high given that this step only requires changing a few entries in the page tables, modifying the interrupt vector table descriptor, and verifying the values of several registers and MSRs. We anticipate that this relatively high latency is caused by a slower processing speed in the SMM. A similar observation was made in [10].

The next step is entering the isolated environment. In this step, we execute the “return from SMM” instruction (\texttt{RSM}) to jump to the security manager. The time needed for this step is 9.3 $\mu s$, which is relatively high. We anticipate this latency is due to the change of the processor execution environment, particularly the page tables, which leads to invalidating all cache and TLB entries.

Finally, the time needed to return from the isolated environment to the legacy host is 30.1 $\mu s$, which is similar to executing all the previous steps in the reverse order (i.e., from the isolated environment to the legacy host). In total, the end-to-end time required to enter and exit the isolated environment is 67 $\mu s$.

In the time-sharing mode, the end-to-end execution time overhead obtained in this experiment occurs on every context switch between the isolated environment and the legacy host. A
context switch is required every time the isolated environment needs a service from the legacy host. In contrast, it is a one-time overhead in the multi-core mode.

### 6.5.2 Isolated Environment Performance

Our next experiment is to evaluate the overall performance overhead on the isolated workload. We run this experiment using the same system configuration of our prototype system, discussed in Section 6.4. In our experiments, we assign a 256 MB to the VM. Our VM runs Linux kernel v2.6.28 and a ram disk that is equipped with BusyBox v1.10.2 [18]. (BusyBox is a program that combines common utilities into a single executable.)

To evaluate the performance of the guest VM, we assemble a set of tests and run each test on the same guest VM in three different environments: (1) unmodified Qemu/KVM using one processor core, (2) SICE time-sharing environment using a single processor core, and (3) SICE multi-core environment using two processor cores, used as the host core and the isolated core. In the multi-core test, we do not run any program other than Qemu/KVM on the host core, so that we can get a precise measurement of the performance overhead.

To measure the performance overhead, we use Qemu to emulate a serial port for the guest VM. We then configure the guest VM to use this serial console to accept shell command. Our performance measurement is the execution time needed for the guest VM to complete each of these tests. The execution time is calculated by using the \texttt{RDTSC} instruction in the legacy host. The time measurement is taken just before sending the last letter of the shell command (the carriage return) and right after the first response is received (echoing the carriage return).

To calculate the performance overhead of each test, we first measure $T_K$, the time needed to run the test using unmodified Qemu/KVM. Afterward, we measure $T_S$, the time needed to run the same experiment using SICE. The performance overhead is then calculated using the equation $(T_S - T_K)/T_K \times 100\%$.

We selected our test cases in three categories: The first is user level programs to test the user level operations of guest VMs running with SICE. The second is a group of tests that test the throughput of the guest VM kernel. The third tests the performance of the emulated network interface.

In the first category, we run three tests. The first measures the latency of copying a 2.1 MB file. Our prototype system does not support hard drive emulation. Thus the file is copied inside the virtual file system that represents the initial ram disk. Our main objective of this test is to estimate the impact of SICE on the guest VM memory operations, because it is basically a copy between two locations in memory rather than an actual disk access. The next two tests in the same category use the programs \texttt{gzip} and \texttt{gunzip} to compress and decompress the same file. These tests aim to test the impact of SICE on the guest VM’s user level computations.
The final results of these tests are obtained as an average of 100 runs.

In the second category, we measure the guest VM kernel responses to three main operations: *fork*, *getpid* and *insmod*. *fork* and *getpid* are tested using a custom program that directly calls these system calls 10,000 times and measures the average response time. For the *insmod* test case, we use the *insmod* program to insert a 16.4 KB loadable kernel module in the guest kernel. This test is repeated 10 times, and all test runs show very little variance.

In the last category, we run a single test that uses *wget* to retrieve a 156.6 KB file from an Apache server running in the legacy host. The test is repeated 10 times.

Figure 6.6 shows the results of our experiments. In the multi-core mode, all tests, except for *wget*, showed a slight performance overhead under 3%. We expect that the slight performance overhead is due to the time required to copy information to and from the shared memory between the legacy host and the security manager.

The *wget* test shows a 17% performance overhead. However, a significant portion of this overhead is due to the non-optimized implementation of the Qemu network emulator for SICE. When a packet is ready to be sent through the network, the VM sends an IO command to the network adapter to inform it that a packet is ready. In normal operations, Qemu first receives this IO command, and then directly accesses the VM physical memory to emulate the DMA read of the packet. In our prototype, Qemu does not have direct access to the physical memory of the guest VM. Thus, we modified Qemu to send DMA requests to the security manager, which copies the packet from the protected guest VM memory to the shared memory so that Qemu can process it correctly. Similarly, an extra step is necessary when the VM receives a packet.

