Portland State University
PDXScholar

Dissertations and Theses | Dissertations and Theses

7-12-2018

EPA-RIMM-V: Efficient Rootkit Detection for Virtualized Environments

Tejaswini Ajay Vibhute
Portland State University

Let us know how access to this document benefits you.

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

 Part of the Computer Sciences Commons

Recommended Citation

Vibhute, Tejaswini Ajay, "EPA-RIMM-V: Efficient Rootkit Detection for Virtualized Environments" (2018). *Dissertations and Theses.* Paper 4485.

10.15760/etd.6369

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. For more information, please contact pdxscholar@pdx.edu.

EPA-RIMM-V: Efficient Rootkit Detection for Virtualized Environments

by

Tejaswini Ajay Vibhute

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

Thesis Committee:
Karen L. Karavanic, Chair
Charles V. Wright
Kristin Tufte
Jeffrey Hollingsworth

Portland State University
2018

© 2018 Tejaswini Ajay Vibhute

Abstract

The use of virtualized environments continues to grow for efficient utilization of the available compute resources. Hypervisors virtualize the underlying hardware resources and allow multiple Operating Systems to run simultaneously on the same infrastructure. Since the hypervisor is installed at a higher privilege level than the Operating Systems in the software stack it is vulnerable to rootkits that can modify the environment to gain control, crash the system and even steal sensitive information. Thus, runtime integrity measurement of the hypervisor is essential. The currently proposed solutions achieve the goal by relying either partially or entirely on the features of the hypervisor itself, causing them to lack stealth and leaving themselves vulnerable to attack.

We have developed a performance sensitive methodology for identifying rootkits in hypervisors from System Management Mode (SMM) while using the features of SMI Transfer Monitor (STM). STM is a recent technology from Intel and it is a virtual machine manager at the firmware level. Our solution extends a research prototype called EPA-RIMM, developed by Delgado and Karavanic at Portland State University. Our solution extends the state of the art in that it stealthily performs measurements of hypervisor memory and critical data structures using firmware features, keeps performance perturbation to acceptable levels and leverages the security features provided by the STM. We describe our approach and include experimental results using a prototype we have developed for Xen hypervisor on Minnowboard Turbot, an open hardware platform.

This thesis is dedicated to my late grandparents, Basavraj Vibhute and Indumati Vibhute, who always encouraged me to fight the odds and pursue my dreams.

Acknowledgments

I am immensely indebted to my advisor Dr. Karen L. Karavanic for giving me the opportunity to work on this project. It was her motivation and guidance during the long research hours and writing phase that helped me achieve the goal.

I would like to extend my sincere gratitude towards my committee members Dr. Charles Wright, Dr. Kristin Tufte and Dr. Jeffrey Hollingsworth (UMD) for their time to read my thesis and provide valuable feedback.

I especially thank Brian Delgado for his foundational work on EPA-RIMM and sharing his expert knowledge on the subject and assisting me to resolve the technical problems during the thesis. I would also like to thank John Fastabend and Cody Shepherd for their insights in the project work.

I am grateful to my parents Ajay Vibhute and Shakuntala Vibhute for their continuous support and encouragement that helped me stay motivated. A big thank you to all my family members, siblings and friends for their patience and support.

Lastly, I would like to acknowledge the support of Computer Science Department staff members and CAT members for their timely assistance.

Table of Contents

List of Tables

List of Figures

Chapter 1 : Introduction

"With great power comes great responsibility."[1]

Virtualization efficiently utilizes ample hardware resources, keeps power consumption under check, reduces hardware deployment cost, and eases backups and recovery. For these reasons, the concept of virtualization has become popular. The computing world has seen a growing trend in the use of virtualized environments in infrastructure deployment, high performance computing environment, automation environment, and software testing [40], [41], [42], [43], [44]. Virtualization reduces infrastructure cost, eases resource management and allows efficient migration of resources and disaster recovery. Server farms, cloud infrastructure typically deploy virtualization software (hypervisor) for better utilization and management of resources. In the software stack, the hypervisor is generally installed at the highest privilege level, thus allowing each user to deploy their own operating system in isolation from each other. Each isolated operating system may run applications without creating any conflicts with other applications on separate operating system. The unique security features and advantages of hypervisors have increased their use in large-scale servers.

1.1 Motivation

Unfortunately, hypervisor being at the highest privilege level in the software stack causes it to become an interesting target for malware to launch an attack, gain control of the system, crash the system, and even steal sensitive information. There have been major security vulnerabilities reported repeatedly in hypervisors provided by different vendors [5], [9]. These risks have been studied by researchers to make hypervisors more secure

[1] Benjamin "Ben" Parker (Amazing Fantasy #15)

[33], [34]. As the code base of hypervisors increases because of new development, new vulnerabilities are being discovered and before they are resolved, a malicious application can take advantage of this and disrupt the system. More recent hypervisors also focus on embedded, automotive and native-cloud-computing use cases along with its traditional infrastructure virtualization application [35]. This new focus increases the scope of attack vectors. The CVE database (Common Vulnerability and Exposure database) keeps a track of the vulnerabilities found in different softwares and hardware components [2]. Some of the vulnerabilities listed in this database could be leveraged by a malicious code to launch Distributed Denial of Service (DDoS) attack, steal sensitive information by elevating privilege of the virtual machine user, or obtain control of the entire system. Such attacks may jeopardize the privacy of individuals and industries or even threaten national security. Hence, along with the operating system kernel, it is necessary to measure the integrity of the hypervisor as well. A study done by Thongthua, A. et al., [33] shows that two currently deployed hypervisors, ESXi and XenServer, have high vulnerability. Different hypervisors exhibit different vulnerabilities. As identified by Tavis Ormandy [5], instructions and emulated IO devices are the most complex pieces of code in the virtualization software and have the potential to include the most vulnerabilities. Attackers can leverage the incorrect implementation or incorrect handling of instructions to gain access to the system.

The Venom [9] vulnerability in QEMU's virtual floppy disk controller (FDC) is one such example of vulnerabilities that can occur in emulated IO devices. It was a major vulnerability discovered in 2015 that affected many hypervisors like Xen, KVM and

VirtualBox. QEMU is a hardware resource emulator used by most hypervisors. Venom is a type of VMEscape attack allowing the attacker to send a data packet from virtual machine to overflow FDC's data buffer and execute arbitrary code in the hypervisor context. Xen's XSA (Xen Security Advisories) [9] currently lists about 212 security vulnerabilities reported in the last five years that can lead to memory corruption, denial of service, privilege escalation and VMEscape. During the same period about 75000 lines of code have been added to Xen's code base [78]. So, there may still be undiscovered vulnerabilities in our hypervisors. Hence, we need a secure and effective mechanism to identify attacks against such undiscovered vulnerabilities, alert the administrator, and safeguard our servers, cloud infrastructure and embedded systems. Hence, it is necessary to measure the integrity of the hypervisor.

De Souza et al., [36] perform an extensive study on the current security trends for measuring hypervisor integrity and suggest the requirements that should be followed by an integrity measurement solution to be effective and performance efficient. Among the techniques implemented for integrity measurement, System Management Mode (SMM) based Runtime Integrity Measurement Monitors (RIMMs) have proven to be more difficult to break by an attacker since SMM is embedded in the firmware, at a higher privilege level than the hypervisor. The current integrity measurement techniques [11], [12], [13], [14] face problems of either poor performance problems or are ineffective against higher privilege layer based attacks.

1.2 Objective

The goal of this research is to develop a reasonably secure, performance aware solution to measure the integrity of the hypervisor from SMM.

1.3 Approach

We implement our approach as an extension to EPA-RIMM [21], a firmware based integrity measurement solution for kernels. It uses SMM to obtain the runtime context of the kernel for measuring its current state. EPA-RIMM takes into consideration the performance constraints that should be followed by an SMM module during its execution. It implements a unique scheduling technique during measurement that bounds the performance overhead. For these reasons, we choose EPA-RIMM as our base SMM based RIMM and build our solution upon it. We extend EPA-RIMM with STM for out-of-context integrity measurement of VT-x capable hypervisor and demonstrate its effectiveness in detecting rootkits in such hypervisors. This work also demonstrates the performance efficiency of the method. We accomplish integrity checking without assistance from any extra hardware or even from the hypervisor. Finally, we present a comparative performance analysis of EPA-RIMM in STM vs. non-STM environment.

Since this is a firmware-based approach, in our research we need to modify the system firmware, deploy the modified version and test it. Firmware being the most privileged and sensitive software it is locked by computer vendors in commercial products. Hence, it is not possible to use the commercially available products to implement and demonstrate our solution. Instead, we use open source hardware development boards, Minnowboard Turbot equipped with 64-bit Intel Dual Core Atom processor [16]. The

firmware of these boards is unlocked and can be changed to have custom-built firmware. The Minnowboards and systems deployed in server farms both have similar x86 architecture, with the difference that the servers have higher compute capability and have more cores than Minnowboards. This makes the server systems faster and more efficient. Hence, Minnowboards are used in this work as proof-of-concept representation of servers. With this approach, we achieve an effective, small Trusted Computing Base (TCB), fast, persistent, extensible SMM based hypervisor integrity solution.

An SMM-based hypervisor integrity measurement solution poses a challenge: understanding the hypervisor context for hypervisors that utilize hardware-based virtualization. Such hypervisors show a peculiar behavior when the processor switches context from non-SMM mode to SMM-mode. This behavior creates a semantic gap problem for the SMM-based measurement unit in which it cannot determine if the interrupted state was from the hypervisor or the hypervisor's virtualized guest. We propose an effective and performance aware firmware solution that solves this problem. We do this using Intel's SMM virtualization interface called SMI Transfer Monitor (STM) [15]. STM uses hardware-based virtualization technique to virtualize SMM itself. STM assists in gaining control over uncertain processor context behavior of hypervisors with VT-x hardware support; additionally, it creates a layer of trust over SMM.

1.4 Thesis Statement

A combination of SMM and STM can be used for performance efficient, reasonably secure, configurable integrity measurement of hypervisors.

1.5 Contributions

1. **Developed a novel technique to solve the SMM-hypervisor semantic gap problem:**

   In a virtualized environment running VT-x (hardware virtualization) supported guests, upon System Management Interrupt (SMI) when all the processors enter SMM, there is no guaranteed knowledge of processor's context before the interrupt. A processor may be operating either in hypervisor context or in a virtual machine's context. This uncertainty makes it problematic to obtain hypervisor information in SMM with confidence. In this research, we implemented a novel technique that uses STM, a firmware based feature, to obtain out-of-context hypervisor status from SMM and eliminates the uncertain processor context problem. This technique is the basis for performing stealthy out-of-context hypervisor integrity checking. It eliminates the need of using special hardware components and avoids injecting instructions in the normal SMI exit code flow as implemented in HyperSentry [11]. Other SMM applications may also use this technique to retrieve hypervisor context. The usage of this module itself has a very low performance overhead. We plan to open source this module and integrate it with the current STM code base.

2. **Implemented STM enabling in the hypervisor and developed a set of fine-grained permissions that should be set for hypervisor resources:**

   In order to enable STM, both BIOS and the hypervisor should execute an STM setup handshake during the system initialization phase. The BIOS implementation of STM setup is part of the UEFI code base. Similarly, the hypervisor should

implement its part of the handshake during its initialization phase, acknowledge enabling of STM, and specify any resource access permissions that should be obeyed by the SMI handlers. We implement the missing hypervisor handshake mechanism in Xen's kernel. We also developed a set of permissions that should be set over hypervisor resources to avoid illegal access by SMI handlers. System Management Mode being at the highest privileged level, it can access all the resources and memory of the hypervisor. On a system with these permissions not enabled, an SMM based rootkit may leverage the ability of SMM to tamper the software stack, steal sensitive information or even crash the entire system. We plan to release a patch of this work to the Xen Project team.

3. **Implemented a performance collector interface between STM and the host software and conducted a full investigation of STM performance:**

   To get the STM performance data in the host software we designed and implemented an API interface between the hypervisor and STM. Using the data obtained from this interface, we present a detailed performance cost analysis of STM. To the best of our knowledge, this is the first STM performance study available for an STM configured system with cooperation between hypervisor and BIOS.

4. **Developed a prototype of EPA-RIMM-V:**

   To validate our architecture, we develop and deploy the prototype on Minnowboards. We successfully perform integrity checking of memory, registers, and model specific registers for the Xen hypervisor. We evaluated the effectiveness of our model by simulating two rootkits that are representative of

the family of hypervisor rootkits, and detected them in subsequent measurement invocations. We plan to release the prototype as open source research infrastructure.

Chapter 2 : Background

This work demonstrates a technique for integrity measurement of hypervisors from SMM using STM. We achieve the goal by combining concepts from three main areas: host software virtualization; security; and hardware execution environments. In this chapter, we summarize these three areas.

2.1 Virtualization

CPU virtualization allows to us run multiple applications or processes simultaneously on the same system. This enables efficient resource utilization. Virtualization can be applied to hardware as well, thus allowing multiple operating systems to share hardware resources. In this section we look at the concepts associated with hardware-based virtualization and the Xen open source hypervisor.

2.1.1 Hypervisor

The virtualization layer called the hypervisor creates virtual execution environments called Virtual Machines (VMs), allowing multiple Operating Systems to run on the same platform. The hypervisor abstracts the hardware resources in order to give each VM a consistent view of the hardware as a virtual resource. Each operating system runs on the allocated virtual resources in isolation from other operating systems. Each VM may be used to host different applications like a web server, mail server, database etc. All such applications that were earlier hosted on different physical systems can now be hosted on the same physical system but in different virtual environments.

Two types of hypervisors are available in the market: Bare-Metal (Type-1) and Hosted (Type-2) hypervisors. Hypervisors are sometimes interchangeably also called Virtual

Machine Monitors (VMMs). *Bare-Metal* Hypervisors are directly installed on top of the firmware. Recent Bare-Metal hypervisors are supported by the platform's virtualization instructions. On Intel platforms these are called VMX instructions, while on AMD architecture they are called SVM instructions [4], [79]. A *hosted* hypervisor is installed as an application in the operating system. There are two main implementation designs for virtualization: Hardware Virtualization and Paravirtualization. In *Hardware Virtualization*, the guest virtual machine, called hardware virtualized machine (HVM), is unaware of the existence of the underlying hypervisor software layer and relies on hardware virtualization technology to execute privileged instructions. While in *Paravirtualization*, the guest virtual machine is aware of the existence of the hypervisor. A Paravirtualized guest's kernel is modified so that it can execute in cooperation with the hypervisor [1]. Since the Bare-Metal hypervisor is the primary software installed directly above firmware, its execution environment is termed VMX-root mode on Intel Architecture platforms.

In this work we look at security challenges and solutions for Bare-Metal hypervisors and refer to them as hypervisor in the entire discussion.

