3-24-2020

Extensible Performance-Aware Runtime Integrity Measurement

Brian G. Delgado
Portland State University

Extensible Performance-Aware Runtime Integrity Measurement

by

Brian G. Delgado

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

Doctor of Philosophy
in
Computer Science

Dissertation Commitee:
Karen L. Karavanic, Chair
Charles V. Wright
Wu-chang Feng
Bruce Irvin
Wayne Wakeland

Portland State University
2020

Abstract

Today's interconnected world consists of a broad set of online activities including banking, shopping, managing health records, and social media while relying heavily on servers to manage extensive sets of data. However, stealthy rootkit attacks on this infrastructure have placed these servers at risk. Security researchers have proposed using an existing x86 CPU mode called System Management Mode (SMM) to search for rootkits from a hardware-protected, isolated, and privileged location. SMM has broad visibility into operating system resources including memory regions and CPU registers. However, the use of SMM for runtime integrity measurement mechanisms (SMM-RIMMs) would significantly expand the amount of CPU time spent away from operating system and hypervisor (host software) control, resulting in potentially serious system impacts. To be a candidate for production use, SMM RIMMs would need to be resilient, performant and extensible. We developed the EPA-RIMM architecture guided by the principles of extensibility, performance awareness, and effectiveness. EPA-RIMM incorporates a security check description mechanism that allows dynamic changes to the set of resources to be monitored. It minimizes system performance impacts by decomposing security checks into shorter tasks that can be independently scheduled over time. We present a performance methodology for SMM to quantify system impacts, as well as a simulator that allows for the evaluation of different methods of scheduling security inspections. Our SMM-based EPA-RIMM prototype leverages insights from the performance methodology to detect host software rootkits at reduced system impacts. EPA-RIMM demonstrates that SMM-based rootkit detection can be made performance-efficient and effective, providing a new tool for defense.

Acknowledgements

This dissertation would not have been possible without the support of many people. Professor Karavanic greatly helped encourage the work forward and navigate the complex intersection of performance and computer security.

I began this academic journey due to the encouragement of my former Intel manager, Jeff Demain, who encouraged me to pursue graduate studies. I also had the fortune to learn about performance measurement while working with Raed Kanjo at Intel who taught me techniques that were invaluable in this analysis. Dave Riss, Dion Rogers, and Chris Kachigian were instrumental in navigate my academic path at Intel in recent years.

The patience of my family has been helpful along the way in dealing with the long hours. My uncle, David, encouraged me on to complete the work and also gave me inspiration due to his own academic path.

I gratefully acknowledge the help of the entire EPA-RIMM team: Tejaswini Vibhute for many helpful discussions on EPA-RIMM and STM, John Fastabend for diving into multicore and coreboot, Cody Shepherd for significantly refining the software stack and many helpful questions, Dylan Abraham for leading the release process for EPA-RIMM and getting the code ready, Payal Joshi for discussions on measurement triggers and the Oracle, Alex Freed for his inputs and jumping in to test the release code, and our intern crew for finding interesting corners of this project to explore.

Grant Information:

"This material is based upon work supported by the National Science Foundation under Grant No. 1528185.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation."

Table of Contents

List of Tables

List of Figures

List of Abbreviations

Abbreviation	Definition	Description
API	**A**pplication **P**rotocol **I**nterface	Software interface
BEM	**B**ack **E**nd **M**anager	Decomposes EPA-RIMM Checks into Tasks
BIOS	**B**asic **I**nput **O**utput **S**ystem	Firmware code that initializes the system
BMC	**B**aseboard **M**anagement **C**ontroller	Out-of-band server management interface
CR0	**C**ontrol **R**egister **0**	Controls write protection of memory pages and paging
CR3	**C**ontrol **R**egister **3**	Specifies page table address
CR4	**C**ontrol **R**egister **4**	Enables for Supervisor Mode protections and other security/functional features
CVE	**C**ommon **V**ulnerabilities **E**xposures	Database of known vulnerabilities
DKOM	**D**irect **K**ernel **O**bject **M**anipulation	Rootkit technique to hide traces from system components
DM	**D**iagnosis **M**anager	Selects EPA-RIMM Checks for processing
DMA	**D**irect **M**emory **A**ccess	Method of accessing memory without CPU involvement
EPA-RIMM	**E**xtensible **P**erformance **A**ware - **R**untime **I**ntegrity **M**easurement **M**echanism	A rootkit detector that limits time spent in a SMM session and supports extensible measurements
FCFS	**F**irst **C**ome **F**irst **S**erved	Process items in order
FDC	**F**loppy **D**rive **C**ontroller	Controls floppy drive
FEM	**F**ront **E**nd **M**anager	Interfaces between Ring 0 Manager and Backend Manager
GDT	**G**lobal **D**escriptor **T**able	Provides global segment descriptors
GDTR	**G**lobal **D**escriptor **T**able **R**egister	Provides address of table of global segment descriptors
HCM	**H**ost **C**ommunication **M**anager	Generates SMI to trigger measurement

HMAC	**H**ash-based **M**essage **A**uthentication **C**ode	Verifies data integrity and authenticity of a message
IDT	**I**nterrupt **D**escriptor **T**able	Provides addresses of interrupt service routines
IDTR	**I**nterrupt **D**escriptor **T**able **R**egister	Provides address of table of interrupt service routines
IOMMU	**I**/**O** **M**emory **M**anagement **U**nit	Provides memory virtualization for devices
KASLR	**K**ernel **A**ddress **S**pace **L**ayer **R**andomization	Technique to randomize kernel code addresses
KPCR	**K**ernel **P**rocessor **C**ontrol **R**egion	Maintains processor state
LDT	**L**ocal **D**escriptor **T**able	Provides local memory segment descriptors
MAC	**M**essage **A**uthentication **C**ode	Authenticates a message
MSR	**M**odel **S**pecific **R**egister	CPU register that controls processor behavior or reports statistics
OS	**O**perating **S**ystem	Privileged software that supports fundamental system control operations
PCI	**P**eripheral **C**omponent **I**nterconnect	A type of bus that connects peripherals
PCR	**P**latform **C**onfiguration **R**egister	A protected storage location on the TPM
OUTB	**OUT B**yte	Write a byte to an IO port
QEMU	**Q**uick **E**mulator	Machine emulator
QOS	**Q**uality **O**f **S**ervice	A performance measurement over a service
RIMM	**R**untime **I**ntegrity **M**easurement **M**echanism	Performs inspections at runtime to detect deviations
SMAP	**S**upervisor **M**ode **A**ccess **P**rotection	Prevents Ring 0 code from accessing Ring 3 memory
SMEP	**S**upervisor **M**ode **E**xecution **P**rotection	Prevents Ring 0 code from executing Ring 3 memory
SMI	**S**ystem **M**anagement **I**nterrupt	Trigger to enter SMM
SMM	**S**ystem **M**anagement **M**ode	Privileged x86 execution mode

SMM-RIMM	**S**ystem **M**anagement **M**ode - **R**untime **I**ntegrity **M**easurement **M**echanism	SMM-based runtime integrity checking method
SMRR	**S**ystem **M**anagement **R**ange **R**egister	Protects SMM memory from external access
SPI	**S**erial **P**eripheral **I**nterconnect	Connects BIOS flash to system
SMRAM	**S**ystem **M**anagement **R**andom **A**ccess **M**emory	SMM memory
SSD	**S**olid **S**tate **D**isk	Solid state disk
SSDT	**S**ystem **S**ervices **D**escriptor **T**able	Provides addresses for Windows system services
STM	**S**MI **T**ransfer **M**onitor	SMM-based hypervisor that applies policy over SMI handler accesses
SWSMI	**S**oft**W**are **SMI**	Chipset SMI generation source
TSC	**T**ime **S**tamp **C**ounter	Counts CPU clocks
TPM	**T**rusted **P**latform **M**odule	Discrete security chip that stores keys and performs integrity measurements
TZ-RKP	**T**rustZone-based **R**eal-time **K**ernel **P**rotection	Runtime integrity mechanism for TrustZone
UEFI	**U**nified **E**xtensible **F**irmware **I**nterface	Standards-based firmware interface
VM	**V**irtual **M**achine	Method of running an operating system over an abstracted layer
VMM	**V**irtual **M**achine **M**onitor	Abstraction layer that virtualizes operating systems

List of Symbols

μs microseconds

1
Introduction

Today's interconnected world presents significant opportunities for digital interactions. A broad set of online activities including banking, shopping, managing health records, and social media rely heavily on servers to store and transmit rapidly growing sets of personal information. Both the hardware and software stack for typical servers have grown in complexity over the past decade: Virtualization has added a layer of system software between the hardware and operating systems; and multicore processors have led to a focus on multithreading and unprecedented levels of resource sharing. Firmware has taken a prominent role in system configuration and providing runtime services to operating systems.