Hence, our prototype requires an extra communication step between Qemu in the legacy host and the security manager for every network packet. Intuitively, this extra communication step requires an extra context switch from the legacy host to the isolated environment in the time-sharing mode or an extra inter-process communication between the host core and the isolated core in the multi-core mode.
This extra communication step is the cause of the high overhead in the `wget` test. To support this claim, we measured the number of exits to and from the isolated environment during this test. In normal operations, there was an average of 1,249 guest VM exits that required hardware emulation by KVM and/or Qemu. In SICE’s time-sharing mode, there was an average of 1,877 context switches between the isolated environment and the legacy host. There was a similar number of messages passed between the host core and the isolated core in the multi-core mode. This 50% increase in the communication time between the two environments explains the high overhead in this test.

To avoid this overhead, the VM device driver can be modified so that it uses the shared memory for the network adapter’s DMA operations. Thus, the Qemu emulated driver would be able to directly read the passed packets without sending an extra request to the security manager.

The time-sharing mode showed a higher performance overhead, which is expectable due to the 67 µs needed to switch between the host and the isolated guest VM. The tests that show higher overhead, such as `gzip` and `wget`, are those that require a higher number of switching between the host and the isolated environment.

It is also noticeable that the time-sharing mode showed a higher overhead (around 40%) in the `wget` test. Receiving the target file in the normal VM operations needs 0.532 seconds compared to 0.739 seconds in the time-sharing mode. As mentioned above, this test requires 1,877 context switches from the isolated environment to the legacy host. Given that each context switch requires an average of 67 µs, the context switch overhead in this test was 0.126 seconds. This test clearly shows that the time-sharing mode is not suitable for operations that are IO intensive. It will be rather useful to run programs that require an enhanced isolation and a smaller IO footprint (e.g., a program that processes secret key operations).

### 6.6 Portability to Intel Platforms

In this section, we discuss the portability of SICE to Intel platforms. Although SMM is supported by both AMD and Intel, SICE uses some hardware features that are only provided by AMD. In the following, we discuss these features and propose alternative techniques for Intel platforms. Implementing SICE on Intel platforms using these techniques needs further research, particularly to achieve both the security and performance required from this system.

**SMRAM Size and Location Modification:** After the SMRAM is locked, Intel platforms do not allow runtime change of its size and location. This restriction has a direct impact on our techniques, which rely on changing the SMRAM size at run time to enable fast context switching to the isolated environment. However, some other Intel features can be used for
SICE. For instance, new Intel hardware is equipped with special instructions to do a high throughput cryptographic operations using the Advanced Encryption Standard (AES). In the time-sharing mode, this feature can be used to encrypt and authenticate the memory of the isolated workload while the legacy host is running, and decrypt and verify the same memory in the isolated environment. Thus, the SMRAM can only be used as a permanent storage of encryption keys. Further research is needed to evaluate the feasibility and performance of this approach.

**Memory Double-view:** Intel platforms define the TSeg SMRAM in its memory controller hub, which is shared among multiple cores. Unless the implementation is modified to move the SMRAM definition to a core specific location, our memory double-view technique will not be able to isolate individual Intel processor cores.

Nevertheless, current Intel implementation can apply this technique on processor nodes as they do not share the same memory control hub, allowing a more coarse grained isolation of the resources on physical platforms.

**Multi-core Protection:** Intel does not provide a capability that can disable all system interrupts. As mentioned in Section 6.3.2, we rely on this capability to prevent the host core from using the startup IPI to modify the execution environment of the isolated core. However, the startup IPI can only set the instruction pointer to an address in the lowest 1MB of the physical memory. This address range can be easily modified to point to a non-writable memory chip.
Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this dissertation, we addressed the following three problems that face the security and trustworthiness of cloud computing: 1) verifying the integrity of cloud computing guest VMs, 2) verifying the integrity of hypervisors, and 3) strong isolation of security-sensitive workloads.

First, we developed HIMA, a hypervisor-based agent that measures the integrity of guest VMs that run on top of the hypervisor. HIMA is located in the hypervisor, and performs a comprehensive “out-of-the-box” measurement of the guest VMs, including both the guest OS kernels and their applications. HIMA provides two complementary mechanisms: active monitoring of critical guest events and guest memory protection. The former guarantees that the integrity measures are refreshed whenever the program layout in a guest VM changes (e.g., creation or termination of processes), while the latter ensures that the integrity measurement of user programs cannot be bypassed without HIMA’s knowledge. Using these two mechanisms, HIMA ensures TOCTTOU consistency of the integrity measurement of user programs. We have implemented a prototype of HIMA based on the Xen hypervisor. Our security analysis and extensive experimental evaluation indicate that HIMA provides a practical solution to integrity measurement.