2.1.2 Guest System (Virtual Machine)

Each Virtual Machine hosted on the hypervisor runs independently of other virtual machines in its own virtual environment, using the VMM interface for required resources. Each VM has its own complete software stack with software applications. The Virtual Machine's execution environment is referred to as the VMX non-root mode. As mentioned in the previous section, guest systems can be of two types either

Paravirtualized or Hardware-Virtualized (HVM) depending on whether the guest Operating System is modified to run in a virtual environment or it is unmodified and takes advantage of the underlying hardware virtualization features to maintain its state.

2.1.3 Virtual Machine Instructions

Since in this work we are working on Intel architecture we explain here some of the VMX instructions. We also explain the life-cycle of a guest machine in terms of VMX instructions. There are two types of operations supported by Intel processors: VMX root and VMX non-root. During system initialization, the software can enter VMX operation by setting CR4.VMXE and then executing instruction VMXON from root privilege. To leave VMX mode, software should execute instruction VMXOFF. The host software that executes VMXON operates in VMX root mode after entering VMX. The guest VMs created by this software will execute in VMX non-root mode. Transitions between VMX root and VMX non-root mode are called VMX transitions. Processor transitions from the hypervisor into the guest are called VM Entry, and transitions from guest to hypervisor are called VM Exit. When the hypervisor launches the guest VMs for the first time it does so by executing instruction VMLAUNCH. All subsequent entries into guest are executed by instruction VMRESUME. The VM guest may request services from the hypervisor by executing instruction VMCALL. This instruction causes a VM Exit into the hypervisor. The hypervisor uses a data structure called the Virtual Machine Control Structure (VMCS), to save the processor state between VMX transitions. The lifecycle of VM guests and the hypervisor with VMX instructions is shown in **Figure 2.1** [4].

2.1.4 VMCS

As shown in **Figure 2.2**, the Virtual Machine Control Structure (VMCS) is a data structure maintained by the hypervisor for each of its hardware-virtualized (HVM) VMs. The hypervisor maintains a VMCS per virtual CPU of each guest. The processing state of the guest is saved in its respective VMCS during a context switch from the guest into the hypervisor. While re-entering the guest from the hypervisor the processing state of the guest is restored by reading its previously saved state from the VMCS. The hypervisor uses instruction VMPTRLD to load the appropriate VMCS in memory and switch control. The hypervisor may read data from the loaded VMCS using instruction VMREAD and write to it using instruction VMWRITE. To save the VMCS back to memory, the hypervisor uses instruction VMPTRST [4].

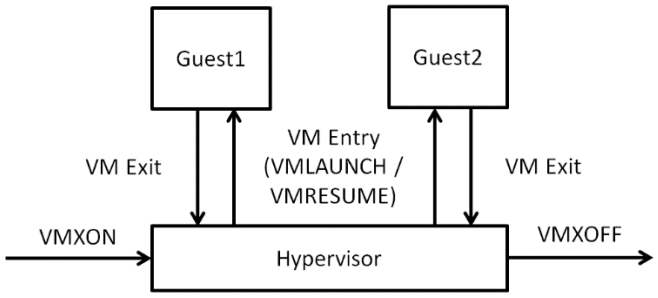

Figure 2.1: Lifecycle of hypervisor and guest machines using Intel VMX instructions.

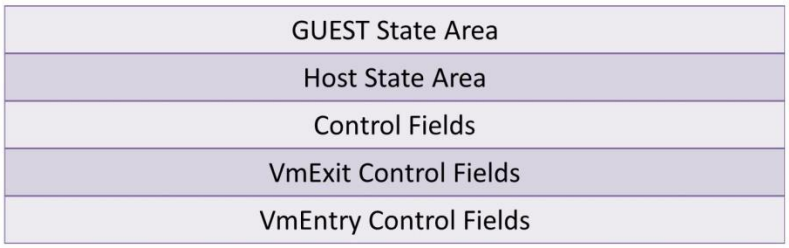

Figure 2.2: VMCS Layout – A data structure per virtual CPU for each Virtual Machine of the Hypervisor.

2.1.5 Xen

Xen is an open source bare-metal hypervisor that supports both Paravirtualized (PV) guests and Hardware Virtualized guests (HVM). Xen execute its guests in environments called as *Domains*. The architecture is lightweight in which the hypervisor itself performs basic control operations, while management software performs the admission control, applies policy decisions and manages CPU sharing among guests. This management software is implemented in a privileged guest called *Domain0* (*Dom0*) and the other unprivileged guests are known as *DomainU* (*DomU*). Xen handles resource access and modification differently for PV and HVM guests. For successful functioning, PV guests require a modified OS kernel and need additional driver support, while HVM guests do not require any OS kernel modifications.

Domain0: This is the first guest that Xen boots up after it is initialized. Dom0 is a PV guest that has elevated privileges and provides the management interface for all the guests. Hardware emulation for different resources is implemented in this domain. Dom0 is responsible for managing how the resources are shared amongst the guests. It executes a hypercall to request Xen to perform any privileged instructions. Hypercalls are like Linux system calls and work between PV guests and the hypervisor. When Xen launches Domain0, it allocates the needed physical memory and manages its memory updates.

HVM Guests: HVM guests are fully-virtualized guests with no modified kernel. When HVM guests are created, Xen assigns the guest a virtual memory region that the guest considers as real physical memory. All the updates to the guest's page table are trapped and handled by the hypervisor. The processor state of the guest is maintained by the

hypervisor in the respective VMCS structure allocated for the guest. This VMCS structure holds the address of the page table the guest treats as the actual machine page table in the CR3 register field. When the processor executing in the HVM guest transitions into SMM on System Management Interrupt the SMM views the HVM guest's processor state instead of the hypervisor's processor state [29], [70].

2.2 Security

Computer Security is a method of protecting the computer systems and the data that they store from unwelcome accesses or modifications. It also focuses on building systems that are dependable in the face of attack due to malicious software. Different types of malicious code or malware can cause different type of attacks like Denial of service, spoofing, tampering, privilege escalation, phishing or social engineering [54], [69]. In our work, we concentrate on rootkits, malicious software that targets the ring0 software.

2.2.1 Malware

Malware is a piece of software that has the ability to change the normal behavior of, gain unauthorized access to, or steal information from a computer or any software running on it. Malware is an umbrella term for spyware, keyloggers, rootkits, ransomware, Trojan horses, botnets, worms and viruses. Each type of malware is distinguished by the intent of the creator associated with it. Spyware steal sensitive information, keyloggers log the keyboard behavior specifically to gather passwords, rootkits attack the operating system or hypervisor and hide their traces, ransomware is deployed to extort money. Different malware can target different systems. The Mirari botnet was an IoT based malware responsible for launching a DDoS attack against the DYN DNS server [17]. WannaCry

ransomware was spread via a computer worm making it one of the biggest cyber-attacks [18]. Judy is an Android phone targeted adware deployed to make money for the attackers by generating ad revenue [19].

2.2.2 Rootkit

Rootkit is a software that gains access to the ring0 software and hides itself under authentic applications and tools and may perform malicious activities. Rootkits have the ability to hijack control flow by modifying return addresses, function pointers, kernel data structures or non-control data. In the virtualization environments, rootkits can hijack the interface between virtual machine and the hypervisor and in certain cases even the hypervisor itself. Once the hypervisor is compromised the entire environment including all the virtual machines running on the hypervisor can be affected. This is possible by exploiting vulnerabilities present at different layers.

2.2.3 RIMM

Runtime Integrity Measurement Mechanism is a method to measure the correctness of the system at any point in time while the system is running. This mechanism may include detecting resident malicious software, or corruption of privileged code and data structures.

2.3 Hardware Execution Environments

Hardware Isolation environments are processor features that provide an isolated environment in the application address space, firmware or as a new chip on the system itself. The primary purpose of these environments is to allow application developers,

system administrators or OEMs to execute critical code or store sensitive information such as cryptographic keys. Different processors provide different types of hardware execution environments. To run applications in an isolated environment, Intel x86 processors provide SMM and SMM Transfer Monitor (STM) as firmware isolated environments, the Management Engine (Intel ME) as a separate on-board processor specially designed for security applications and Software Guard Extensions (Intel SGX). SGX consists of a set of special hardware instructions that application developers can use to create isolated environments in the ring3 application layer. AMD processors have Platform Security Processor (PSP), a separate processor from the main CPU designed for security applications similar to Intel ME. Along similar lines, ARM has TrustZone with SMM like functionality. Trusted Computing Group (TCG) provides another feature called Dynamic Root of Trust for Management (D-RTM) that they developed independently. D-RTM allows establishing trust in the ring0 kernel at any time instead of only at the time of system initialization. Intel implemented its version of Root of Trust for Management (RTM) as Intel Trusted Execution Technology (TXT) and AMD implemented it as Secure Virtual Machine (SVM) [53]. Intel's TXT provides S-RTM (Static Root of Trust for Management), by measuring the components from the platform reset stage until the host software components against keys saved in Platform Configurable Registers (PCR) [39], [71].

In our work we focus on Intel processors, we utilize SMM and STM as hardware isolated environments to execute the integrity measurement unit.

2.3.1 System Management Mode (SMM)

SMM is a privileged operating mode present on Intel processors since the 386SL processor as well as on AMD x86 processors [4], [79]. This mode is used to perform system-wide functions related to power management and system hardware control [4]. A system can enter SMM by asserting an System Management Interrupt (SMI) on the SMI# pin on the processor or via an SMI message received on the APIC bus. SMI is the highest privileged instruction and is a nonmaskable external interrupt. It has higher priority than any other interrupt even Non Maskable Interrupt (NMI). On an SMI the processor saves its current context and switches to a separate address space. The SMI handler then starts executing in this address space. SMM can be exited only by instruction RSM (resume) on Intel processors and with IRET instruction on AMD processors. On RSM or IRET, the saved processor state is restored and the processor starts executing from where it was interrupted. When the system enters SMM, the hostside execution is suspended. The SMI handler code and the processor save state are located in a dedicated memory location called SMRAM [3], [4]. On a multi-processor system, only one CPU executes the SMI handler while other CPUs spin wait until the SMI handler execution is completed.

2.3.2 SMI Transfer Monitor (STM)

The SMI Handler has the ability to access memory region beyond SMRAM. Thus, if an SMI handler is compromised, it can read or write any memory region [15]. One way to solve this problem is by de-privileging SMI handlers. The SMI handler can be de-privileged only if there exists a higher privileged software than SMI handler such as the

STM. The SMI handler should be de-privileged in such a fashion that it should be able to access the required appropriate hardware resources for its normal operations [15].

Since both SMI handler and host software are distrustful of each other, a higher privileged trusted software is required to establish trusted communication between these two. The STM is a hypervisor at the SMM level, which acts as a trusted channel between the SMI handler and the host software. STM thus becomes the highest privilege software, with the SMI handler and the host software as its guests. It has access to the entire system and can restrict an SMI handler from accessing resources outside of its allocated address space. Since STM is at a higher privilege level than the SMI handler and the host software, both of these entities must agree on opting in to enable the STM. This opt-in should occur before host software starts running. During boot-time, STM is loaded in a special region of SMRAM called the Monitor Segment (MSEG). The STM memory layout in SMRAM is shown in **Figure 2.4**. On a TXT capable system, during system initialization state TXT verifies STM and then loads it into MSEG region, thus establishing S-RTM.

The STM follows the SMI property that on an SMI all the CPUs must enter STM. Once all the processors enter the STM, one processor is elected to execute the SMI handler while other processors spin in a wait loop. To identify its guests and to store the processor state between context switches, STM keeps a VMCS (Virtual Machine Control Structure) for both. An exit from host software either from hypervisor or from the hypervisor's virtual machine into STM due to SMI is called SMM VM exit since it is an exit from the host software. On receiving SMM VM exit, STM saves the current VMCS

pointer for the host software in its associated VMCS. It also stores the processor context in a special region called the save state area. The return from STM into host software is called an SMM VM entry since it is an entry into the host software. While on the return path, STM refers to the host software's VMCS and the save state region to restore the processor context. An exit from STM into the SMM guest for SMI handler execution is accomplished with a call to VmResume. Once the SMI handler finishes its execution, it executes the RSM instruction that causes an exit into STM. The STM saves the essential processor state information for SMM guest in its VMCS. The STM also maintains a database of VMCS pointers for host software hypervisor's (i.e. Xen) virtual machines. At the time of virtual machine creation, the host software hypervisor may execute a ManageVmcsDatabase VMCALL to request STM to store the newly created Virtual Machine's VMCS pointer along with the access policy for this VM. The access policy enforces a restriction on the VM register states visible to the SMI handlers.

In virtualized environments, the processor may be executing either in VMX root mode or in VMX non-root mode at any instant. If an SMI is triggered when the processor is operating in VMX root mode, STM saves the VMXON pointer in the host software's VMCS region. **Figure 2.3 (a)** shows the flow when SMI occurs in VMX root environment. If an SMI is triggered when the processor is operating in VMX non-root mode, i.e. in the host software's virtual machine context, STM saves the VMCS pointer for the virtual machine in the host software's VMCS region. **Figure 2.3 (b)** shows the flow when SMI occurs in VMX non-root environment.

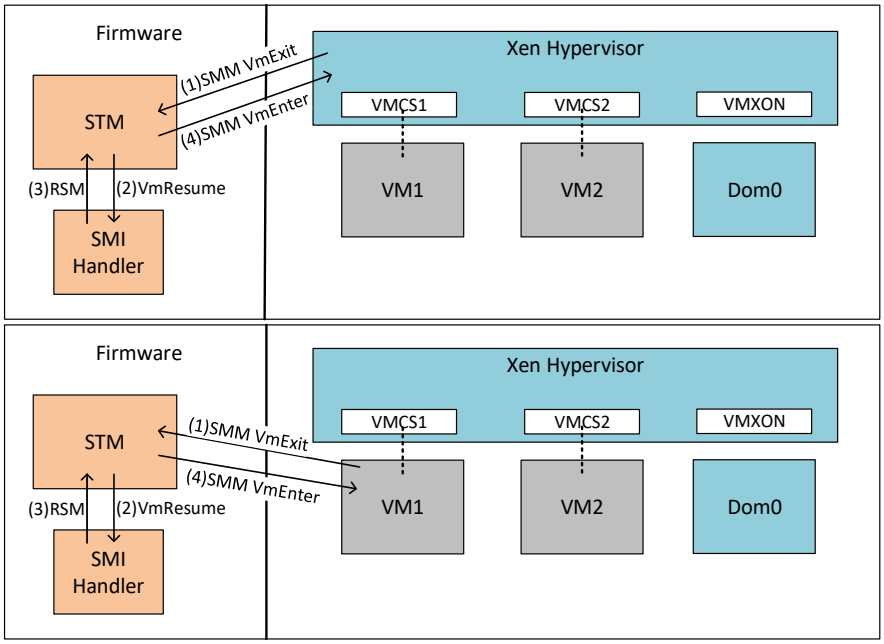

Figure 2.3: SMI behavior for Virtualized Environment - (Top to Bottom) (a) SMI Triggered when CPU is running in VMX root mode - It causes an SMM VM Exit into STM, it saves the VMXON pointer for Xen and transfers control to SMI Handler via VmResume. SMI Handler finishes its execution and returns via RSM. STM restores the state of the processor it previously stored and returns control to Xen. **(b) SMI Triggered when CPU is running in VMX non-root mode -** It causes an SMM VM Exit into STM, it saves the VMCS pointer for VM and transfers control to SMI Handler via VmResume. SMI Handler finishes its execution and returns via RSM. STM restores the state of the processor it previously stored and returns control to VM.