While researchers have made advances in improving the security features of computing platforms, attackers have achieved extensive success with system compromises and ability to persist undetected. The varied and dramatic attacker exploits have caused a new realization: "Companies are beginning to accept that they will be compromised, so the demand is growing to know just how often and how deep… [98]" The trend towards increased focus on intrusion detection is also reflected in corporate spending: "Enterprises are transforming their security spending strategy in 2017, moving away from prevention-only approaches to focus more on detection and response, according to Gartner, Inc. [33]."

However, while increased focus is being spent on detection, there are a variety of examples where current detection capabilities are lacking. In December 2016, attackers calling themselves "TheDarkOverlord" contacted the president of Larson Studios to tell them that their servers had been compromised and unless ransom was paid, the attackers would leak all of their

data which included upcoming new episodes of the popular Netflix series, "Orange is the New Black". The attackers also deleted the episodes from the compromised servers. The company sent $50K to the attackers in an unsuccessful effort to prevent the online release [98]. In a more concerning attack, the same group compromised the Cancer Services of East Central Indiana – Little Red Door, retrieved data, and encrypted the original data, demanding a ransom for restoration. The agency declined to pay noting that they "will not pay a ransom when all funds raised must instead go to serving families, all stage cancer clients, late stage care/hospice support and preventative screenings... [22]" These compromises are representative of inadequate computer defenses and detection capabilities. The compromised organizations did not detect the attack themselves, but rather from the hackers.

Attacks can appear in a variety of forms including computer viruses, ransomware, and rootkits. Rootkits present a special concern as they are designed to evade detection and provide attackers with a direct channel into the system to return undetected. Security researchers focusing on rootkit detection have endeavored to provide new detection mechanisms at lower levels in the platform to better observe malicious code while remaining protected from it. These mechanisms can reside within the operating system, hypervisor, or at an even more privileged level, system firmware (also commonly referred to as BIOS). Recent years have also seen a growth in ransomware including instances that leverage kernel vulnerabilities such as WannaCry, Sage, Locky, and Bad Rabbit [87, 9, 63, 105]. These ransomware also modify privileged operating system resources to gain control of the system and provide a place for their malware to reside.

A key challenge with securing computer systems is that vulnerabilities such as buffer overflows, integer overflows, improper input checking, and

inadequate testing can allow an attacker to gain privileges such that they can overwrite host software code with malicious code. This ongoing issue facilitates rootkits as attackers can simply select a suitable code vulnerability and extend it to facilitate greater control over the system or hide their traces. These insecurities in host software design may never be completely resolved as they are inherent in complex software. As details on kernel and hypervisor rootkit prevalence are not readily available, we focus on survey data from the Common Vulnerabilities and Exposure (CVE) database [19] to gather counts of privilege escalation and arbitrary code execution vulnerabilities in Xen and Linux from the years 2011 to 2019. This analysis provides the basis for examining the scale of the software vulnerabilities that can breed kernel and hypervisor rootkits. The data, as shown in Figure 1.1, clearly shows that the scale of the problem has increased since 2011.

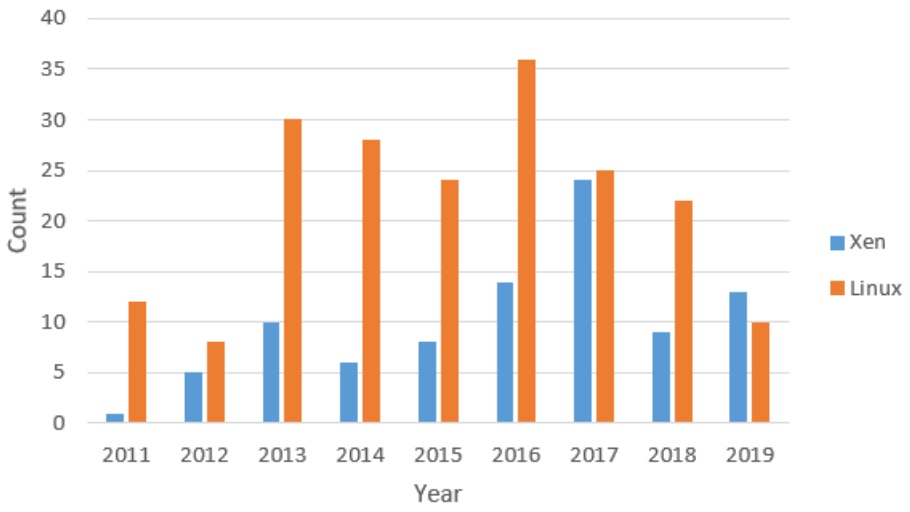

FIGURE 1.1: Survey of CVE Database for Privilege Escalations and Arbitrary Code Execution for Xen and Linux (Years 2011-2019)

These vulnerabilities are initial vectors that an attacker can weaponize by inserting modified code into the host to gain persistence and stealth. With a vulnerable system and rootkit techniques, the attacker has the two necessary

requirements to obtain persistent control over host software. Attackers seek out privileged data structures such as the System Services Descriptor Table (SSDT) which contains a table of privileged handlers for Windows system services similar to the syscall table in Linux. Compromising this data structure would allow installation of malicious handlers to replace the original versions. A similar attack can be made for the Interrupt Descriptor Table (IDT) which registers handlers for exceptions and interrupt routines. Patching the kernel is another method of injecting attacker code so that when a given kernel function is executed, the attacker code is also triggered. Filter drivers establish a chain between several layers of device driver functionality and rootkits can insert themselves between two of the layers to intercept traffic [40].

On x86 platforms, a special CPU operating mode called System Management Mode (SMM) typically handles important runtime platform management tasks including managing CPU power states, controlling low-level hardware such as the CPU fans, handling thermal throttling, performing BIOS flash updates, and handling memory errors, among other tasks [70]. Intel introduced SMM with the 386 SL microprocessor [45]. SMM code operates in a supervisory mode in which the CPU register state and memory are accessible to it. SMM also benefits from hardware protections over its memory region that, when properly configured, prevent other code from viewing or modifying it.

In recent years, some researchers propose a more active role for SMM, incorporating rootkit detection capabilities below the hypervisor and operating systems. SMM's desirable properties such as broad visibility and protected execution present an intriguing approach to better detect host software rootkits. SMM would represent a new layer in the system hierarchy with the ability to perform this detection (Figure 1.2). The SMM-based rootkit detectors, which

we term SMM Runtime Integrity Measurement Mechanisms (SMM-RIMMs), monitor operating system or virtualized environments for rootkits. They accomplish this by preempting execution periodically and inspecting the interrupted state, looking for unexpected changes compared to a previously-gathered baseline. The baseline is established based on initial measurements of presumed static resources which are gathered at a time where the system is considered to be in an uncompromised state.