Second, we developed 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.

Finally, we developed SICE, a framework that provides a hardware-level strongly isolated computing environment for commodity x86 platforms. The key contribution of SICE is a set of novel techniques that provides protection for an isolated environment without relying on any software in the host environment. SICE provides a set of unique capabilities to the isolated environment, including (1) fast context switch to and from the isolated environment in the time-sharing mode, (2) protected concurrent execution with the host environment in the multi-core mode, and (3) attestation to its integrity. Our prototype implementation showed that SICE is a practical solution for commodity x86 hardware and software architectures. Its unique ability to run isolated workloads concurrently with the untrusted host environment provides a cost-effective solution for compute clouds to host sensitive workloads without using dedicated hardware platforms to such workloads.

7.2 Future Work

Despite the research accomplishments done by the three systems presented in this dissertation, there still remain several problems to be addressed. As a future work, we propose a set of enhancements that would extend the security and the applicability of each of our systems.

**Integrity Measurement of Guest VM:** In its current form, HIMA provides both active monitoring and TOCTTOU consistency for integrity measurement. However, HIMA’s memory protection, which guarantees the TOCTTOU consistency, is limited to user processes and cannot protect the kernel code. As discussed in Chapter 4, this limitation results from the fact that almost all major OS kernels need mixed code and data pages to function correctly. This behavior contradicts with HIMA’s memory protection mechanism, which requires W⊕X mapping to protect the kernel code segment. However, this fact is not entirely true anymore. In 2010, Liakh [64] et al. proved that Linux can operate without mixed code and data pages. Their work was accepted by the Linux community and published as a part of the official Linux kernel version 2.6.38.

We propose using this feature to extend HIMA to verify the integrity of the Linux kernel, as well as the user processes. To achieve this objective, HIMA should first extend its active monitoring to include any events that lead to loading or modifying executable pages inside the kernel. Afterward, it should store a complete list of all the executable modules inside the kernel to do attestation. For evaluation, we suggest testing this new HIMA extension by verifying its ability to detect an injected malicious kernel code (e.g., a rootkit).
Integrity Measurement of the Hypervisor: As discussed in Chapter 5, HyperSentry provides a totally isolated environment to measure the integrity of the highest privileged software. However, providing this environment requires a certain architecture that relies on both the SMM and the IPMI interface. While SMM is only available to x86 platforms, IPMI is popular in server platforms. Thus, both SMM and IPMI are widely present together in cloud computing infrastructures. However, they are not widely used together in other computing devices (e.g., desktop computers, tablets, and smart phones).

We propose extending HyperSentry to be a standard technique to measure the highest privileged software of a wide range of computing platforms. Such extension requires further research to prove the feasibility of implementing HyperSentry on various platforms. For example, to verify the integrity on desktop computers and laptops, HyperSentry can use Intel’s AMT [44] instead of IPMI. To run HyperSentry on smart phones, we have to investigate running it on ARM processors. In this case, HyperSentry would replace its SMM part with another part that uses ARM TrustZone [8].

Isolated Execution Environments: As discussed in Chapter 6, SICE’s multi-core mode directly enables two isolated execution environments to run concurrently. Using this unique capability, we propose to extend SICE to provide continuous monitoring of the highest privileged software (i.e., the hypervisor). Currently, existing mechanisms (including HyperSentry) can only be used for periodical integrity verification rather than continuous active monitoring. Intuitively, periodical integrity measurement cannot detect transient attacks.

We propose using the properties of SICE to allow an integrity monitor to concurrently run on a separate processor core, while allowing the monitoring target (e.g., the hypervisor) to use all other processor cores for its normal operations. SICE could provide a hardware-based isolation for the integrity monitor so that it cannot be manipulated by the running hypervisor.

However, SICE protection alone does not overcome the challenges that faces this proposal. Our proposed solution is required to provide techniques to overcome two additional technical challenges. The first is how to extract the right contextual information needed for the integrity monitor. For instance, each processor core has its own CR3 register, which presents the layout of the whole memory accessed by this core. The current multi-core processors architecture does not allow processor cores to access the registers of other cores. The second is how to provide an event driven mechanism to notify the monitor about changes that occur to the host memory and/or its execution environment. Such mechanism is necessary to protect against transient attacks. Further research is required to overcome both these challenges.
REFERENCES


[71] PaX. http://pax.grsecurity.net/.