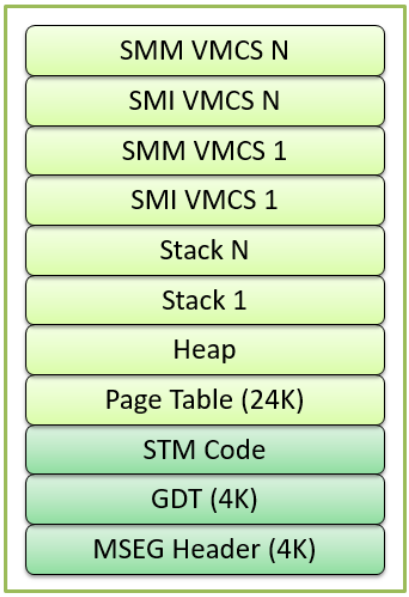

Figure 2.4: STM Memory Layout – The Dark green part is the static STM image loaded in MSEG and the Light Green portion shows the region of MSEG used by STM image during its execution [15].

Chapter 3 : Related Work

In the past researchers have made attempts and proposed solutions to verify and protect different layers of the software stack. As the software evolved, along with it evolved rootkits trying to penetrate each layer. Hence, it is necessary to protect all of the layers of software running on the system. The approaches taken by researchers can be classified depending on the techniques implemented and software layer that it protects. This chapter takes a look into different rootkit detection techniques.

3.1 Kernel Introspection

Some approaches like rkscan [58], [59], St. Michael [60], LIDS [61] implement rootkit detection methods from within the kernel. This method gives them high privilege to inspect the ring3 layer (application layer) as well as ring0 layer (kernel layer). However, these detectors may be susceptible to attacks from the privileged layer-based rootkits before these rootkits are even detected. Hence, this technique suffers from self-protection and may be vulnerable to attacks.

3.2 Kernel verification via Virtual Machine Introspection

One way to secure the detector from a kernel level rootkit and overcome the problem of self-protection is to run the detector in a higher privilege level than the rootkit. A number of solutions have been developed to measure the kernel code from the hypervisor. Livewire [54], VMwatcher [56], VirtAv [57] are some of the solutions that implemented this concept to extract kernel knowledge and perform inspection over it to detect anomalous behavior. Hypervisor being at the highest privileged layer in the software stack is a very attractive layer for attackers to place their rootkits and take control over

the entire system on which it has been deployed. Again, these solutions also suffer from self-protection. This makes it essential to verify even the hypervisor.

3.3 Hypervisor Verification

Multiple researchers have tried to find solutions to measure the integrity of the hypervisor efficiently but unfortunately, they do not address all the concerns of hypervisor integrity checking and the performance constraints that should be followed by the measurement unit itself. In the following section, we discuss these research methods and see how our approach of using SMM and STM for hypervisors addresses these concerns.

3.3.1 In-context hypervisor verification

One of the early works in this domain was HyperSafe [22], which illustrated an approach of securing the hypervisor by locking down the hypervisor code and static data pages and introducing pointer indexing for control flows. Even though their technique seems promising, non-control flow data like IDTR, MSRs also need verification and protection. In case a rootkit is able to change IDTR to point to a malicious IDT page then HyperSafe will be attesting the malicious IDT page instead of valid IDT page. The Event driven VMM monitoring [23] approach makes use of Instrumentation-based Privilege Restriction (IPR) and Address Space Randomization (ASR) techniques to securely perform in-context integrity measurement. This approach implements IPR with Shadow Page tables, accessible by a monitoring unit, and use ASR to randomly place the monitoring unit in memory during initialization phase.

3.3.2 Nested Virtualization

CloudVisor [24] uses nested virtualization, running a VMM in a virtualized environment, to de-privilege VMM and secure the control transition channel between the hypervisor and the virtual machine. Using similar nested virtualization concept, Nezhad, A.S. et.al. [26], propose an architecture that uses many layered hypervisors each with varying and adjustable functionalities to secure the virtual machines. Nested hypervisors are vulnerable since they face similar threats as traditional hypervisors.

3.3.3 Microhypervisors

Some researchers have studied extending the secure microkernel approach developed for operating systems to virtualization as well. One such study is NOVA [47] that implements decomposed virtualization while supporting full virtualization. They decompose virtualization into fine-grained functional components as root-partition manager, multiple virtual machine monitors, device drivers and other system services and implement the principle of least-privilege among all of these components.

3.3.4 Trusted Execution Environment based integrity monitoring

In a study by Zhang, F. et al. [53] the authors present a detailed report on benefits and vulnerabilities in currently available hardware assisted Trusted Execution Environments (TEEs). Since these TEEs are supported by the hardware vendors themselves they can be used for trustworthy computing with no additional modifications to the system. SMM and STM are one of the many TEE available on Intel platforms. Although SMM was not primarily designed for security purposes, research using this mode for system health checking have shown that SMM can be an effective environment for an integrity monitor.

For example, HyperSentry [11], HyperCheck [12], HyperGuard [13] and SPECTRE [14]. Unfortunately, they face some serious shortcomings: for example HyperGuard and HyperCheck do not invoke the SMI handler stealthily. HyperGuard makes use of a system timer to trigger each SMI while HyperCheck uses Network Interface Card (NIC). A rootkit may poll for SMIs or specific inputs on these resources and scrub its attack before the integrity checker can measure the hypervisor. As studied by Delgado et al. [8] and recommended by Intel's BITS tool [80] exceeding 1.5ms threshold for execution time in SMM may affect the execution of applications adversely. All of the above methods exceed this time constraint by orders of magnitudes: HyperSentry takes 35ms for measuring the integrity of hypervisor kernel. SPECTRE measures the integrity of Linux kernel, applications and heap and is able to detect heap overflow attacks and heap spray attacks. However, even this mechanism takes an execution time of at least 5ms.

SMM faces semantic gap challenge for hypervisors executing on VMX enabled systems. HyperSentry addresses this problem while triggering SMI via IPMI BMC out-of-band communication channel to achieve stealthy SMI invocation. It uses performance counters to obtain the hypervisor context in case the SMI is triggered when the processor was executing in a VMX guest. To accomplish hypervisor integrity checking, the approach places the measurement agent inside the hypervisor context. However, locating a measurement agent inside the same context as compromised software's context may lead to manipulation of results by an advanced rootkit.

An STM based approach that is currently being researched at Trust Mechanisms, Information Assurance Research is STM/PE with XHIM [50]. The XHIM (Xen

Hypervisor Integrity Measurer) which is based on LKIM (Linux Kernel Integrity Measurer) [51] collects hypervisor state at a given moment and sends it to the DM (decision maker) for analysis. DM may be placed either on the same system as the target machine or on a different machine. The measurement agent exports hypervisor context information from the target machine, which may be vulnerable to network attacks. This approach relies on an SMI timer to invoke measurement agent. A rootkit in the kernel may disable the SMI timer, thus not allowing the measurement agent to run.

3.3.5 Additional hardware for integrity monitoring

HyperWall [25] presents a different approach towards securing the VM from a malicious hypervisor. In their design, they propose to modify the microprocessor and the memory management unit and introduce new confidentiality and integrity protection (CIP) tables in memory to protect the memory of the guest virtual machines. Copilot [38] uses co-processor based technique to introspect the system. In this technique copilot monitor is periodically triggered and it reads kernel data over a PCI device. A smart rootkit might scrub its attack base before the monitor is triggered and restore itself after the measurement.

3.4 Hardware assisted isolated execution environment

Other researchers have proposed creation of isolated environments to run critical applications. One such work is SICE [52] where the authors create an isolated execution environment using SMM for the critical application to run. This technique saves the critical context of the application before switching to a different application in the SMRAM.

3.5 Other SMM and STM based applications

SMM and STM-based related research have looked into SMM-based malicious environment debugging, for example, MALT [63]. This capability could be useful while doing forensic analysis on the traces of the malware and the affected environment. SMM has also been used to protect login credentials with TrustLogin [64]. Dell holds a patent on using STM and SMM to sandbox SMI handlers and different SMM Drivers from each other [65]. This work aims at constraining resource accesses requested by the SMI handler to their own virtual environment.

3.6 Security challenges of System Management Mode

Some prior studies have shown that there is a possibility of system threat due to vulnerabilities in hardware-based TEEs including SMM [53], [66]. These vulnerabilities arise due to incorrect implementation of features in the environment. Research has shown some SMM-based rootkits [74], [75], [76], [77] and exploiting the SMM itself by cache poisoning [72] or by exploiting vulnerability in Intel's TXT [73]. In the recent versions, these SMM vulnerabilities have been patched while adding some added security features to UEFI-SMM implementation such as restricted memory access to system memory [28]. Moreover, STM provides additional security by virtualizing SMM. STM may itself be validated via Static Root of Trust mechanism (SRTM) during system boot phase.

Chapter 4 : Prototype Design and Implementation

In this chapter, we describe a novel method for detecting persistent rootkits in hypervisors from SMM. Our starting point for this work was an existing measurement tool under development at Portland State called EPA-RIMM. We describe EPA-RIMM in Section 4.1. To extend EPA-RIMM for use with hypervisors we developed an STM-based method to bridge the semantic gap between SMM and the Xen hypervisor, implementing the STM opt-in in Xen and developing an STM performance collector interface between Xen and STM. We call this modified version EPA-RIMM-V (EPA-RIMM for Virtualized platforms). We describe the design and implementation of EPA-RIMM-V in Section 4.2.

4.1 EPA-RIMM

EPA-RIMM is an SMM-RIMM framework developed by Delgado et al. [21] with the goal to provide quick detection of kernel or hypervisor rootkits by identifying unexpected changes in system state snapshots. It accomplishes this by periodically interrupting the running system to inspect sets of presumed static resources to identify changes, any of which would be a strong indicator of compromise. EPA-RIMM decomposes large integrity measurements to remain consistent with expectations regarding SMI latency. It implements a scheduler that facilitates a varying time budget for measurements allowing the performance-security tradeoff to be adjusted during runtime based on the threat landscape. EPA-RIMM architecture as shown in **Figure 4.1** has the components Diagnosis Manager, Backend Manager, Host Communications Manager and Inspector. There are three key abstractions in the architecture: *Checks*, *Tasks*, and *Bins*. A *Check* is a

description of a system resource measurement, including a command and its arguments, a priority, and a decomposition target. *Checks* allow the administrator to specify measurements over sets of memory regions, Control Registers, and Model-Specific Registers (MSRs). Sample *Checks* include: “Static Kernel Code Sections” that measures the kernel code sections to identify code injections, the “IDTR” *Check* that verifies that the IDTR register does not point to a different Interrupt Descriptor Table structure, the “GDT” *Check* that measures the Global Descriptor Table (GDT) to determine if it has changed. Other *Checks* measure specific MSRs or CPU control registers (e.g. CR0, CR4), for example, to determine if the Supervisor Mode Execution Protection (SMEP) were disabled by malware. At runtime, *Checks* are decomposed into some number of *Tasks*, or partial resource measurements. *Tasks* are scheduled by filling *Bins*, where each *Bin*’s size is defined as the sum of the execution times of the *Tasks* it contains.

The Backend Manager (BEM) receives *Checks* from the Diagnosis Manager and decomposes them into smaller *Tasks* to avoid prolonged SMM session times. The Backend Manager schedules *Tasks* by filling *Bins* based on a target *Bin* size. It signs and encrypts each *Bin* then provides it to the Host Communications Manager. The BEM receives back the Inspector’s signed and encrypted results. It decrypts the results and checks the signature to ensure that they came from the proper Inspector. The BEM merges individual *Task* results into a single *Check* result of true or false and sends the results to the Diagnosis Manager. The Host Communications Manager communicates with the Inspector either via an out-of-band mechanism or by an in-band mechanism. The Inspector is an SMI handler compiled into the BIOS and is initiated by an SMI.

The Inspector module of this RIMM computes the hash values of the specified measurements during the provisioning phase and writes the hash values back in the result descriptor. During the subsequent measurements, it utilizes these hash values previously computed to verify the current state of the system [21].

This model of EPA-RIMM addresses the SMI handler performance problem - Inspector can be executed close to 150µs time constraint. It can even successfully detect rootkits in hypervisor running a para-virtualized guest. However, it does not solve the problem of processor state uncertainty for VMX-supported hypervisors running HVM guests.

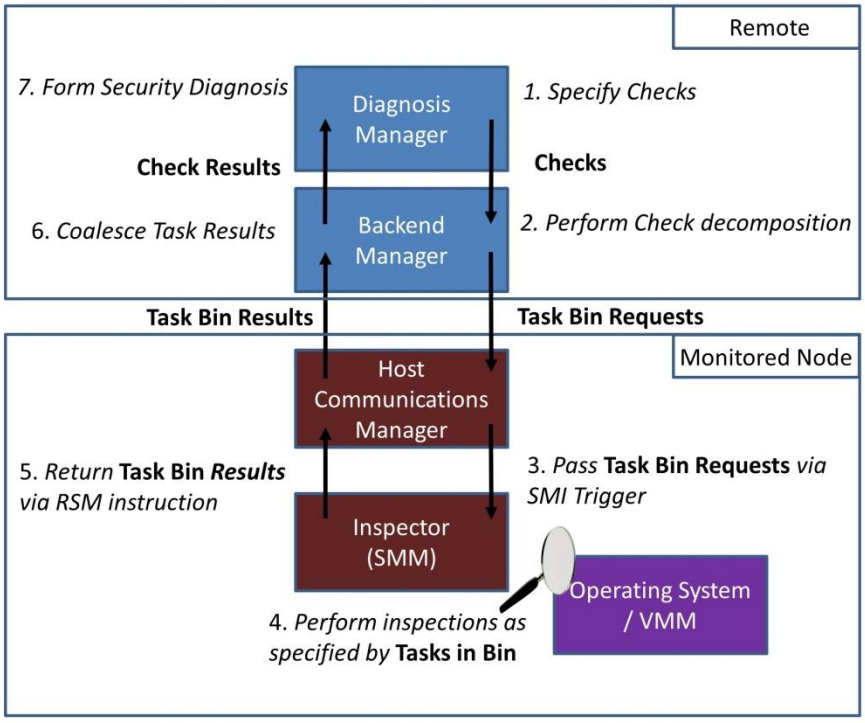

Figure 4.1: The EPA-RIMM Architecture - The Diagnosis Manager is an application-level component that specifies the *Checks* to be performed. The Backend Manager decomposes the *Checks* into work units called *Tasks* and schedules these into *Bins*. The Host Communications Manager is a component of the Monitored node that forwards the *Task-Bin* to the Inspector and after inspection collects the results and returns the *Task-Bin* updated with results to the Backend Manager. The system analyzer component (Inspector) measures the ring0 software for anomalies and sends results back to Diagnosis Manager [21].