When rootkits compromise the lowest level of host software, e.g., the kernel or hypervisor code that controls the system, they become very difficult to identify as the detection code is at risk. Security compromises at this level would have significant repercussions as this privileged layer as well as all software layers above it are vulnerable. SMM-RIMMs benefit from operating from within an isolated and hardware-protected SMM memory region (SMRAM). They also are privileged to look into host software's memory and register state which presents a very useful property for a rootkit detector as these resources are typically changed by rootkits. Beyond this, once an SMM-RIMM is triggered via an SMI, all host-side execution is paused for the duration of the inspection which presents the opportunity to interrupt malicious code or detect traces of its past operation. As x86 platforms broadly support SMM, there is no additional hardware required or significant modifications needed for host software to take advantage of an SMM-RIMM. For these reasons, SMM-RIMMs present intriguing possibilities for adding new detection capabilities to combat host software rootkits.

However, there are challenges. The entry into SMM is accomplished by a System Management Interrupt (SMI) which typically takes all CPU threads out of the operating system environment and into the BIOS's SMI handler. This can be a disruptive asynchronous operation from the perspective of any

code executing outside of SMM as it would be unexpectedly preempted for the duration of the SMI. Neither the hypervisor nor the operating system layer is aware of time spent in SMM. The disruption of an SMI is further magnified in a multicore platform, since all of the cores are preempted from the time an SMI occurs to the SMI's completion.

Proponents of SMM-RIMMs have provided limited measurements of performance impacts on applications [7, 113] or provided a brief treatment on system calls [119] but did not extend this analysis further or establish an upper bound on SMM time.

These approaches [7, 119, 113] would spend up to 233x the amount of time in SMM compared to the SMI latency guideline [23]. Our investigations [23] found significant negative impacts of this amount of time spent in SMM including performance degradations, subtle correctness issues, and problematic impacts on device drivers. Improperly scheduled SMM activity has the potential of creating highly perceptible degradation as well as subtle but negative effects.

As the proposals for utilizing SMM for runtime integrity measurement may result in fundamental changes over the utilization of SMM, devising methods of scheduling potentially long security inspections has become necessary. These scheduling methods bring order to what would otherwise add uncertainty over platform performance and correctness.

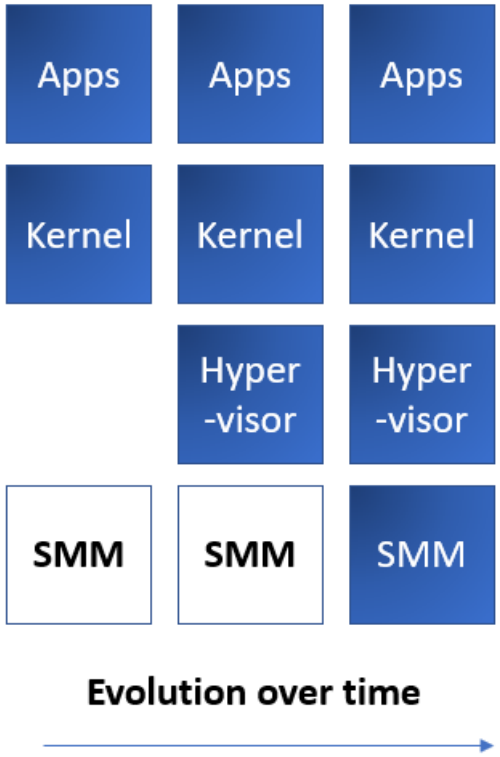

FIGURE 1.2: Usage of System Layers Over Time

A second key limitation in the current SMM-RIMM state of the art is that there are no architected methods for adding new inspections at runtime to SMM-RIMMs. This capability becomes critical as attackers develop new rootkits and the detection mechanism needs to be able to detect these new attacks. Additionally, rootkit malware may seek to hide from RIMMs and a non-extensible set of measurements would not provide adequate detections. The ability to dynamically vary the set of monitored resources helps keep malware unaware about what is being inspected and would raise malware's difficulty in avoiding evasion. Operating systems are also more dynamic today as techniques such as kernel space address layer randomization vary the locations of kernel code sections further challenging static approaches.

Our reformulation of the SMM-RIMM concept requires a new method of

guiding the SMM-based inspections without building in specific operating-system or hypervisor details into SMM. Updating hard-coded values in SMM at runtime is not possible as SMM is flashed onto a hardware chip that requires a system reboot in order to update and supports a finite amount of write cycles. Compounding the challenge of providing a usable SMM-RIMM, no SMM-RIMMs have been publicly released which significantly limits research into the SMM-RIMM concept. Beyond these challenges, the increased potential for SMM as an interesting attack surface also presents challenges for SMM-RIMMs.

To address these limitations, we developed a new approach, EPA-RIMM (Extensible Performance-Aware Runtime Integrity Measurement Mechanism). EPA-RIMM targets these key challenges for SMM-RIMMs. Using the results of our research into these new capabilities, we show that it is possible to implement an SMM-RIMM for servers that is extensible, performance-aware, and effective. We also demonstrate methods of addressing the security of the SMM-RIMM with our architectural design that enforces encrypted and signed communications for our SMM-RIMM and leverages the principle of least privilege.

For the extensibility requirement, we provide a flexible mechanism to dynamically vary the integrity measurements and provide developers with the necessary access to design their own measurements. To ensure that the SMM-RIMM is performance-aware, we provide a method of bounding execution times in SMM. The approach is in sync with the unique performance properties of SMM and decomposes large security checks into shorter tasks that fit within the timing constraint. These tasks can be scheduled independently without severe system performance degradation. The performance foundation for our approach is informed by our extensive characterization of

SMM impacts. The amount of security checking can be increased in times of attack or decreased to reduce system impacts, providing a balance between performance and needs for security inspections. By removing impediments to real-world deployments of SMM-RIMMs, we enable a powerful new tool for identifying the presence of malicious persistent rootkit software in sensitive environments. Providing rootkit detection allows discovery of malicious code that facilitates data leaks and lateral movement in the network.

Section 1.1 provides details on scope of attacks we address. We describe the key challenges for SMM-RIMMs and how we address them in Section 1.2, our contributions to advance the state of the art in SMM-based runtime integrity measurement in Section 1.3, and provide an orientation to the remainder of the document in Section 1.4.

1.1 In-Scope Attacks

We focus on persistent in-memory rootkit and ransomware attacks that compromise presumed-static operating system and hypervisor resources. Rootkits are malicious software that hides their traces and also provides an attacker with the ability to control the system. Rootkits often include one or more of the following techniques: interrupt hooking, changes in low level CPU registers that control paging and permissions (e.g. CR0, CR3, CR4 CPU registers), changes in the operating system kernel or hypervisor code, exception handlers, and other similar resources. While there may be some applicability of our mechanism to detect malicious application code, we consider this out of scope. We also focus on monitoring resources with contents that are accessible at provisioning time. Our priority is to first ensure adequate inspection of the host software resources in an effort to strengthen the lowest levels of the platform. Without a strong foundation, securing the upper layers of software

becomes infeasible.

Ransomware attacks can leverage similar rootkit techniques such as kernel code compromise to gain control and then begin to encrypt the user's data, requiring payment before a decryption key is provided. These attacks are also in-scope as the kernel resources they compromise can also be detected with EPA-RIMM.

1.1.1 Sample Rootkit Techniques

1.1.1.1 IDT Hooking

One technique that allows the attacker to hide traces of their presence on the system is to modify the Interrupt Delivery Table (IDT) mechanism [52]. The IDT is a data structure that links interrupts on the system with the code that handles each of these interrupts when they are triggered. Several attack variants are possible. One attack changes the value of the IDTR which is a CPU register that provides the address of the IDT table. An attacker could set up a fake IDT in memory with malicious code in it and then adjust the IDTR to point to this fake IDT. Once a system interrupt would be triggered, for example, after a page fault, malicious code would automatically be executed in place of the original code. Another possible attack would be to keep the original IDTR but insert malicious code into one of the handlers in the IDT. When this compromise interrupt handler would fire, malicious code would transparently execute. SMM-RIMMs can readily detect these attacks by watching for changes in the IDTR register or hashing the IDT table memory.