4.2 Design of EPA-RIMM-V

EPA-RIMM-V is divided into four components: hardware, firmware and the host software. Measuring virtualized environments required changes to the Inspector, the Host Communications Manager and the Backend Manager. In this section we detail these changes.

4.2.1 Research Testbed

For implementing our prototype, we need a platform that allows us to modify firmware, and a processor that supports STM. On the commercially available systems, the firmware is locked down for security reasons, thus restricting the user to change the firmware manually. Hence, we use an open source development board Minnowboard Turbot as our prototype system. This is an open source hardware platform with Intel Atom E3826 dual-core processor, 2GB RAM, 1.46 GHz frequency [16]. This processor series supports Intel's STM feature. We use UEFI as our firmware, since Minnowboard is UEFI firmware compatible, and open source EDK-II firmware development kit to develop and build UEFI, SMM and STM [27]. We use Xen 4.9 as our hypervisor and Ubuntu 14.04 with Linux kernel 4.11 as Domain0. Xen and Ubuntu make up our host software.

UEFI BIOS and binary object modules for Minnowboard are available as open source resources to build custom firmware images [28]. STM is also an open source project with its source code available on GitHub [30]. We use the most recent UEFI Minnowboard firmware version 0.97 for platform specific binaries. The EDK-II implementation of UEFI for this Minnowboard firmware version does not have complete support for STM. We updated the UEFI EDK-II build specification to support STM. STM is built in

isolation first, and then the binary generated during the build is integrated with EDK-II. The entire package is then compiled to get one single binary. During system boot up, in the platform initialization phase UEFI loads STM image in the MSEG region and opts-in to STM.

4.2.2 Host Software

We use an open source bare-metal hypervisor, Xen 4.9, as our host software. Xen runs Ubuntu 14.04 with Linux kernel 4.11 as Domain0. We use Ubuntu 16.04 as our HVM guest (DomainU).

For every HVM guest Xen maintains a VMCS per virtual CPU to save the processor context between context switches. It maintains a unique pointer called VMXON for itself, which is used while executing VMXON instruction. To enable STM and use it, the host software should also do an opt-in. This opt-in is like a handshake between firmware and host software to setup STM. Host software should perform this step during its initialization phase before it starts running. We implement this opt-in mechanism in Xen. We also define and implement a set of fine-grained access policies over Xen's resources, such that SMI handler will have restricted access to Xen's resources.

4.2.3 Implementation

To completely implement and test our approach we made changes to Xen launch code for opt-in to STM, added a new module to STM to interpret hypervisor context from available VMCS and modified EPA-RIMM's Inspector module to obtain hypervisor specific information from STM for inspection. We also modified EPA-RIMM's Host Communications Manager module for special handling of hypervisors. Furthermore, to

analyze and collect STM performance data we implement a VMCALL interface between STM and host software. If performance collection is enabled as part of firmware compilation, this interface invokes a VMCALL to gather performance data from STM and report it to Xen. We also define a set of fine-grained resource access policies over Xen resources that STM enforce on the SMI handlers.

4.2.4 STM Support in Xen

Xen is the example hypervisor we have considered in our prototype to test our modified RIMM for detecting rootkits in the hypervisor. To enable STM, the ring0 software has to complete a handshake procedure with the firmware. This handshake is called the STM opt-in. This opt-in is performed during system initialization. Xen should perform this operation from VMX root mode. The current Xen kernel does not implement this opt-in. We introduce a new function, launch_stm(), which is executed at the end of Xen setup stage after hardware virtualization is enabled. This function executes a series of VMCALLs to enable STM. A VMCALL is an Intel VT-x instruction that allows guest software (Virtual Machine) to make a call for service to its underlying hypervisor. When a VMCALL is executed in VMX root mode, it invokes STM if Dual-Monitor Mode is supported on the processor and the firmware has done an opt-in to STM. If it is not supported or not enabled by the firmware it raises an error [4]. The following steps illustrate the STM initialization procedure in Xen:

1. **Check if firmware has activated Dual-Monitor mode:**

   Bit 0 of IA32_SMM_MONITOR_CTL (0x9B) Model Specific Register (MSR) is the IA-32e mode SMM feature bit. This bit is set by firmware

during initialization process if it does an opt-in for STM. This step also checks whether hardware virtualization, Intel VT-x, is enabled.

2. **Allocate temporary VMCS per logical CPU:**

   Execution of a VMCALL performs some initial checks on the VMCS associated with the VM that invoked this instruction. Intel VT-x does this before transferring control to STM. Since, we are executing in hypervisor's context (VMX root mode), we need to create a temporary VMCS to record the current processor state and exit control information.

3. **Initiate resource list initialization on only one logical processor:**

   On receiving this VMCALL, STM initializes the resource list and sets up the environment for running SMI handler as its guest. This VMCALL must be executed on only one logical CPU.

4. **Execute Start on all logical CPUs:**

   In this step, STM launches the SMM guest. VMCALL for this step must be executed on all the logical CPUs.

5. **Execute protect resources on hypervisor critical resources.**

   In this step, hypervisor specifies a list of its resources that it wants to protect from being tampered by SMI handlers. STM agrees to this resource list if the BIOS has not made a request for any of the resource specified.

All the above steps are performed only if their previous step was successfully executed. If any of the step fails then changes are rolled back and STM is not enabled.

During system reset stage, all the services are stopped by the kernel. Similarly, during this stage, Xen should have a mechanism to safely opt-out of STM, thus disabling it. We implement this opt-out mechanism in the form of an API teardown_stm() before disabling VMX. VMX is disabled by executing VMXOFF instruction. The teardown_stm() API calls Stop STM VMCALL on all the logical CPUs. On receiving this VMCALL, STM does a reverse execution of the steps it performed during initialization time. For hypervisor resource protection, we specify read-only policy on Model Specific Registers (MSRs) and host software code with the ProtectResource VMCALL. This restricts all SMI handlers from modifying host software or MSRs, or Control Registers. The STM utilizes permissions set over the Enhanced Page Tables (EPT), MSR bitmaps, and VMCS controls to limit the SMI handler virtual machine's access to these resources.

4.2.5 Resolving the SMM-Hypervisor Semantic Gap

To detect hypervisor modifications, an SMM based RIMM such as EPA-RIMM should be able to read the hypervisor context during verification. Hypervisors that support Intel's VT-x feature and run hardware-assisted virtualized guests exhibit a unique behavior when interrupted due to SMI. During protected mode operation, the logical processors may be executing either in the VMX root mode or in the VMX non-root mode. On SMI, the logical processor saves the current state of the operating environment internally to them and then enters SMM. This uncertain operating environment state before SMI causes uncertain processor context problem in SMM and thus the semantic gap between SMM and hardware virtualized hypervisors. Moreover, SMM can obtain VMX root

information only by reading the saved processor state. Hence, SMM needs an additional support mechanism to overcome this semantic gap problem and successfully read the the hypervisor state on each SMI.

The authors of HyperSentry [11] solved this problem by instrumenting the SMM code flow. Whenever an SMI is generated from VMX non-root mode, they inject an instruction to force one core to jump from VMX non-root mode to VMX root mode unconditionally. From here the core re-enters SMM thus obtaining the VMX root context. Their technique introduces forced changes to the regular flow and relies on configuring performance counters and the Local APIC. There is the possibility that, if a rootkit is monitoring the SMI counter register, then it can detect the presence of the SMM RIMM and scrub its changes before HyperSentry can identify the rootkit. On each SMI, the SMI counter value gets incremented.

To address this challenge, we develop a stealthier and more straightforward technique to extract hypervisor context. We use STM features to develop this technique. On an STM enabled system, the SMI counter does not get incremented. Thus an SMI generated will go undetected by a rootkit that is monitoring the SMI counter.

To measure the integrity of the hypervisor, the Inspector needs to be able to read the hypervisor's current CPU context. To overcome the semantic gap between the SMM and hypervisor and have guaranteed context of hypervisor state on every SMI, the EPA-RIMM Inspector uses the assistance of STM. In this section, we describe the implementation details of a new function in STM that identifies the hypervisor context and stores the data in a special data structure to be shared with the SMI handler.

On creation of a new Virtual Machine guest, the hypervisor associates a data structure called Virtual Machine Control Structure (VMCS) on a per CPU basis with this guest. This structure keeps a record of the processor state for its respective specific guest. The processor uses this data structure while exiting and entering the guest. Each VMCS has a guest region, host region and control region. The guest region holds information related to the processor behavior for that specific Virtual Machine. The host region holds the processor behavior information for the hypervisor. Finally, the control region holds VM exit, entry, execution control information. On creation of VM, STM saves a record with the new machine's VMCS pointer in its VMCS database with read write policies (degradation policies) as enforced by the hypervisor for the VMCS. Xen and STM both use VMCS in a similar fashion to store and restore the processor context of their respective guests while exiting and entering their virtual guest. We leverage this capability provided by Intel VT-x for reading hypervisor context from STM and sharing it with SMM.

We introduce a new function *Get_Executive_Monitor_Context* that can be invoked via VMCALL interface between the SMI handler and STM. Using this VMCALL, Inspector can request hypervisor context from STM. The VMCALL function retrieves hypervisor specific information either from the Save State area or by loading a Virtual Machine's VMCS and reading the host state region depending on the context from where the SMI was triggered. When the SMI is triggered from VMX root context, the SMI VMCS will have a pointer to VMXON, a special pointer to the hypervisor, and the Save State region will have the hypervisor's processor state. According to Intel VT-x policies, if software

tries to load VMXON it results in a VmFail. Hence, in this case we read the data from Save State area and store it in a special descriptor **MLE_VMM_DESCRIPTOR**. On the other hand, if SMI is triggered from the VMX non-root context, SMI VMCS will hold the pointer to VMCS for the Virtual Machine and the Save State area will be populated with the guest context of VMCS. The guest region of VMCS holds the respective Virtual Machine specific information and the host region holds the hypervisor specific information. In the *Get_Executive_Monitor_Context* function we check the state in which the processor was executing before SMI. If a logical CPU was executing in a Virtual Machine, we load the respective VMCS and read the host region of the VMCS. After completion, we return **MLE_VMM_DESCRIPTOR** in a SMI handler allocated 4K aligned buffer.

This function is executed by invoking VMCALL GetExecutiveMonitorContext with opcode **STM_API_GET_EXECUTIVE_MONITOR_CONTEXT**. The specifications of this VMCALL are as follows:

Input Registers:

EAX = **STM_API_GET_EXECUTIVE_MONITOR_CONTEXT**

EBX = low 32 bits of caller allocated 4K aligned destination buffer.

ECX = high 32 bits of caller allocated 4K aligned destination buffer.

EDX = 0

Output Registers:

CF = 0: No Error, EAX is set to **`STM_SUCCESS`**. Destination buffer contains the VMX-root context.

CF = 1: Error, EAX holds the error value.

The flowchart in **Figure 4.2** explains the flow for VMCALL GetExecutiveMonitorContext.

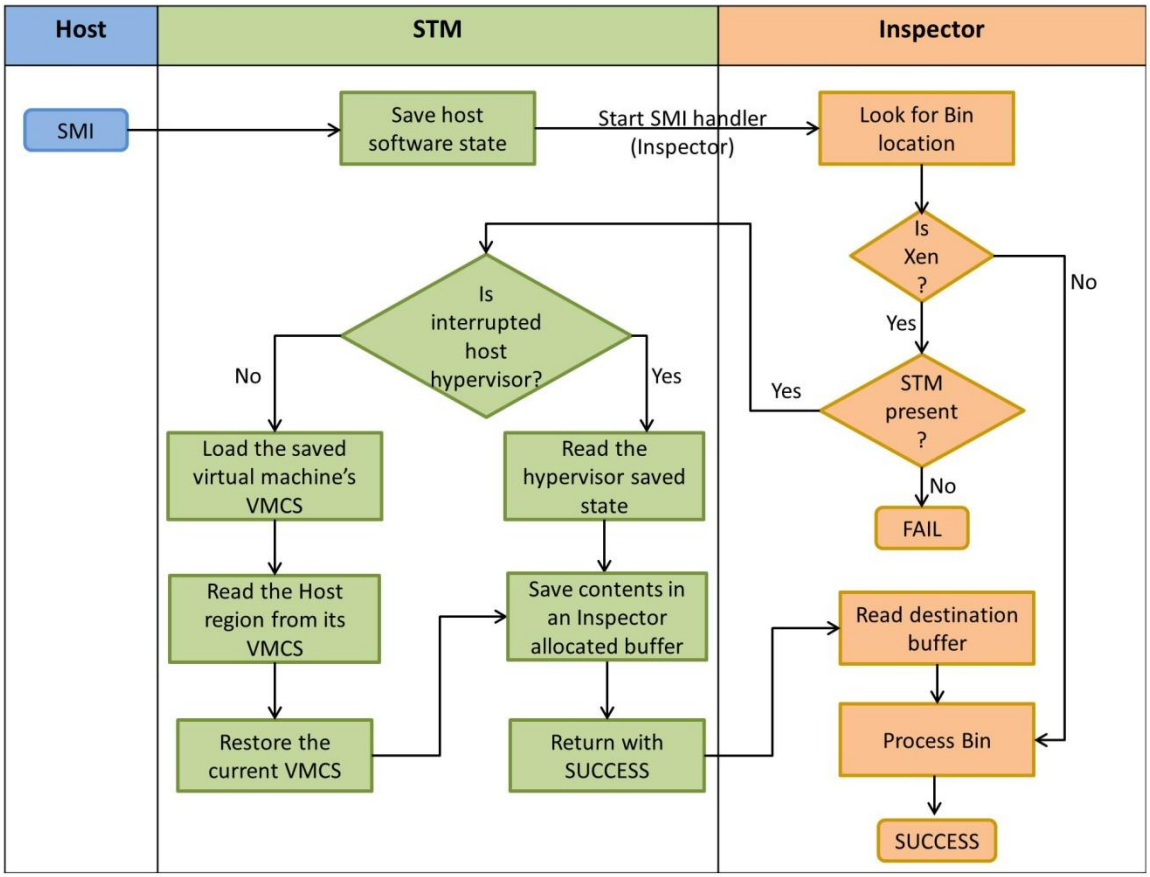

Figure 4.2: GetExecutiveMonitorContext VMCALL Flowchart

4.2.6 STM Performance Gathering

For STM performance analysis, we implement a new VMCALL, GetPerformanceData, between host software and STM. This VMCALL allows the host software to request

performance statistics from STM. These statistics are collected when the Ring0Manager issues a software SMI. Performance data collection is enabled via setting of a knob as part of the UEFI compilation process. This knob allows collecting execution time of different STM events. The current implementation of the performance collector allows for viewing the data only during STM shutdown phase and when UEFI is configured to operate in DEBUG mode. In a production environment, this data cannot be interpreted for analysis purpose by the host software. This VMCALL enables reporting the collected data upto that point to the host software. The host software invokes this VMCALL via opcode **STM_API_GET_PERFORMANCE_DATA**. The specifications of this VMCALL are as follows:

Input Registers:

EAX = **STM_API_GET_PERFORMANCE_DATA**

EBX = low 32 bits of caller allocated 4K aligned destination buffer.

ECX = high 32 bits of caller allocated 4K aligned destination buffer.

EDX = page index

Output Registers:

CF = 0: No Error, EAX is set to **STM_SUCCESS**. Destination buffer contains the VMX-root context.

CF = 1: Error, EAX holds the error value.

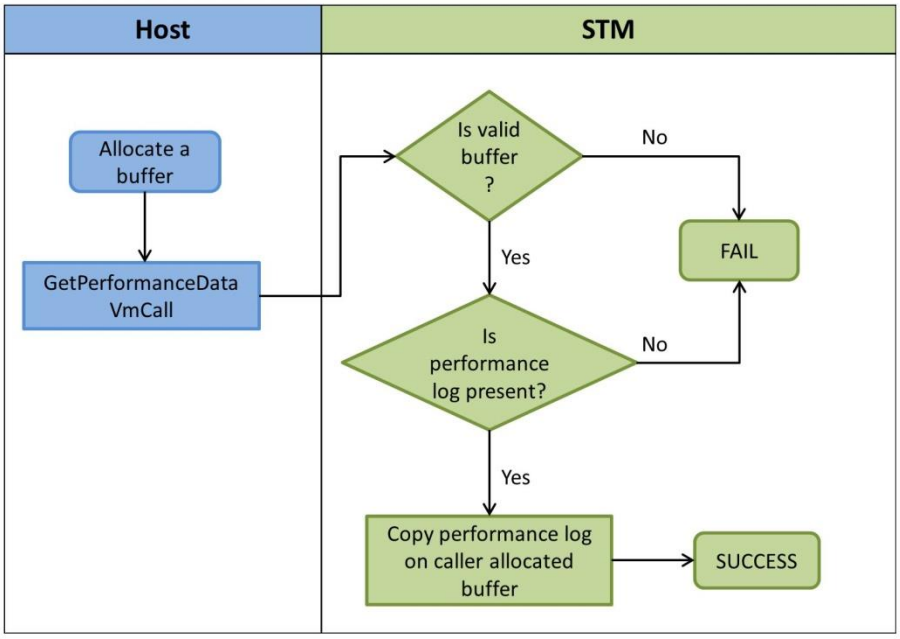

Figure 4.3: GetPerformanceData VMCALL Flowchart

4.2.7 Modifications to EPA-RIMM

In order to use EPA-RIMM successfully to measure the integrity of the hypervisor environment, we make some modifications in its Host Communications Manager and its Inspector components.

4.2.7.1 Host Communications Manager

In the current prototype of EPA-RIMM, the Host Communications Manager is implemented as a ring0 linux kernel module called Ring0Manager. In order to work with hypervisors, we execute Ring0Manager in Xen's Dom0. Ring0Manager takes the *Task-Bin* it received from the Backend Manager and forwards it to the Inspector via SMI. It also receives the results returned from the Inspector, after execution of SMI, back to the Backend Manager. Ring0Manager writes the *Bin* contents on a page and sends the virtual address of this page to the Inspector. Dom0 is the paravirtualized guest and the

management domain of Xen, and hence Xen creates a virtual address space for Dom0. This virtual space of Dom0 may not be contiguous in physical memory. SMM can translate the virtual address belonging to Xen's address space to physical address to read the *Bin*. Hence, we modify Ring0Manager to write the *Bin* on a hypervisor page. In order to write *Bin* to a hypervisor page, we introduce a new hypercall *copy_to_xen.* This hypercall takes the *Bin* from Ring0Manager and the size of the *Bin*, allocates a Xen page and returns the page to the Ring0Manager. Now, the Inspector reads *Bin* from this page, and after completion of its execution writes results back to this page. On returning from SMI, Ring0Manager has copy contents back from the page SMI wrote to the local *Bin* buffer to send results to Backend Manager. We introduce another hypercall *copy_from_xen* that reads the page contents into the local *Bin* buffer.

4.2.7.2 Inspector

When an SMI occurs, the context in which the CPU might be running is indeterminate. On a multi-processor system, on SMI, a CPU is chosen randomly to execute the SMI handler while the other CPUs spin wait for the SMI handler to complete its execution. This CPU may be different from the one on which the HCM generated a software SMI. If the chosen CPU were executing in the HVM guest context, then the context found by the Inspector is that of the HVM guest instead of the hypervisor. To resolve this problem and close the semantic gap problem in SMM, from the Inspector we make a VMCALL GetExecutiveMonitorContext to STM. The STM identifies the hypervisor context from the VMCS of the interrupted host software and returns the critical hypervisor information, CR3, CR4, IDTR, GDTR values, to the Inspector. Thus, our method avoids

using performance counters, modifying the SMI and RSM code flow and dependency on an in-context hypervisor measurement agent. When Inspector receives the *Bin,* it receives *Bin*'s virtual address. Inspector needs to convert this virtual address to physical address to read the *Bin* contents. Using the obtained CR3 information for the CPU currently executing SMI handler, the Inspector finds the physical location of the *Bin*. The Inspector invokes the VMCALL only for the CPU that carries the *Bin* address. This avoids multiple invocations of VMCALL for other CPUs and thus optimizes the Inspector performance. We note that if the Inspector needs the context of other CPUs for measurement purposes, it may invoke the VMCALL to request data for these CPUs. Thus, the Inspector has the context of the hypervisor regardless of the context the CPU was executing in before SMI. With this method, we are able to do out-of context hypervisor inspection. **Figure 4.4** shows the updated EPA-RIMM-V framework with STM enabled on Monitored node.

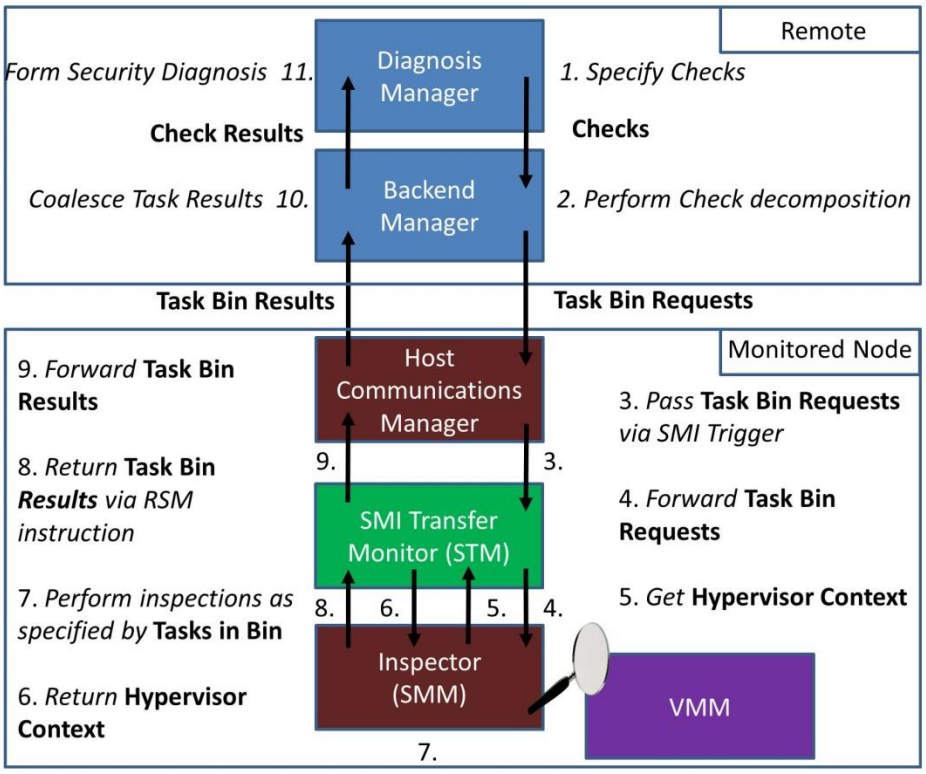

Figure 4.4: EPA-RIMM in STM enabled system – The Inspector on receiving the *Task-Bin* (4) makes a request to STM via VMCALL GetExecutiveMonitorContext to obtain the hypervisor specific processor context (5). This VMCALL services the request and returns results (6). Once the Inspector receives details it reads the *Bin* to perform measurement as specified in the *Task* (7) and returns results (8).

Chapter 5 : Effectiveness

In this chapter, we describe experiments conducted to verify the effectiveness of the prototype designed to detect rootkits in the hypervisor. We demonstrate rootkit detection by EPA-RIMM-V in the hypervisor by simulating known attack behavior. Xen Security Advisories (XSA) regularly releases Xen vulnerabilities and patches for them. The list suggests that there are vulnerabilities present that can cause guest machines to launch attacks based on denial of service, privilege escalations and execution of arbitrary code in the hypervisor [9], [31]. These attacks may be achieved by illegal access of either Xen kernel functions, privileged registers or memory. In our experiment, we simulate attack behavior that modifies kernel memory, the Interrupt Descriptor Table register and the Interrupt Descriptor Table. These simulated attacks may be accomplished by leveraging one or more vulnerabilities in the OS kernel and Xen kernel like CVE-2017-17566, CVE-2017-17045, or CVE-2017-15588. We demonstrate that it is possible to detect such an attack that causes hypervisor resource modification with EPA-RIMM-V. Once, such an anomaly is detected the Diagnosis Manager can alert the administrator of the system being under potential attack.

We perform hypervisor memory and register verification on the prototype running Xen 4.9 and Ubuntu14.04, with Linux kernel 4.11 as Domain0 and Ubuntu 16.04 as HVM guest.

5.1 EPA-RIMM-V Runtime Flow

In order to understand the experiments performed for integrity measurement, we first look at the runtime flow of EPA-RIMM-V. We perform the integrity measurement in two phases, namely, the provisioning phase and the measurement phase.

Provisioning Phase. During the provisioning phase, the Diagnosis Manager initiates a good hash measurement collection *Check*. This *Check* is initiated for kernel code region, Control Registers, and Model-Specific Registers (MSRs). For kernel code memory, we read Xen kernel functions from /proc/xen/xensyms file. This file contains the address mapping of the currently running Xen kernel functions. We find the address location of the kernel in memory and divide it into 4K blocks of memory. We assume that the provisioning phase is performed right after boot, when the system has not yet come under the influence of malicious software. The Diagnosis Manager then provisions these 4K memory blocks as *Checks*. The Backend Manager decomposes these *Checks* into *Tasks* and sends them to the Inspector. Since this is the provisioning phase, the Inspector hashes the current state of the system and records the hash values in the *Task* structure. These initial hashes are stored with the Backend Manager to be later used for comparison during the measurement phase.

Measurement Phase. Once the Backend Manager gets all the good hash values for Xen resources, the measurement phase commences. The Diagnosis Manager now schedules *Checks* for verification of the current state of Xen. These *Checks* may be scheduled periodically or at random intervals. During this phase, the Backend Manager decomposes *Check* into *Tasks* and records original hash values stored for that specific *Task*. This *Task*

could be memory measurement *Task*, control register measurement *Task* or MSR measurement *Task*. The Backend Manager creates a *Bin* of *Tasks*, encrypts it and sends the *Bin* to the Host Communications Manager, which forwards it to the Inspector. When the Inspector receives the *Bin*, it decrypts *Bin* and reads each *Task* to find out what measurement must be performed. The Inspector then hashes the current state of the system and checks it against the hash value passed by the Backend Manager. If the hash values match, Inspector reports the current state as good state in the results field of the *Task*. If the hash values do not match then the Inspector reports this as an error state of the system. Inspector also records entry, exit times and execution times encryption, decryption, hashing functionalities. These statistics are later used to conduct performance analysis. In the next sections we describe Xen rootkit simulation and its detection with EPA-RIMM-V.

5.2 Hypervisor Kernel Code Memory corruption detection

A rootkit may leverage vulnerabilities like Venom, to escalate its privilege and inject shell code to overwrite the original good kernel code with malicious content. We demonstrate the identification of a rootkit that corrupts kernel memory by simulating a hypervisor code corruption attack, modifying function '`xenoprof_log_event`'. During the provisioning phase, EPA-RIMM-V recorded a good hash value for the 4K memory block range containing this function. We then compromise the function memory by overwriting it using the debug capability of EPA-RIMM-V. In subsequent runs, the EPA-RIMM-V Inspector performed a SHA-256 hash on the memory block range and the

Inspector computed current hash value did not match the provisioned hash values. The change was detected and reported to the Diagnosis Manager.

5.3 IDTR corruption detection

The Interrupt Descriptor Table (IDT) is a data structure that maintains the interrupt handler entry points. An appropriate handler is invoked from this table as a response to an interrupt or exception. The Interrupt Descriptor Table Register (IDTR) holds the address for the IDT. This table is fixed at system boot time and does not change. If an attacker can compromise either the IDT or IDTR then potentially a malicious IDT or a malicious interrupt handler may be added. Thus, an attacker would be able to execute the custom malicious handler by invoking the corresponding interrupt. We demonstrate the identification of a rootkit that corrupts IDTR, by simulating an IDTR corruption via a malicious Xen hypercall. This hypercall creates a duplicate copy of the original Interrupt Descriptor Table and points the IDTR to this duplicate IDT. During the provisioning phase when the IDTR measurement is enabled, EPA-RIMM-V collected a good hash value for this register. In subsequent runs when IDTR inspection is scheduled after the exploit has been deployed, the EPA-RIMM-V Inspector performed a SHA-256 hash on the current value of the IDTR and the change was detected.