1.1.1.2 SMEP Disable

One important CPU-based security feature is Supervisor Mode Execution Protection (SMEP). This prevents supervisor (Ring 0) code from being able to execute instructions from user-mode (Ring 3) pages. The SMEP feature is

enabled by setting bit 20 in the CR4 control register. The Sage ransomware utilized a Windows vulnerability (CVE-2015-0057) to transition from Ring 3 to higher Ring 0 privileges, disable SMEP, and modify the LDT.

1.1.1.3 Kernel Rootkit Code Injection

The Snakso rootkit appeared in 2012 and targeted 64-bit Linux kernels. The WannaCry ransomware leveraged leaked versions of the NSA exploit, EternalBlue, to attack other networked computers with a kernel-code exploit. The Locky ransomware compromised Windows kernels with vulnerabilities to CVE-2015-1701 to execute payload code with kernel privileges. In a write-up of Bad Rabbit, security researchers are fairly confident that this ransomware used the EternalRomance exploit to do arbitrary reads and writes to kernel memory space. These attacks rely upon modification of kernel code which could be accomplished by kernel vulnerabilities such as CVE-2013-2850, CVE-2017-12188, or CVE-2016-8633 [19].

1.1.1.4 System Call Hooking

This attack searches for the location of the system call table in the System.map file, clears the write-protect bit in CR0, modifies the system call table to insert a new attacker-provided function, then re-enables the write-protect bit [29].

1.1.1.5 Xen Venom Rootkit VM Escape

The Venom vulnerability in QEMU's virtual floppy disk controller (FDC) was discovered in 2015 and affected multiple hypervisors including Xen, VirtualBox, KVM. It enables a VmEscape attack in which an attacker could escape the constrained environment of the virtual machine and gain execute permissions on the hypervisor [21].

1.1.1.6 Xen Exception Handler

Examining one in-scope rootkit attack in detail, we refer to a Xen hypervisor compromise from Invisible Things Lab [40]. This attack leverages any available buffer overflow or logic error that resulted in Ring 0 privileges. Once this access was gained, the hypervisor attack begins with the installation of a stealth backdoor by replacing the hypervisor code for one of the privileged interfaces for a virtualized guest to communicate with the hypervisor (Xen hypercall) with malicious code. When the hypercall is triggered, the attacker's code will execute in place of the original code. The attack also alters the debug exception handler to detect and executes code contained in malicious packets instead of handling the debug exceptions. The attack takes advantage of a higher priority for debug exceptions than firewall rules as the code is executed prior to the firewall inspecting the packet [40]. This attack would be difficult to detect from a compromised hypervisor but could be detected by measuring Xen's hypercall handling code and debug registers.

Table 1.1 summarizes these attack techniques.

TABLE 1.1: In-scope Attacks Detectable with EPA-RIMM Checks

Attack	**Example**	**Cmd**	**Operands**
IDT Hooking	Phrack IDT	Reg VM	IDTR IDT
CR4.SMEP Disable	Sage	Reg	CR4
Kernel Code Injection	Snakso, WannaCry, Locky, Bad Rabbit	VM	Kernel Code
System Call Hooking	sys_call_hijack	VM Reg	Kernel RO Data CR0
Xen Code Injection Xen Exception Handler	Venom Xen Exception Hooking	VM	Hypervisor Code

1.2 Key Challenges for SMM-RIMMs

Despite the promise of SMM-RIMMs, five significant challenges remain. In this section, we describe these fundamental challenges (C1-C5) that greatly reduce the effectiveness of SMM-RIMMs.

C1 : SMM-RIMM Security: SMM-RIMMs operate with high privileges. Mechanisms are necessary to ensure that they are not compromised.

C2 : SMM-RIMM/OS Semantic Gap: The SMM-RIMM is not aware of which resources should be inspected and where they reside.

C3 : SMM-RIMM Performance: Existing SMM-RIMMs greatly exceed SMI latency guidelines which would result in significant performance degradations and performance impacts.

C4 : SMM-RIMM Measurement Variability: SMM-RIMMs lack measurement variability. They do not vary the sets of measurements over time or adjust measurement frequency dynamically.

C5 : SMM-RIMM Code Availability: No SMM-RIMM implementations have been published which significantly limits researcher access to this technology.

1.2.1 C1 - SMM-RIMM Security

Research Question 1: How to design a more secure measurement agent?

While SMM has promise for host software rootkit detection [7, 113, 119], other researchers have raised concerns over the privilege granted to SMM by highlighting security issues [99, 53, 78]. Security researchers have leveraged SMM vulnerabilities to implement SMM rootkits and other compromises [59, 15, 67,

53, 14, 20]. Some compromises have resulted from improper platform configurations. Others resulted from SMM coding errors. One promising path for reconciling the need for a privileged SMM layer for security monitoring with the concern of the broad reach of SMM is shown by STM PE [84] (independent concurrent work) and EPA-RIMM-V [110] (concurrent work within the EPA-RIMM research group). These approaches use the SMI Transfer Monitor (STM), an SMM-based hypervisor that virtualizes SMI handler code, to apply a protection policy over the handler execution. This policy prevents arbitrary accesses to system resources, instead confining an SMM-based measurement agent to a permitted set of resources. The STM's protection capabilities go significantly beyond a related UEFI effort that implements SMM page table isolation to constrain SMI handlers accesses to host memory [117], by allowing restrictions over other resource types such as CPU Model-Specific Registers (MSRs), IO Ports, among others. The power of SMM also has the potential to be misused. Therefore EPA-RIMM can be used with the STM to limit the access of the measurement agent to a minimal set of resources [110, 84].

Securing the EPA-RIMM measurement agent presents special challenges as there is a need to balance limited amount of available time in SMM and the overheads of the STM as well as security features such as encryption, checking message authenticity, and signature checks. The STM helps solve one security issue, however, others remain. Attackers could attempt to mount several types of compromises on the EPA-RIMM measurement agent as enumerated below.

1. **Attacks on measurement agent communications**: There are several key attacks on EPA-RIMM's measurement agent possible:

   (a) *Communications spoofing*: Attackers may try to construct measurement requests to send to the measurement agent. This could cause

it to spend time monitoring unimportant resources instead of resources that could indicate signs of an attack. This attack could also, optionally, manifest as a denial of service attack as described below.

(b) *Communications tampering*: Attackers may try to intercept and modify messages while in transit. Examples of these tampered messages would be changing the memory address to be measured or another measurement guidance parameter to cause it to differ from the intentions. The goal of such an attack would be to divert the measurement agent's attention from the intended measurement target. Similarly, the attack would also be possible on the return of results. The attacker could attempt to replace an alert with a message that indicated that there was no detected issue. A successful attack of this nature would obscure an attacker's compromise of a monitored resource, neutralizing EPA-RIMM's detection capabilities.

(c) *Denial of service*: Attackers may attempt to cause a denial of service by overwhelming the measurement agent with requests causing the system to spend too long in SMM, preventing other meaningful work. There are two denial of service scenarios:

   i. An attacker is able to trigger measurement SMIs but is only able to provide spoofed Bins.

   ii. An attacker succeeds in compromising the signing and encryption keys.

   For the first scenario, processing time for each of these Bins will be minimal. Once the measurement agent identifies a problem with

Bin correctness (e.g. problems decrypting the Bin or with HMAC integrity), it ceases processing of the Bin. For the second scenario, the attacker would specify very large hash operations to be performed to cause extensive time to spent in SMM. This attack attempts to exploit a design consideration for EPA-RIMM, namely that the primary method of decomposing large measurement operations is done outside of SMM.

(d) *Replay attacks*: This attack "replays" a previous measurement result by first intercepting a valid measurement and saving a copy of it. The attacker may block future measurements but pass off previously-collected measurements in their stead.

(e) *Breaking measurement and result confidentiality*: An attacker may try to observe the measurement requests and returned results while they are in transit. This would allow the attacker to determine what resources were being measured and whether the attack was detected. This information could aid an attacker in determining which system resources to compromise and also gauge the stealthiness of their attack.

2. **Use of EPA-RIMM as a side channel**: Since the measurement agent has deep visibility into the running operating system or hypervisor, attackers could attempt to exploit this visibility into leaking information about the running system. For example, attackers may wish to view the contents of memory or register values to aid in later compromising the system.