5.4 IDT corruption detection

Similar to IDTR corruption, an attacker may modify one of the existing handlers in the IDT or replace it with a new interrupt handler. We demonstrate that EPA-RIMM-V can detect a rootkit that modifies the IDT. We simulate an attack to modify an interrupt handler from the IDT. During the provisioning phase, with the IDT measurement

enabled, EPA-RIMM-V collected a good hash value for this table. In subsequent runs after the exploit has been deployed, EPA-RIMM-V Inspector performed a SHA-256 hash on the current value of IDT and the change was detected.

5.5 Conclusions

In this chapter, we demonstrated the capability of EPA-RIMM-V to detect modifications to the kernel code section, IDT register and IDT of the Xen hypervisor.

Chapter 6 : Performance

In this chapter, we describe the experiments conducted to analyze the performance of EPA-RIMM-V. Our focus is on the performance of the Inspector that contains the code running in SMM, since that time must be bounded to avoid potentially serious perturbation. First, we present a performance model for EPA-RIMM-V in Section 6.1, followed by baseline SMM latency guidelines in Section 6.2, description of the performance measurement methodology we apply in Section 6.3 and in Section 6.4 we present our evaluation and results.

6.1 Performance Model

We evaluate the performance of EPA-RIMM-V in an STM-enabled environment using the following model formulated by Delgado and Karavanic [21]:

Equation 1: EPA-RIMM performance model

$$T_m = T_{entry} + T_{work} + T_{exit}$$

T_{entry}: This is the transition time from Xen into SMI handler when an SMI occurs.

T_{work}: This is the time taken by the Inspector to perform verification of the *Bins* provided by the Backend-Manager. This time may vary with the size of the *Task* [21].

T_{exit}: This is the transition time out of the SMI handler and back to the instruction where the processor was interrupted before SMI occurred.

For measuring the STM performance during SMI transitions, we break down T_{entry} and T_{exit} as follows:

Equation 2: T_{entry} for EPA-RIMM-V

$$T_{entry} = SMI_{enter} + STM_{enter} + Inspector_{enter}$$

- SMI_{enter}: Time taken by processor to transition from the source of SMI into STM.
- STM_{enter}: Execution time in STM during T_{entry}
- $Inspector_{enter}$: Time taken by processor to exit STM and enter SMI handler, which is EPA-RIMM-V's Inspector in our prototype.

T_{exit} can be broken down as:

Equation 3: T_{exit} for EP-RIMM-V

$$T_{exit} = Inspector_{exit} + STM_{exit} + SMI_{exit}$$

- $Inspector_{exit}$: Time taken by processor to exit from EPA-RIMM Inspector and enter STM.
- STM_{exit}: Execution time in STM during T_{exit}.
- SMI_{exit}: Time taken by processor to exit STM and resume execution of host software that was interrupted due to SMI.

T_{work} for EPA-RIMM-V from the performance model includes an additional cost of executing the VMCALL GetExecutiveMonitorContext to get the hypervisor's context. Hence the T_{work} can be expressed as:

Equation 4: T_{work} for EPA-RIMM-V

$$T_{work} = T_{work-EPA-RIMM} + T_{VMCALL}$$

- $T_{work-EPA-RIMM}$: Time taken by EPA-RIMM framework to perform measurement.
- T_{VMCALL}: Time taken by VMCALL GetExecutiveMonitorContext to retrieve hypervisor context.

This breakdown of T_m in different components is as shown in **Figure 6.1**

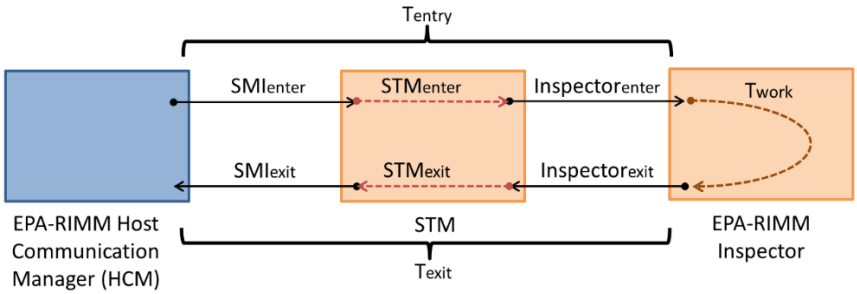

Figure 6.1: SMI Round Trip Time components

6.2 Latency Guidelines

To get a baseline measurement for SMI latency with STM enabled on Minnowboard, we measure the cost of issuing basic software SMI by writing 0 to port 0xB2 and by taking CPU timestamps before and after SMI is issued. Writing 0 to port 0xB2 invokes a no-op SMI handler. Here the processor is interrupted from its current execution sate, transitions into SMI handler and immediately returns to its interrupted state. We measure this cost for a sequence of 1000 SMIs. The average time to process each SMI was found to be 135µs, as opposed to 112µs on a system where STM is not enabled. We attribute this additional time of 23µs to extra work performed by STM and the additional transition costs the processor incurs to transition to and from STM. With this SMI, from the performance model in **Equation 1**, T_{work} is essentially non-existent and the total cost is incurred by T_{entry} and T_{exit}.

6.3 Methodology

We measure the performance of the prototype for a Xen kernel code region (memory) and register measurements. Performance of the prototype depends on the size of measurement performed by the Inspector.

The size of memory measurement is determined by the size of the memory being examined during an SMI as specified in the *Task*. The *Check* decomposition module of the Backend Manager determines the memory decomposition size in the *Task*. We collect a list of Xen kernel code section from /proc/xen/xensyms file for Xen kernel version 4.9. The total size of the Xen kernel code is 4MB. We study the performance impact for memory decompositions of sizes 0.5K, 1KB, 2KB, 4KB and 8KB. For each of these decomposition sizes, we account the wall clock time taken to process one *Task* per *Bin* scheduled at a frequency of one *Bin* per second. Similarly, to measure EPA-RIMM-V's performance while measuring the integrity of Xen registers, we schedule one register measurement *Task* per *Bin* at a frequency of one *Bin* per second. We use instruction 'rdtsc' around SMI trigger instruction in the Ring0Manager to record round trip time taken for each *Bin* per SMI. This round-trip time is T_m. To study T_{entry} and T_{exit} times we record time stamps at various locations in the STM code base where processor enters and exits STM. We record these timestamps in a special STM performance data structure, `STM_PERF_DATA`. Ring0Manager collects these stats from STM with VMCALL GetPerformanceData after the measurement phase.

In the current prototype an SMI is initiated by writing to port B2 from the Host Communications Manager. To get accurate performance data, we need to handle two factors: 1. Processor C-States and 2. Device interrupts.

Processor C-States: Intel CPUs have idle-power saving state known as C-states. These states throttle the power consumed by a core and drop the core in an idle state ranging from C0 to C6. C0 state is the non-idle state while C6 state is the highest idle state. While

transitioning between C-states, processor incurs some latency [67], [68]. To avoid including these impacts in the SMI performance costs, we set the CPU frequency to its highest and set the cores to run in the non-idle C0 state to avoid C-State transition latencies.

Device Interrupts: We observed on Xen that device interrupts were queued for later processing if they fired when the CPU was in SMM. This is because SMIs take precedence over all other interrupts. Once the host software resumes from SMI, the Xen scheduler schedules the previously queued higher priority interrupts before returning to Ring0Manager. This causes a deviation in the T_{exit} time and an anomaly in the *Bin* statistics. The effect of these interrupts on SMI round trip time for 1K memory measurement can be observed in **Figure 6.2**. This effect is also illustrated in **Table 6-1**, where the column 'No Instruction Serialization' shows the T_{exit} times when interrupts are not disabled and the 'Instruction Serialization' column shows the T_{exit} times when interrupts are disabled. Hence, for reliable data collection, we also disable interrupts which effectively serializes the instructions for the round-trip time measurements.

Table 6-1: Effect on T_{exit} time due to Instruction Serialization – All times are in microseconds.

	No Instruction Serialization	**Instruction Serialization**
Mean	149	140
Minimum	60	64
Maximum	481.5	153
Std. Deviation	65	8.5

Figure 6.2: Effect of interrupts due to Xen's virtual machine on SMI round trip time – (Top to Bottom) a) When interrupts are enabled these take priority before exit time is recorded in the Ring0Manager. b) Interrupts are disabled before recording the *Bin* start timestamp and enabled after *Bin* end timestamp is recorded.

6.4 Evaluation Result and Analysis

In this section we evaluate the cost of every component, T_m, T_{work}, T_{entry} and T_{exit}, in the performance model for the EPA-RIMM-V prototype. To study T_{entry} and T_{exit} we enabled performance knob in STM during compile time. Recording performance data itself adds an overhead of 55µs to the baseline SMI latency. T_{entry} and T_{exit} analysis shows the performance of STM and the latency it adds to an SMM-based RIMM virtualized by STM. Later in the section, we present our analysis on the impact of EPA-

RIMM-V on the performance of applications from different categories. We also compared the performance of EPA-RIMM for native kernel and EPA-RIMM-V for hypervisors and present our results in this section.

6.4.1 EPA-RIMM-V SMI Performance Evaluation

In **Figure 6.3** we study the round-trip latencies for measuring Xen memory hash divided into different sizes and CR4 register measurement. These latencies follow the above performance evaluation model from **Equation 1**. This graph is plotted for average latencies obtained after running 500 *Tasks* of each resource. From the graph, we can see that T_{work} increases as the size of the memory we are measuring increases, i.e. size of the work being done by Inspector increases. T_{entry} and T_{exit} remain the same across all the *Tasks*.

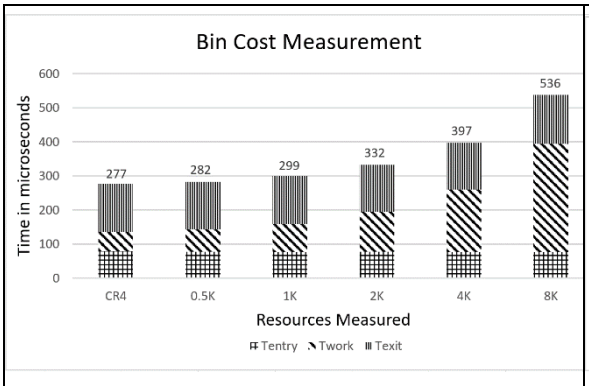

Figure 6.3: SMI Round Trip Time for Xen kernel code region and register measurement. – This figure shows the total *Bin* cost (T_m) as a factor of T_{entry}, T_{work} and T_{exit} as defined in the Performance model.

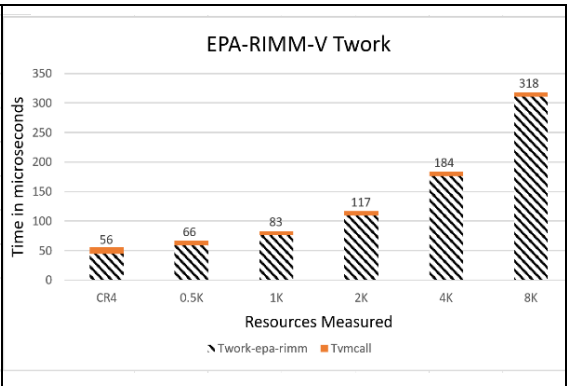

Figure 6.4: Inspector execution time by EPA-RIMM-V – This cost has two components, the original T_{work} of EPA-RIMM Inspector ($T_{work-EPA-RIMM}$) and the additional T_{VMCALL} due to hypervisor context retrieval.

6.4.2 Inspector Performance Evaluation

We study the overhead added by the VMCALL to $T_{work-EPA-RIMM}$ in **Figure 6.4**. EPA-RIMM-V invokes the VMCALL GetExecutiveMonitorContext to obtain the hypervisor context for the processor which is executing the Inspector. We do this to optimize the performance of Inspector. Inspector invokes this VMCALL for locating the *Bin* each time an SMI is fired for EPA-RIMM-V measurement. This VMCALL adds an overhead of approximately 7µs to the total T_{work} accounted under T_{VMCALL}. For memory measurement, Inspector has the required context from the first VMCALL and does not do any more invocations of GetExecutiveMonitorContext VMCALL. During register measurement, Inspector measures the hash of value of the register on all the CPUs. Thus, it requires the context of hypervisor on all the processors. To get context on other CPUs, Inspector invokes GetExecutiveMonitorContext VMCALL to request data for other CPUs. CR4 measurement is a type of register measurement and Inspector measures the hash values of CR4 on all the CPUs on the system. Since our prototype is a dual core system, the Inspector requires only one additional VMCALL during register measurement. Hence, for CR4 measurement T_{VMCALL} adds approximately 12 µs overhead to the total T_{work}.

6.4.3 SMI enter Performance Evaluation

We studied T_{entry} by recording timestamps at various entry points in the STM code execution path. As seen in **Figure 6.3**, T_{entry} remains fairly constant for any type of resource measurement. The time taken during T_{entry} at various stages for 0.5K memory measurement and CR4 measurement is as shown in **Figure 6.5 a) and b)**.

6.4.4 SMI exit Performance Evaluation

We studied T_{exit} by recording timestamps at exit points in the STM code execution path. As seen in **Figure 6.3**, T_{exit} also remains constant for any type of resource measurement. The time taken during T_{exit} at various stages for 0.5K memory measurement and CR4 measurement is as shown in **Figure 6.5 c) and d)**.

For both T_{entry} and T_{exit} STM adds a latency of 4.5µs while context switch from host software to STM and from Inspector to STM in respective cases take most of processor execution time during SMM entry and exit.

6.4.5 Total Memory Measurement Time

For a constant rate of issuing *Bins* at 1 *Task Bin* per second, 8KB queue takes 0.38 seconds to measure the entire Xen kernel code section, whereas 4KB queue takes 0.58 seconds. **Figure 6.6** shows the total time taken to measure Xen kernel memory when divided into different memory sizes. It can be inferred from this graph that it takes longer to measure the entire kernel if partitioning size is small. Even though the per *Bin* measurement time is smaller for smaller memory size *Tasks*, depending on the need and urgency of measuring the memory, the administrator should adjust the memory sizes to be verified at a given time.