1.2.2 C2 - SMM and OS Semantic Gap

Research Question 2: How can an SMM-based measurement agent comprehend the host-side software layout without hard-coded layout information?

An additional challenge results from a semantic gap between SMM and the host software due to their disjoint operating environments. SMM does not have a native understanding of the host software layout or what should be measured. Building in host-software specific information to SMM as done in SPECTRE [119] is not a feasible solution as it is brittle. Host software layouts can change over time, updating SMM code at runtime is not possible, and kernel address space layer randomization thwarts the scheme. Thus, resolving this challenge requires a flexible mechanism to specify the set of resources to be measured as well as where these resources reside.

1.2.3 C3 - SMM-RIMM Performance

Research Question 3: How can the performance impact of SMIs be measured and analyzed?

Research Question 4: How can an SMM-RIMM be designed to minimize time spent in a single session to meet SMI latency guidelines?

Research Question 5: What performance optimizations can reduce the overall overhead of integrity measurements?

Performance is a major challenge for SMM-RIMMs. All SMM-RIMMs that we are currently aware of exceed the published SMM guidelines by orders of magnitude [7, 113, 119, 84]. As SMIs preempt all CPU threads, this can result in significant performance degradations and correctness issues if too much time is spent in a single SMM session. Our research demonstrated that major

system impacts can occur such as significant perturbation of the kernel and device drivers as well as clear performance impacts such as severe application slowdowns [23]. Interrupt handling is delayed until the CPUs return from SMM and latency-sensitive applications can be affected. System software assumptions regarding scheduling regularity as well as task durations are challenged by prolonged periods of time in SMM. SMM's strong isolation from host software also results in a lack of overall scheduling mechanism between the two environments. While process scheduling is a key operating system feature, the operating system's scheduling mechanism is not aware of SMM and there are no established methods of efficiently scheduling security measurement events that span these two contexts. The operating system has no mechanism to preempt an SMI during its execution. It is also not aware that time was spent in SMM which impacts the accuracy of scheduling and process time accounting. Additionally, CPU intensive workloads have no mechanism to avoid the throughput loss as time has passed but no workload computations were performed.

1.2.4 C4 - Measurement Variability

> *Research Question 6: How can variable integrity measurements be supported in SMM?*

The fourth key challenge pertains to measurement variability both for measurement type as well as frequency and durations of checking. Currently-proposed SMM-RIMMs utilize pre-configured inspections that do not dynamically alter the set of monitored resources, measurement frequency, or duration. one approach hard-codes particular kernel addresses to measure [119].

A key complication for variable integrity measurements is that the SMM code is infeasible to modify at system runtime since they reside within a protected memory range and updating SMM code would require re-programming the BIOS flash and a system reboot, thus incurring downtime. A lack of variability provides a significant complication for an effective SMM-RIMM for three key reasons: 1. Kernel address space layer randomization (KASLR) varies the addresses of kernel functions upon boot. Thus, statically hard-coding addresses in SMM of particular kernel functions is infeasible in modern systems. 2. Continued malware evolution and response to SMM-RIMMs implies that an unchanging set of measurements will not remain effective indefinitely. Compounding this issue, a simplistic scheduling mechanism would cause them to be increasingly vulnerable to the "scrubbing attack" [81, 112] that compromises the system but then cleans up traces of the attack before an inspection would occur. An example of this would be a rootkit that loads malicious code into a virtual memory page but then unmaps the page prior to an integrity measurement. If malware were able to derive the inspection times, it could readily remove its traces before the next inspection, avoiding detection. 3. New operating system and hypervisor software updates may change internal layouts which could invalidate previous assumptions over locations and structure of kernel objects and functions.

1.2.5 C5 - SMM-RIMM Code Availability

The final key challenge is that no SMM-RIMM implementations have been published. Previous approaches were closed-source and researchers could not examine, evaluate, or extend them. This limits the feasibility of the entire approach, preventing adoption of this protective mechanism.

1.3 Contributions

EPA-RIMM constitutes an effective inspection capability that targets stealthy host software rootkits. With an extensible, performance aware, and effective SMM-RIMM, rootkit developers can not count on a lack of detection. Providing a usable scheduling mechanism for measurement SMIs also presents a method to bring order to the scheduling of an important class of platform management tasks. As we advanced the SMM-RIMM concept, there were some established system techniques that we were able to draw from, such as the knapsack problem and real-time operating system schedulers, however, other aspects required original thinking. As the prevalent approach of unbounded time in SMM for rootkit detection was infeasible, we researched and identified methods of decomposing large measurements into smaller portions to fit within an SMI time quantum. We demonstrate the merits of measurement decomposition, priority-based scheduling, and aging to prevent measurement task starvation. As entries and exits from SMM consume time that would otherwise be used for processing, there is a risk of the overheads from transitioning into and out of SMM becoming the dominating cost. Therefore, we maximize the amount of work spent in an SMM session up to the specified limit. With this approach, negative system impacts due to prolonged periods of SMM execution can be avoided and effective rootkit detection performed with minimal impact. We further optimize EPA-RIMM's performance by implementing an SMM-RIMM variant of Paradyn's performance hypothesis, instead focused on identifying rootkits.

1.3.1 First Linkage of SMI Latency Guidelines and Performance Impacts to SMM-RIMMs

At the outset of this work, the state of the art for SMM-based runtime integrity measurement mechanisms consisted of: 1. SMI durations that greatly

exceeded the Intel BIOS Test Suite (BITS) SMI latency guideline of 150μs, and 2: Unbounded SMI time. This Intel-designed tool generates alerts if detected SMI durations exceeded 150μs, however, SMM-RIMM developers did not limit SMI time in their design and the consequences of exceeding the threshold were not clearly demonstrated. In our paper, "Performance Implications of System Management Mode", we tied the SMI latency guideline (**LimitSMI$_{\mathbf{BITS}}$**) for the first time to SMM-RIMMs and based on detailed measurements, demonstrated a wide-range of negative impacts when the guideline was exceeded which included kernel correctness issues, performance degradations, and increased power usage [23].

Our performance analysis impacted the design of an HP Labs and Centrale-Supelac SMM attack detection mechanism as they targeted their mechanism to support the 150μs SMI latency guideline that we proposed adherence to, noting "The Intel BIOS Test Suite (BITS) defined the acceptable latency of an SMI to 150μs. Delgado and Karavanic showed that, if the latency exceeds this threshold, it causes a degradation of performance (I/O throughput or CPU time) or user experience (e.g., severe drop in frame rates in game engines) [18]." Our SMI performance characterization also provided insights for researchers from UCLA and Microsoft who referred to our performance analysis of SMM [23], noting that "Delgado et al. . . . were the first to experimentally expose the performance implications of Intel's System Management Mode (SMM), which is often used for memory error reporting (and which we discuss in this work). They observed inconsistent Linux kernel performance and reduced quality-of-service (QOS) from SMM on latency-sensitive user applications [35]."

These citations reflect the new awareness of SMM performance impacts that our measurements and analysis has contributed. The design of the HP

Labs and CentraleSupelac SMM attack detection which adhered to the 150μs SMI latency guideline demonstrates the impact of our SMM performance measurement methodology and our linkage of the SMI latency guideline to SMM-RIMM performance.

1.3.2 First performance-aware SMM-RIMM design incorporating measurement decomposition

EPA-RIMM's ability to flexibly schedule integrity measurements with sensitivity to the current threat levels provides the ability to dynamically increase the amount of security inspection during times where systems in an enterprise are experiencing heightened attack activity. This allows system impacts to be tuned to acceptable tolerances in the general case as well as providing a new ability to increase the amount of inspections when needed. Our method demonstrates that it is possible to take longer-running measurements and decompose them into smaller components that can be scheduled in accordance with SMI latency bounds. To allow us to evaluate differing approaches in RIMM scheduling, we created a scheduler simulator that allows the evaluation of changing key scheduler parameters to investigate their impacts.