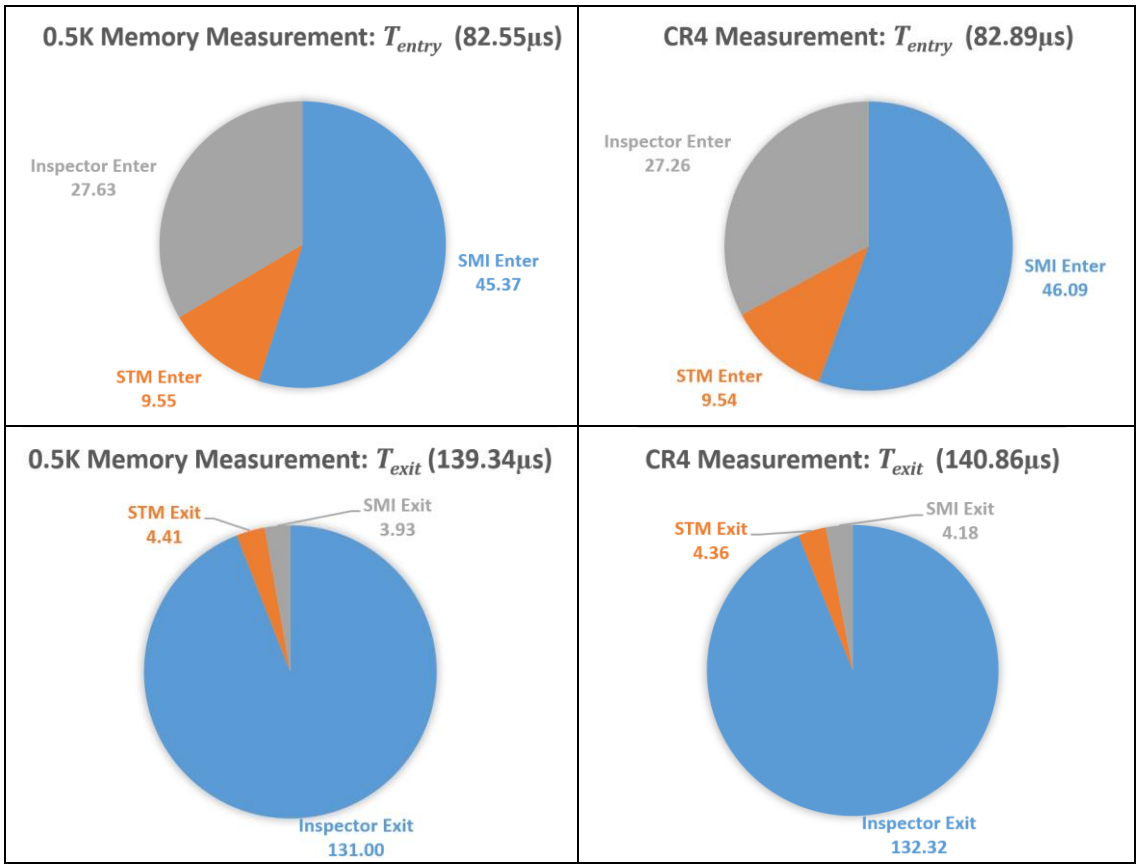

Figure 6.5: T_{entry} and T_{exit} analysis on STM enabled environment. – (Clockwise starting from Top Left) a) T_{entry} for Memory measurement b) T_{entry} for CR4 measurement c) T_{exit} for CR4 measurement d) T_{exit} for Memory measurement

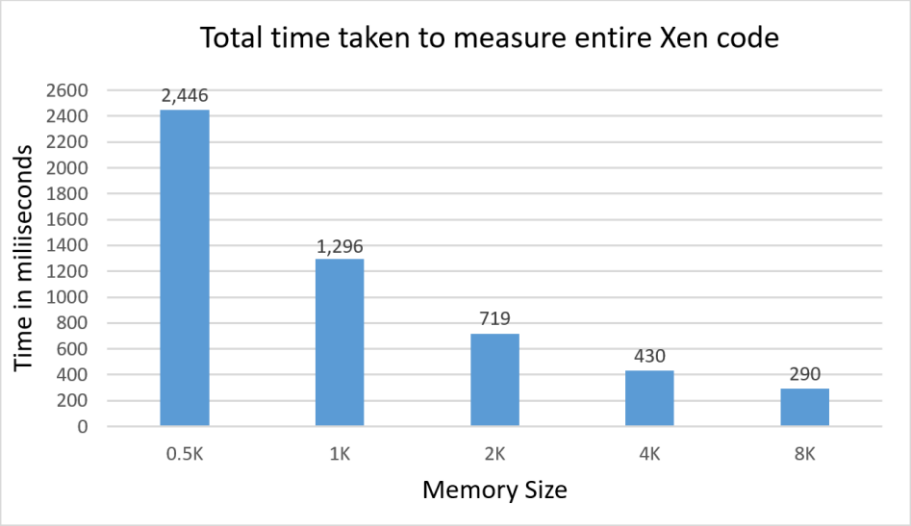

Figure 6.6: Total time to measure entire Xen kernel code memory – Entire memory is partitioned in different chunk sizes from 0.5K to 8K. Each of this chunk is verified separately. The total time to measure all the chunks of a memory decomposition size is displayed in this graph.

6.4.6 Application Performance

We studied the impact of EPA-RIMM-V on application performance by running benchmark applications with the Phoronix test suite. We chose to test the performance of Cachebench, pybench, C-ray and ffmpeg applications. Cachebench exercises memory and cache, pybench tests the system's Python performance, c-ray is a multi-threaded application that exercises the CPU, and ffmpeg performs multi-threaded audio/video encoding. We measured the performance of these applications for light, medium and heavy intensity of SMM measurements and compared them with baseline measurements when there were no SMIs on the system. For light measurement we chose 2 0.5K memory measurements per second, for medium we ran 4 0.5K memory measurements per second and for heavy intensity SMM workload we ran 51 64K memory measurements per second. As seen in the graph in **Figure 6.7**, C-ray and ffpmpeg are most affected under heavy SMM workload. EPA-RIMM-V affects all the applications at minimum for light and medium SMM workload.

6.4.7 EPA-RIMM-V vs. EPA-RIMM Native

We compared the performance of EPA-RIMM with EPA-RIMM-V and found that EPA-RIMM-V on an average adds an overhead of 17 µs to EPA-RIMM's performance because of STM. This overhead is very minor considering there is a virtualization layer in the firmware that constraints the SMI handlers from illegally accessing host side memory and solves the semantic gap problem from hypervisors and SMM. **Figure 6.8** shows the comparison between EPA-RIMM without STM and EPA-RIMM-V.

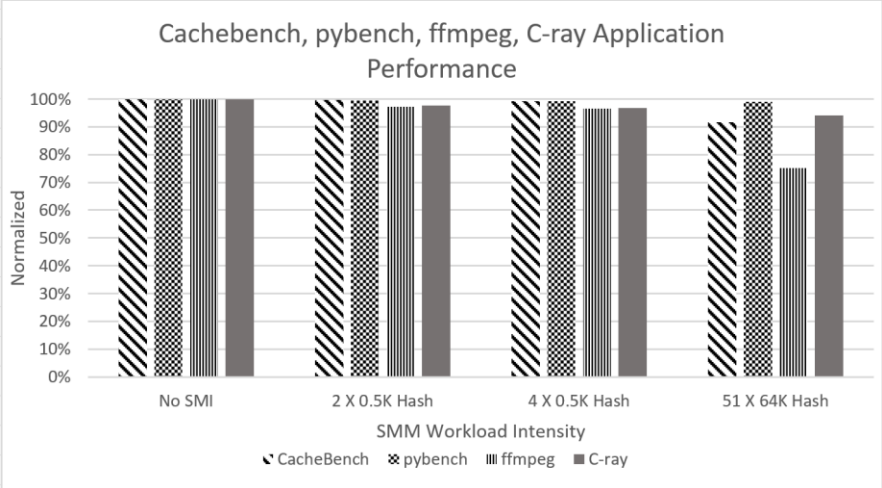

Figure 6.7: Application Benchmark Performance for EPA-RIMM-V

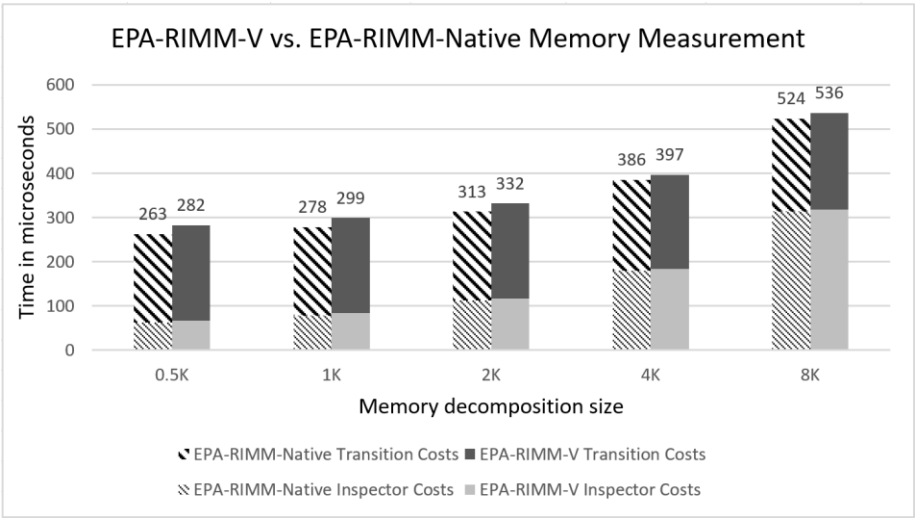

Figure 6.8: EPA-RIMM-V vs. EPA-RIMM-Native Memory Measurement comparison

6.5 Conclusion

Performance results of EPA-RIMM-V shows that our prototype comes very close to meeting the SMI latency guidelines. When SMIs are scheduled less frequently at two or four 0.5K *Tasks* per second, the CPU intensive applications observe a performance degradation of up to 3%. We observed that transitions between host-software and STM and between SMI handler and STM add the maximum latency to the performance of

EPA-RIMM-V. In **Table 6-2** we show a comparison of SMI duration and frequency of different SMM based RIMMs for Virtualized environments. Despite combining EPA-RIMM model's scheduling and decomposition technique, with the SMM de-privileging property of STM and its SMM-hypervisor semantic gap resolution capability, EPA-RIMM-V is able to perform orders of magnitude better than currently researched SMM-based RIMMs. Thus, by applying decomposition and correct host software access policies with STM, EPA-RIMM-V can be an effective practical approach for SMM-based RIMM.

Table 6-2: Comparison of SMI time taken by different SMM RIMMs – Frequency indicates the measurements initiated for a time

SMM RIMM	SMI Duration	Frequency
HyperCheck	40ms	1 per second
HyperSentry	35ms	1 per 8 or 16 second
SPECTRE	5 to 32ms	16 per second to 1 per 5 seconds
EPA-RIMM (no STM) Minnowboard	0.26ms+	Dynamic
EPA-RIMM (with STM) Minnowboard	0.28ms+	Dynamic
Goal	0.15ms	Not specified

Chapter 7 : Security Analysis of EPA-RIMM-V

Assumptions: We assume that this prototype is being run on Intel TXT and Intel TPM capable and enabled the system. Intel TXT assists with static boot time verification of Xen and STM. Hence, we assume that when the system boots it loads a pristine Xen kernel and when Xen is initialized and opts-in to STM the state of the hypervisor is pristine. EPA-RIMM-V gathers the initial good data of the kernel critical data structures during the provisioning process. We also assume that SMRAM is secured by hardware protections and the page table and MSR protections set by STM.

STM: We list some scenarios that could be used to attack STM and how the EPA-RIMM-V infrastructure catches these attacks.

1. **Loading malicious STM:** An attacker could attempt to load a malicious STM via modification of the flash chip if it was not properly protected. However, as the STM is designed to work with Intel TXT and undergoes a measurement by an authenticated code module (ACM) prior to its launch which is reported in TPM PCR 17, Xen would be able to detect that the STM measurement was not consistent with expectations and choose not to launch the STM.
2. **Removing VMCS from STM's VMCS Database:** A Xen rootkit could try to remove the guest VMCS from the STM's VMCS database by firing a ManageVmcsDatabase VMCALL from Xen. However, when the STM is working on GetExecutiveMonitor VMCALL to locate the VMCS in the database, it will fail and would return an error to the Inspector. The Inspector would not be able to produce the specified measurement and will be caught by Backend Manager.

3. **STM Teardown:** A rootkit may attempt to disable STM by launching a command for STM teardown. In case of Xen measurements, Inspector checks whether STM is present. If it finds STM absent, then it does not proceed with the measurement and returns. BEM will catch a missing result and thus it can potentially flag this as STM missing.
4. **Tampering with the VMCS of Guest virtual machines of Xen:** In some scenarios, a hypervisor rootkit could tamper with the VMCS region itself for a VM guest thus attempting to making it difficult for STM to read the guest VMCS for Xen's context.
    a) A rootkit could attempt to delete a VMCS region for a guest, remove its entry from STM's VMCS Database and create a new VMCS for this guest but not register it with STM's VMCS Database. In this case, during the GetExecutiveMonitorContext VMCALL, STM would not be able to locate the VMCS in the database thus returning an error state to Inspector. Inspector on receiving this error will return without proceeding with the measurement, and the BEM will catch it.
    b) In another case, the rootkit may delete the VMCS region for the guest but not delete it from the STM's VMCS database. During the GetExecutiveMonitorContext VMCALL and attempt to read from the VMCS address stored in the database will trigger an error and the Inspector will catch it, and it will be propagated to the BEM.

Chapter 8 : Conclusion

In this section we summarize our work, discuss future work, and conclude.

8.1 Summary

Hypervisors are popular in environments such as Cloud Infrastructure and Data Centers because of their multiple advantages over traditional operating systems like reduced hardware cost and power consumption, better error and disaster recovery. They have also become popular amongst the software intruders' community since hypervisors have access to all the resources of the virtual machines hosted on them. Multiple works in the past have shown that it is possible to compromise different hypervisors mainly because of the vulnerable code introduced in these softwares. Industries are continuously looking for new methods to secure their infrastructure while maintaining performance efficiency.

Although effective, current methodologies are not performant. We developed EPA-RIMM-V an SMM-based RIMM that combines on-chip hardware features of SMM and STM for rootkit detection in the hypervisor. We demonstrated that it is possible to detect rootkits in hypervisors using a novel technique implemented in STM with minimum SMI performance overhead. We implemented a prototype for gathering of out-context hypervisor state information. This method takes approximately 7μs to execute in SMM and solves the problem of uncertain processor state in SMM for VT-x enabled hypervisors. We implemented STM enabling in Xen, which is currently not present in Xen's code base. This enabling feature can be easily ported to other hypervisors as well. We also present a full study of STM performance evaluation that can help other researchers understand the overall impact of STM to the system's performance. We

developed a new performance collector interface between STM and Xen that can be used by analysts for studying performance impact due to different STM modules.