Our SMI latency system impact measurements establish guardrails that help keep maximum SMM-RIMM preemptions at desired levels while also allowing for additional headroom for enhanced detection when it is needed. To accomplish this performance analysis, we created an SMM performance measurement methodology and utilized it to conduct a detailed performance characterization of SMM. Our analysis was the first in-depth study of the impacts of SMM on hypervisors, operating systems, device drivers, and applications. This performance analysis provided the necessary empirical results to allow us to re-design SMM-RIMM scheduling and demonstrate a mechanism of scheduling platform tasks in an orderly manner. Loutfi, I.,

notes that EPA-RIMM is a "novel" way of using SMM for non-traditional purposes [69].

1.3.3 First application of measurement triggers to SMM-RIMM

SMM-RIMMs have traditionally featured measurements that were timer-based. EPA-RIMM's approach allows for reducing the amount of measurements required to evaluate hypothesis regarding the system state. This approach leverages measurement triggers that schedule less-intensive measurements first to determine if more intensive measurements need to be run to further evaluate the hypothesis. EPA-RIMM supports this capability using the Diagnosis Manager and flexible Check descriptions. This new capability is a first for SMM-RIMMs and can significantly reduce the amount of SMM measurement time required to evaluate a hypothesis.

1.3.4 First SMM-RIMM Benchmark: EPA-RIMM Bench

Runtime security inspections have the essential property that performance efficiency is a key concern. With a virtually unlimited set of resources to measure and re-measure over time, the amount of security inspections that can be performed without degrading the user experience beyond acceptable tolerances requires quantification. As processor performance and degree of parallelism increases, the achievable measurement increases resulting in a reduced time to discover attacks. Additionally, EPA-RIMM could support a variety of hashing, encryption, and message authentication code algorithms, each of which has their own performance and security characteristics. EPA-RIMM Bench provides the ability to directly quantify achievable SMM integrity measurement performance which would allow careful performance analysis to support important design and implementation decisions.

1.3.5 First Publicly-Available SMM-RIMM Prototype

Before our work, there have been no public release of an SMM-RIMM. We constructed a functional EPA-RIMM prototype that allows research into SMM-based runtime integrity measurement and accompanying performance measurements. With this prototype, we have demonstrated its ability to detect rootkit attacks and also quantified the impact of the RIMM's fundamental operations of register accesses and memory hashes. We provide our prototype to allow researchers to build upon the framework. Taken together these improvements remove key limitations that reduce the practicality and effectiveness of SMM-RIMMs and show that the approach can be an implementable and useful mechanism for detecting host software rootkits. The ideal outcome of this work would be the availability of the EPA-RIMM framework that provides this new capability to aid in the detection of rootkits and a community of security researchers who would develop and share checks for the architecture.

1.4 Document Organization

The remainder of this document is organized as follows: Chapter 2 provides background on the current security threat landscape, treatment of varied approaches for securing computer systems and data, describes the urgent need for runtime measurements, identifies EPA-RIMM's scope, and provides details about the system impact of SMM. Chapter 3 examines related work covering a variety of approaches in performing runtime integrity measurement including software, discrete hardware devices, and firmware among other approaches, allowing an understanding of how EPA-RIMM compares to existing techniques. Chapter 4 describes our SMM performance measurement methodology that we used to perform performance sensitivity testing of

varying degrees of SMIs which provides the performance underpinnings of EPA-RIMM. In Chapter 5, we provide a detailed performance study into the system and application effects of varying degrees of SMM activity, as would be incurred with EPA-RIMM. Chapter 6 presents the design requirements of EPA-RIMM including two new requirements we propose. Chapter 7 presents the EPA-RIMM architecture. Chapter 8 describes our EPA-RIMM prototype including the design, examples of rootkit detection, and performance measurements. In Chapter 9, we describe our SMM-RIMM scheduler simulator which provides the ability to simulate the results of a variety of important scheduler-related parameters that would improve the performance efficiency of EPA-RIMM. Chapter 10 describes RIMM-Bench which provides a mechanism for comparing SMM-RIMM performance between systems. Chapter 11 summarizes our research.

2
Background

The attention to the need for runtime integrity measurement is driven by current trends in which rootkits remain undetected for prolonged periods of time. We discuss the contemporary threat landscape in Section 2.1, various approaches to preventing or detecting malicious activity in Section 2.2, describe the urgent need for runtime checking in Section 2.3, and show performance limitations of current approaches in Section 2.4. We describe information on measurement triggers which fine-tune the of the approach in Section 2.5.

2.1 Threat Landscape

In recent years, the potential damage for system compromises has grown significantly. Attacks have taken on financial and geo-political angles. Executive assistant director of the FBI's Criminal Cyber Response and Services Branch, Robert Anderson, noted that "We're in a day when a person can commit about 15,000 bank robberies sitting in their basement [54]." Cyber-espionage is also increasing with the majority of the attacks by state-affiliated attackers (87%) and organized crime (11%). Of the attack methods employed by cyber-espionage actors, 37% exploited software vulnerabilities and 24% leveraged rootkits. Compounding these issues, a recent study by Verizon noted that only around 20% of attacks in organizations were detected internally with the majority of attacks detected by law enforcement and third parties [109]. This low percentage of attack detection within an organization demonstrates that timely detection of attacks continues to be an issue. Efforts to utilize external telemetry information from third parties could also bring targeted expertise to organizations that are less prepared to detect attacks.

The retail sector in 2014 also experienced large-scale attacks with both Target and Home Depot suffering broad compromises in their point of sale systems. A survey by the Ponemon Institute showed that "the average cost of cyber crime for U.S. retail stores more than doubled from 2013 to an annual average of $8.6 million per company in 2014. The annual average cost per company of successful cyber attacks increased to $20.8 million in financial services, $14.5 million in the technology sector, and $12.7 million in communications industries [42]." The study also notes that "cyber attacks can get costly if not resolved quickly" and the average time to resolve a cyber attack was 45 days. An examination of today's threat landscape shows that there are large financial consequences to compromises and quick detection and remediation become determining key factors in the overall cost of the attack.

Attacks may arrive by a number of means. These can include: malicious code that targets the application layer, denial of service, web-based attacks, phishing and social engineering, malicious insiders, stolen devices, malicious code that has entered a network, viruses, worms, and trojans that reside on endpoint systems, and botnets [42]. Different attacks require different detection mechanisms. Application-level attacks are very prevalent as these are easier targets while lower-level attacks can have a broader impact on the system and are harder to remediate.

2.2 Varied Approaches for Securing Systems

Efforts to secure computer systems and data fall into a number of categories. One key approach focuses on improving the design of computer systems to strengthen security in one aspect. For example, the Address Space Layout Randomization (ASLR) feature helps prevent malicious code from being able

to rely upon a consistent user-space memory layout across a number of machines by randomizing the locations of the "stack, mmap region, heap, and the program text itself" [27]. This feature is also present in the Linux kernel to randomize where the kernel's code is placed at run-time. Kernel ASLR (KASLR) could complicate the effort of certain runtime integrity approaches that rely upon kernel functions being located at certain known addresses. Secure Boot protects against an operating system being compromised early in the boot process from a malicious boot loader [114]. Microsoft has added integrity checks over the Windows heap and added Data Execution Prevention (DEP) that prevents certain programs and services from executing from memory regions set aside for Windows [79].

Formal Verification is a technique of logically checking whether a design meets a set of given requirements, under a set of assumptions [56, 16, 111]. One of the most prominent approaches in this area is the seL4 microkernel that has undergone a complete functional correctness proof for its code [56]. While Formal Verification presents the ability to make strong claims over security properties, the approach has scalability limitations due to limits in the amount of code that can be reasonably formally verified. It also requires making assumptions over system design and configurations that may not broadly hold. For example, the formal verification of the seL4 microkernel assumes that the widely-used Direct Memory Access (DMA) feature is not being used, leaving users to formally verify their own DMA-capable device drivers. Additionally, SMM presents serious complications to formal verification on x86 platforms as it operates completely out of band of the formally verified environment and has the privilege of modifying arbitrary resources. Azab, et al. note that the impacts of SMM and firmware "could negate all seL4 guarantees" [6].

Another approach attempts to construct strongly isolated computation environments on systems that may be less trusted [75, 6]. For example, Flicker can provide strong isolation of running code even when the "BIOS, OS, and DMA-enabled devices are all malicious" [75]. SICE leverages SMM protections to protect running code from other code running on the system. Additionally, when SICE-protected workloads are not running, their data is stored in a hardware-protected SMRAM region. One new method of providing isolated execution is Intel's Software Guard Extensions (SGX) which leverages new CPU instructions to construct protected environments for user code to execute in. These protected applications ("enclaves") have hardware-based protection against access from malicious software running at higher privilege levels such as hypervisors, BIOS, or operating systems. SGX also protects the confidentiality and integrity for the protected application's code and data [76, 39].

A separate approach acknowledges that despite the efforts to construct host software with improved security, computer systems will likely be compromised and that vigilant observation is necessary. This runtime integrity monitoring approach mitigates the risk of successful compromises by periodically checking the status of key resources in host software to determine if they have been unexpectedly changed. This leverages the observation that certain kernel data structures and code regions are generally static once configured [50]. Runtime integrity monitors can quickly alert administrators to alert them that a potential attack is in progress [7, 90, 119, 113].

2.3 Urgent Need for Runtime Checking

Currently, servers that host sensitive data for thousands of users typically run without any checking of the low-level fundamental resources that control the

system. Unlike antivirus programs which regularly scan user programs and data, there are often no checks over a number of important security-sensitive hypervisor and low-level operating system resources. Once attackers are able to gain a foothold, they can often persist for extended periods of time undetected. For example, the security analysis of the highly publicized attack of Sony in 2014 suggests that attackers remained in Sony's network for months before unleashing a devastating ten minute attack [1].

Despite improvements in operating system and hypervisor security, rootkits continue to compromise crucial host software environments. Rootkits provide attackers with the ability to hide all traces of their activity on the system from host software, allowing for stealthy operation. Host software rootkits compromise sensitive software and hardware resources that control fundamental operations such as interrupt handling, memory access, and event handlers. If a system is compromised at the lowest level, all code running above it is at risk. Quick runtime detection of attacks becomes increasingly critical to alert administrators so that they can prevent attacks from spreading or information leaking. Currently there is little host software runtime integrity measurement software actually deployed and the topic is largely limited to the research arena.

SMM-based runtime security measurement mechanisms can reside outside of potentially compromised host software regions and provide ongoing host software rootkit detection. However, the proposed runtime integrity measurement mechanisms of today are not likely to be deployed. The current simplistic scheduling of security checks, inflexible specification mechanism for new checks, fairly predictable checking, lack of telemetry information, and performance impacts reduce the attractiveness of the potential solution. With a number of host software resources to check, the performance impacts

of the design become crucial. The authors of one runtime security checker note that they could trigger a security check on "every CPU instruction by using performance counters in the CPU, thus guaranteeing that every state is introspected [119]." Could this improve security? Probably so, if the mechanism were sound. However, no user would accept this system due to severe performance degradation, regardless of the additional security it provided.

Tolerances for security delays fall upon a spectrum. Some environments such as stock trading are very sensitive to latency. A TABB Group study "estimated that if a broker's electronic trading platform is 5 milliseconds behind the competition, it could lose at least 1% of its flow. That equated to $4 million in revenues per millisecond. Up to 10 milliseconds of latency could result in 10% drop in revenues. Today, latency is often measured in microseconds (μs), with the current impact per μs commonly accepted by many traders to be $100,000 per year [32]." Similarly, a Blackhat security presentation explained why firewalls, routers with access permission lists, and intrusion detection systems do not exist in stock trading environments: "In the vast majority of interconnection scenarios, a few milliseconds is not that much of a problem. In the case of low latency trading, it's about 100,000 times too slow [104]."

However, not all environments are this sensitive to latency. The requirements for credit card processors (PCI DSS) prescribe the use of a firewall, encryption of the transmission of cardholder data across public networks, the use of antivirus programs, and monitoring of access to network resources and cardholder data. Each of these requirements can improve the security, if properly implemented, however would come at the cost of system performance [3]. With a varied set of industry requirements and tolerances for security delays, enterprises would benefit from the ability to dynamically vary the amount

of security checking performed on their systems. Scheduling fewer security checks may improve performance but could result in a less-secure system. Likewise, performing extensive checks may uncover well-hidden malicious code but could degrade the user experience significantly. An effective runtime security checking mechanism should be designed to operate within the target environment. To provide strong coverage, environments utilizing runtime integrity measurement mechanisms should support increasing the amount of checks as needed to respond to rapidly spreading attacks.

Current research on RIMMs is largely limited to the security of the mechanism itself and a canned set of attacks that can be detected. For these mechanisms to become practically implementable in real computing environments, the mechanisms must be generalized to be made adaptable to newly emerging threats and feature performance tuning knobs that encourage the server administrator to not disable the mechanism and its protections.

2.4 System Impact of SMM

The key challenge with SMM-RIMMs is that their system impact may be significantly disruptive to other processing on the system, depending on how it is scheduled. When an SMI occurs, all CPU threads transition into SMM, saving their interrupted state in an SMM memory region called the SMRAM Save State Map. When SMM processing has completed, the interrupted context of the CPU threads is restored and the threads return back to where they were executing when the SMI was received. Thus, from the perspective of the code, time has passed but the code was transparently interrupted. In more recent Intel processors, an SMI counter will increment upon each SMI but other than this mechanism, there is not a direct method of determining that the SMI occurred.

Given these unexpected preemptions, the time spent in SMM should be short in order to not unduly preempt executing code for too long which could result in performance degradations or correctness issues. Intel has released a tool called the "Intel BIOS Implementation Test Suite" (BITS) [106] that counts and measures SMIs occurring on a system and checks that their latencies are within "acceptable" limits (currently defined as 150μs). This rule of thumb has been the only available guideline for latency tolerance. At present, none of the SMM-RIMM approaches that we are aware of has limited their time spent in SMM according to SMI latency guidelines. Each of the SMM-RIMM approaches dramatically exceed current SMI latency guidelines. These mechanisms take between 27 and 267 times the current SMI latency guidelines which presents concerns over their system impact. Table 2.1 shows the stated time requirements for several prominent SMM-RIMMs.

Our previous work in understanding system sensitivities to prolonged SMI delays shows a variety of impacts that result when SMI latencies are increased well beyond the guidelines. The range of effects includes correctness issues in process time accounting, jitter in interrupt processing, loss of timer ticks, and significant performance degradations [23]. Balancing the tension between SMM-RIMMs and other processing is a key focus of this work as we investigate methods of enabling SMM-RIMMs without drastic system impacts. We present our measurement results on the impact of varying degrees of SMIs in Chapter 5.

TABLE 2.1: SMI Time Comparison of SMM-based RIMM approaches

Approach	**SMI *ms***	**Frequency**
HyperCheck	40	1 per sec
HyperSentry	35	1 per 8 | 16 sec
SPECTRE	5 to 32	16 per sec to 1 per sec
Delgado et al	1.5	Not specified
EPA-RIMM-Linux	0.26+	Dynamic
EPA-RIMM-Xen	0.28+	Dynamic
Intel BIOSBits	**0.15**	**Not specified**

2.5 Triggers

The Paradyn automated performance analysis tool implements a W3 search model (Why, Where, When) that seeks to answer why an application is performing poorly, where the performance problem resides, and when the problem occurs [80]. To reduce the performance overhead of this instrumentation, Paradyn does not apply all available performance instrumentation at once. Instead, it explores several hypothesis candidates to determine why a performance problem occurs. It selectively refines the hypothesis by means of Boolean functions. When one light-weight test indicates a potential performance problem in a particular module, a heavier-weight test can be triggered. By only invoking the heavier-weight test if the light-weight test indicated a problem, Paradyn effectively explores the performance problem search space with minimized system impact.