8.2 Future Work

This research opens new possibilities of extending EPA-RIMM-V to solve other problems. One such problem is the capability of performing Virtual Machine integrity measurement from SMM with EPA-RIMM-V. Virtual Machines serve as the primary workspace for running user applications. Earlier studies [58], [59], [60], [61] have shown the necessity of securing the kernel from malicious applications. Malicious applications and kernels serve as one of the primary sources to leverage hypervisor vulnerabilities and launch attacks against the infrastructure itself. For these reasons, it is required even to monitor the security of the virtual machines. By obtaining the virtual machine semantics from the hypervisor, monitoring of virtual machine can be accomplished [54], [56], [57]. As we have seen, even hypervisors are vulnerable and placing a monitoring unit in a vulnerable hypervisor may make the integrity checker vulnerable to attacks. SMM is comparatively more secure since it is difficult to detect and compromise. Also, with STM, SMM can be hardened by virtualizing it and imposing access restriction rules.

This research may also be extended to monitor the SMI handlers. The current techniques rely on the presence of a co-processor and SMM instrumentation to periodically send the status information to the monitoring unit on the co-processor [62]. Using EPA-RIMM-V's framework, the SMM monitor may reside in the STM and periodically check the health of the handlers. This technique avoids having to rely on extra hardware on the system and implement methods to overcome the semantic gap between the main

processor and the co-processor. Implementing a VM monitor and SMM monitor would make EPA-RIMM an end-to-end solution for complete system monitoring via SMM.

Researchers could use the performance collector interface between STM and Xen to study the performance behavior of different modules of STM and UEFI and their impact on overall SMM execution.

8.3 Conclusions

In this thesis we have shown that a combination of SMM and STM can be used for performance efficient, reasonably secure, configurable integrity measurement of hypervisors. The key contributions of this thesis are:

1. **Developed a novel technique to solve the SMM-hypervisor semantic gap problem:**

   In this research, we implemented a novel technique that uses STM, a firmware based feature, to obtain out-of-context hypervisor status from SMM and eliminates the uncertain processor context problem. This technique is the basis for performing stealthy out-of-context hypervisor integrity checking. It eliminates the need of using special hardware components and avoids injecting instructions in the normal SMI exit code flow as implemented in HyperSentry [11]. Other SMM applications may also use this technique to retrieve hypervisor context. The usage of this module itself has a very low performance overhead. We plan to open source this module and integrate it with the current STM code base.

2. **Implemented STM enabling in hypervisor and developed a set of fine-grained permissions that should be set for hypervisor resources:**

a. We implemented the missing hypervisor-STM handshake mechanism in Xen's kernel.
b. We developed a set of permissions that should be set over hypervisor resources to avoid illegal access by SMI handlers. We plan to release a patch of this work to Xen Project team.

3. **Implemented a performance collector interface between STM and the host software and conducted a full investigation of STM performance:**

   To get the STM performance data in the host software we implemented an API between the hypervisor and STM. Using the data obtained from this interface, we present a detailed performance cost analysis of using STM. To the best of our knowledge, this is the first STM performance study available for an STM configured system with cooperation between hypervisor and BIOS.

4. **Developed a prototype of EPA-RIMM-V:**

   We designed and implemented a prototype of EPA-RIMM-V, and used it to perform integrity checking of memory, registers, and model specific registers for Xen hypervisor. We evaluated the effectiveness of our model by simulating two rootkits that are representative of family of hypervisor rootkits and detected them in subsequent measurement invocations. We plan to release the prototype as open source enabling researchers to experiment with it and further advance the research.

Chapter 9 : References

[1] Obasuyi, G. C., & Sari, A. (2015). Security challenges of virtualization hypervisors in virtualized hardware environment. *International Journal of Communications, Network and System Sciences*, *8*(07), 260.

[2] https://www.cvedetails.com/vulnerability-list/vendor_id-6276/opec-1/XEN.html

[3] Duflot, L., Etiemble, D., & Grumelard, O. (2006). Using CPU system management mode to circumvent operating system security functions. *CanSecWest/core06*.

[4] Intel 64 and IA-32 Architectures Software Developer's Manual, September 2016.

[5] Ormandy T: An empirical study into the Security exposure to hosts of hostile virtualized environments. In *CanSecWest applied Security conference*. Vancouver; 2007. http://taviso.decsystem.org/virtsec.pdf.

[6] Le, C. H. H. (2009). *Protecting xen hypercalls* (Doctoral dissertation, UNIVERSITY OF BRITISH COLUMBIA (Vancouver).

[7] Jin, S., Seol, J., Huh, J., & Maeng, S. (2015, March). Hardware-Assisted Secure Resource Accounting under a Vulnerable Hypervisor. In *ACM SIGPLAN Notices* (Vol. 50, No. 7, pp. 201-213). ACM.

[8] Delgado, B., & Karavanic, K. L. (2013, September). Performance implications of system management mode. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on* (pp. 163-173). IEEE.

[9] http://venom.crowdstrike.com/

[10] Xen Security Advisories (XSA): https://xenbits.xen.org/xsa/

[11] Azab, A. M., Ning, P., Wang, Z., Jiang, X., Zhang, X., & Skalsky, N. C. (2010, October). HyperSentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security* (pp. 38-49). ACM.

[12] Wang, J., Stavrou, A., & Ghosh, A. (2010, September). Hypercheck: A hardware-assisted integrity monitor. In *International Workshop on Recent Advances in Intrusion Detection* (pp. 158-177). Springer Berlin Heidelberg.

[13] Rutkowska, J., & Wojtczuk, R. (2008). Preventing and detecting Xen hypervisor subversions. *Blackhat Briefings USA*.

[14] F. Zhang, K. Leach, K. Sun, and A. Stavrou, "SPECTRE: A dependable Introspection Framework via System Management Mode," DSN, Budapest, Hungary, 2013.

[15] SMI Transfer Monitor (STM) User Guide, August 2015, Revision 1.00

[16] https://minnowboard.org/

[17] https://krebsonsecurity.com/tag/mirai-botnet/

[18] https://en.wikipedia.org/wiki/WannaCry_ransomware_attack

[19] http://www.wired.co.uk/article/judy-malware-android

[20] https://github.com/chipsec/chipsec/wiki/1.2.4

[21] Delgado, B., & Karavanic, K. L. (2018, May). EPA-RIMM: A Framework for Dynamic SMM-based Runtime Integrity Measurement, Technical Report, arXiv:1805.03755 [cs.CR], available at: http://arxiv.org/abs/1805.03755.
[22] Wang, Z., & Jiang, X. (2010, May). Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In Security and Privacy (SP), 2010 IEEE Symposium on (pp. 380-395). IEEE.
[23] Deng, L., Liu, P., Xu, J., Chen, P., & Zeng, Q. (2017, April). Dancing with Wolves: Towards Practical Event-driven VMM Monitoring. In Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (pp. 83-96). ACM
[24] Zhang, F., Chen, J., Chen, H., & Zang, B. (2011, October). CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (pp. 203-216). ACM.
[25] Szefer, J., & Lee, R. B. (2012, March). Architectural support for hypervisor-secure virtualization. In ACM SIGPLAN Notices (Vol. 47, No. 4, pp. 437-450). ACM.
[26] MODEL, I. T. A Multi-Layered Architecture for a Secure Virtualization Environment.
[27] http://www.tianocore.org/
[28] https://firmware.intel.com/projects/minnowboard-max
[29] https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview
[30] STM Source Code: https://github.com/jyao1/STM
[31] Xen Memory corruption, Information leak Vulnerabilities: https://www.debian.org/security/2017/dsa-4050
[32] IDT-hook attack: http://phrack.org/issues/59/4.html, http://www.phrack.org/issues/69/15.html, https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2015/february/adventures-in-xen-exploitation/
[33] Thongthua, A., & Ngamsuriyaroj, S. (2016, May). Assessment of Hypervisor Vulnerabilities. In Cloud Computing Research and Innovations (ICCCRI), 2016 International Conference on (pp. 71-77). IEEE.
[34] Ibrahim, A. S., Hamlyn-Harris, J., & Grundy, J. (2016). Emerging security challenges of cloud virtual infrastructure. arXiv preprint arXiv:1612.09059.
[35] Xen Project 4.9 release: https://www.linuxfoundation.org/press-release/new-features-in-xen-project-4-9-provide-better-usability-in-automotive-and-embedded/
[36] De Souza, W. A. R., & Tomlinson, A. (2015, November). SMM-based hypervisor integrity measurement. In Cyber Security and Cloud Computing (CSCloud), 2015 IEEE 2nd International Conference on (pp. 362-367). IEEE.
[37] Bonkoski, A., Bielawski, R., & Halderman, J. A. (2013, August). Illuminating the Security Issues Surrounding Lights-Out Server Management. In WOOT.

[38] Petroni Jr, N. L., Fraser, T., Molina, J., & Arbaugh, W. A. (2004, August). Copilot-a Coprocessor-based Kernel Runtime Integrity Monitor. In USENIX Security Symposium (pp. 179-194).
[39] Greene, J. (2010). Intel Trusted Execution Technology: Hardware-based Technology for Enhancing Server Platform Security. *Intel Corporation, Copyright*, *2012*(8).
[40] https://www.apriorit.com/qa-blog/223-virtualization-in-testing
[41] http://www.datacenterknowledge.com/archives/2015/10/05/hpc-virtualization-use-cases-best-practices
[42] https://www.automationworld.com/virtualization-it-ot-crossroads
[43] http://www.datacenterknowledge.com/google-alphabet/google-cloud-platform-introduces-96-cpu-machines
[44] Heiser, G. (2009, January). Hypervisors for consumer electronics. In Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE (pp. 1-5). IEEE.
[45] Godfrey, M. W., & German, D. M. (2008, September). The past, present, and future of software evolution. In *Frontiers of Software Maintenance, 2008. FoSM 2008.* (pp. 129-138). IEEE.
[46] https://www.darwins-theory-of-evolution.com/
[47] Steinberg, U., & Kauer, B. (2010, April). NOVA: a microhypervisor-based secure virtualization architecture. In Proceedings of the 5th European conference on Computer systems (pp. 209-222). ACM.
[48] Intel Chips timeline: https://www.intel.com/content/www/us/en/history/history-intel-chips-timeline-poster.html, *2012*.
[49] http://www.tomshardware.com/picturestory/784-intel-chipset-history.html#s27
[50] Myers, E.D., (2017 August). STM/PE and XHIM. Poster presented at the USENIX Security Symposium, Vancouver, BC.
[51] Pendergrass, J. A., & McGill, K. N. (2013). LKIM: The Linux Kernel Integrity Measurer. Johns Hopkins APL technical digest, 32(2), 509-516.
[52] Azab, A. M., Ning, P., & Zhang, X. (2011, October). Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In Proceedings of the 18th ACM conference on Computer and communications security (pp. 375-388). ACM.
[53] Zhang, F., & Zhang, H. (2016, June). SoK: A Study of Using Hardware-assisted Isolated Execution Environments for Security. In Proceedings of the Hardware and Architectural Support for Security and Privacy 2016 (p. 3). ACM.
[54] Anderson, R. J. (2010). *Security engineering: a guide to building dependable distributed systems*. John Wiley & Sons.
[55] Garfinkel, T., & Rosenblum, M. (2003, February). A Virtual Machine Introspection Based Architecture for Intrusion Detection. In Ndss (Vol. 3, No. 2003, pp. 191-206).

[56] Jiang, X., Wang, X., & Xu, D. (2007, October). Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In Proceedings of the 14th ACM conference on Computer and communications security (pp. 128-138). ACM.
[57] Tang, H., Feng, S., Zhao, X., & Jin, Y. (2017). VirtAV: an Agentless Runtime Antivirus System for Virtual Machines. KSII Transactions on Internet & Information Systems, 11(11).
[58] http://www.hsc.fr/ressources/outils/rkscan/
[59] http://seclists.org/incidents/2000/Oct/165
[60] Lawless, T. St Michael: detection of kernel level rootkits: https://github.com/tomasz-janiczek/stmichael-lkm
[61] Huangang, X. (2002). Building a secure system with LIDS. Linux Intrusion Detection System.: http://www.de.lids.org/document/build_lids-0.2.html
[62] Chevalier, R., Villatel, M., Plaquin, D., & Hiet, G. (2017, December). Co-processor-based Behavior Monitoring: Application to the Detection of Attacks Against the System Management Mode. In Proceedings of the 33rd Annual Computer Security Applications Conference (pp. 399-411). ACM.
[63] Zhang, F., Leach, K., Stavrou, A., Wang, H., & Sun, K. (2015, May). Using hardware features for increased debugging transparency. In *Security and Privacy (SP), 2015 IEEE Symposium on* (pp. 55-69). IEEE.
[64] Zhang, F., Leach, K., Wang, H., & Stavrou, A. (2015, April). Trustlogin: Securing password-login on commodity operating systems. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (pp. 333-344). ACM.
[65] Unified Extensible Firmware Interface System Management Mode Initialization Protections with System Management Interrupt Transfer Monitor Sandboxing. : https://patents.google.com/patent/US20170132164
[66] Zhenyu Ning, Fengwei Zhang, Weisong Shi, and Weidong Shi. 2017. Position Paper: Challenges Towards Securing Hardware-assisted Execution Environments. In Proceedings of the Hardware and Architectural Support for Security and Privacy (HASP '17). ACM, New York, NY, USA, Article 6, 8 pages.
[67] https://software.intel.com/en-us/articles/power-management-states-p-states-c-states-and-package-c-states
[68] Schöne, R., Molka, D., & Werner, M. (2015). Wake-up latencies for processor idle states on current x86 processors. *Computer Science-Research and Development*, *30*(2), 219-227.
[69] Schatz, D., Bashroush, R., & Wall, J. (2017). Towards a More Representative Definition of Cyber Security. Journal of Digital Forensics, Security and Law, 12(2), 8.
[70] Chisnall, D. (2008). *The definitive guide to the xen hypervisor*. Pearson Education.
[71] Intel Trusted Execution Technology (Intel TXT) Software Development Guide, November 2017, Revision 015.

[72] https://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf
[73] Wojtczuk, R., & Rutkowska, J. (2009). Attacking intel trusted execution technology. *Black Hat DC, 2009*.
[74] Loucaides, J. "BIOS and Secure Boot Attacks Uncovered", *Ruxcon 2014, Melbourne, Australia,* https://ruxcon.org.au/assets/2014/slides/rux-BiosAttackSummary-ruxcon2014.pdf .
[75] Kallenberg, C., Kovah, Xeno. "How Many Million BIOSes Would you Like to Infect?", *CanSecWest 2015, Vancouver, Canada, 2015*.
[76] http://www.kb.cert.org/vuls/id/912156
[77] "Lenovo ThinkPad System Management Mode arbitrary code execution exploit", Cr4sh, https://github.com/Cr4sh/ThinkPwn.
[78] Xen Code Review Dashboard: xen.biterg.io
[79] AMD, AMD64 Architecture Programmer's Manual, Volume 2: System Programming.
[80] J. Triplett and B. Triplett, "BITS: BIOS Implementation Test Suite,"http://www.linuxplumbersconf.org/2011/ocw/system/presentations/867/original/bits.pdf