In Paradyn, transient performance overheads do not merit analysis as they consist of a small fraction of total execution time which is not worth tuning. However, SMM-RIMMs should detect attacks as soon as possible after they occur to limit potential damage. Short-lived attacks that hide their traces can possibly evade detection. Thus, this constitutes one key difference between Paradyn's approach and the requirements of SMM-RIMMs. However, by

relaxing this constraint, the trigger mechanism presents a potential method to reduce the overall overhead of SMM-RIMM measurements.

3
Related Work

This section describes related work that most closely relates to our focus of providing mechanisms to determine when an operating system or hypervisor has signs of an attack. With this knowledge, a system administrator could take various actions to reduce the ability of malware to spread further within the environment. A key difference in these varied approaches is the location of the measurement agent. Pure software-based approaches can more easily be deployed than mechanisms that require special hardware or custom firmware, however, attaining effective protection for the software-based measurement mechanism is challenging due to running the mechanism in the same context as potentially malicious code. Hardware-based mechanisms can achieve a greater isolation from malicious code due to lower-level hardware protections over their agent, however, due to their isolated contexts, gaining full access to the necessary CPU and memory state can be challenging. Firmware presents intriguing possibilities due to its leveraging of hardware-based protections and also ability to gain access to a wide amount of system context. Open challenges with this approach are how to provide the ability to properly understand the host software environment and how to resolve potentially significant performance issues.

3.1 Race to the Bottom

Researchers have proposed a number of runtime integrity measurement modules. In the years 2000 and 2001, two kernel module-based rootkit detectors called rkscan [4] and St. Michael were released [101]. Kernel-based approaches benefit from native visibility into the kernel. However, there is no protection from potentially malicious code operating in the kernel. If the

kernel were to be compromised, a kernel-based security checker is also clearly at risk due to running in the same context.

A more privileged layer became necessary in order to check the kernel without being put at risk from other code running at that privilege level. Hypervisors [102, 5, 73] and hardware [100, 65] provided a means to get deeper observation capabilities. System firmware approaches leveraged the powerful, widely available, but lesser-known System Management Mode on x86 CPUs [7, 113, 119]. The choice of placement of the runtime monitor presented a number of trade-offs, namely the degree of protection of the runtime monitor itself, degree of visibility into host state, performance impacts, extra cost required for additional hardware, and ease of monitor updates. Attackers are motivated to entrench their mechanisms deeper into the platform, for example, into the hypervisor itself [31]. This correspondingly drives the need for security checkers to obtain visibility from an even more privileged location.

3.2 Software-based Approaches

One current example of a software module that performs runtime integrity for the Windows operating system is Microsoft PatchGuard. This mechanism was introduced in Windows Vista [77] and monitors operating system modules, the System Services Dispatch Table (SSDT), the Global Descriptor Table (GDT), and the Interrupt Descriptor Table (IDT). However, as a pure software mechanism, it is hampered by operating at the same privilege level as the code that it is trying to monitor. This requires PatchGuard to modify the operating system to protect itself against various attacks. One example is the setting up of a new IDT to avoid a debug exception-based attack. PatchGuard also employs a variety of advanced coding techniques such as "self-modified code,

code obfuscation, self-integrity checking, and randomization" [82]. These types of operating system modifications and advanced coding techniques add complexity to host software and also work against the efforts of lower-level runtime integrity checkers designed to identify unexpected changes in the IDT. Providing a runtime integrity measurement solution that operates outside of the operating system avoids the additional complexity of advanced coding techniques and modifying the very resources the runtime integrity monitor is designed to watch.

3.3 Hardware-based Approaches

Hardware devices generally improve the isolation of the measurement mechanism from the monitored code making them harder for an attacker to tamper with. Additionally, they can offload security measurements from the host CPU which could be a way to improve performance. The main challenges with discrete hardware devices are that they do not have access to important CPU registers that control the platform. Some discrete hardware devices may also have their memory accesses to host memory blocked by an IOMMU mechanism such as Intel VT-D. Besides these drawbacks, there is a financial cost for another hardware component as well as a lack of generality since the mechanism only works when the specialized hardware are present.

3.3.1 Discrete Devices

Researchers have proposed RIMMs rooted in devices on the PCI bus [90], memory modules (MGUARD) [65], SOC [81], and the chipset [13]. The PCI device approach, Copilot, utilized a PCI card with DMA (Direct Memory Access) access to inspect the host's memory in order to locate rootkits. While this approach offloaded the inspection from the host CPU, it was unable to access CPU register state causing its visibility to be limited. Today, IOMMUs (such

as Intel's VT-D) are prevalent on modern x86 CPUs which can shield host memory from PCI devices (including Copilot) which renders this approach less useful. Copilot also relies upon a System.map file built when the kernel was originally compiled to locate kernel functions in memory.

MGUARD has the unique capability to detect SMM rootkits due to its integration into the memory module itself. MGUARD receives direction on what memory pages to monitor via a serial link. This allows customization for different operating systems. MGUARD monitors activity on the memory module itself to determine whether accesses fall into regions that it was directed to monitor. Upon identifying accesses in the monitored area, MGUARD saves copies of these pages to a private DRAM for later analysis. Key benefits of this approach are the strong isolation from host software, continuous monitoring (as opposed to snapshot-based approaches common among SMM-RIMMs), and negligible performance overheads. The drawbacks include a lack of visibility into CPU registers which hampers visibility into the current state of the system. Additionally, new memory module hardware would be required for each system to use this mechanism increasing the financial cost significantly.

The SOC approach, Vigilare, works by snooping memory bus traffic on the system. One clear benefit from this approach is its ability to catch transient attacks which compromise the system, but then hide traces of their presence in order to avoid detection. However, as Vigilare lacks access to the CPU registers, it is vulnerable to relocation attacks in which inspected code is moved to a new location out of the view of the measurement agent. Without CPU register access, Vigilare must also rely upon the System.map file that is generated upon compilation of the Linux kernel to locate the address in memory of the items that it should measure. Additionally, the performance of the SOC may not be sufficient to keep up with high bandwidth memory

busses.

DeepWatch can detect and remediate low-level virtualization-based rootkits by using chipset hardware out of band of the host CPU, for example, Intel Active Management Technology (AMT). DeepWatch uniquely enables the removal of rootkits while other approaches focus on detection. This approach utilizes DMA to access system memory from an embedded CPU in the chipset. The benefits include offloading inspections from the host CPU which would reduce performance overheads, can scan host memory and the SSD while the system is in a sleep state, and can identify SMM rootkits. A key drawback is the necessity for chipset-specific implementations due to different hardware. Nighthawk [120] is an updated approach that uses DMA from the Intel Manageability Engine (ME) to detect rootkits in the operating system, hypervisor, or in SMRAM. However, the ME lacks SMM's ability to have direct access to the CPU register state.

A recent study shows that these discrete hardware devices are vulnerable to the Address Translation Redirection Attack (ATRA) [50]. Note: Nighthawk has special handling for this attack by leveraging SMM's ability to observe the page table address. In one implementation of this attack, an attacker makes a copy of page table data structures and configures the system to refer to this non-legitimate version. External hardware-based monitors that lack access to the CPU registers are typically unable to detect that this switch has happened and continue to monitor the old location. However, at this point, the attacker is in control of the host software memory and the runtime integrity code is unable to detect the attack. This approach to runtime integrity measurement would need to improve its resilience to the ATRA attack in order to remain a viable approach.

3.3.2 CPU Virtualization

Hardware-based CPU virtualization can also aid rootkit detection. Given the wide availability of this CPU feature, there is no additional hardware to purchase in contrast to the dedicated hardware approaches. However, the performance cost of virtualizing the host software becomes a key factor. One prominent example is McAfee Deep Defender which was introduced in 2011 [62]. Deep Defender utilizes a security hypervisor to monitor several key Windows resources in a virtualized Windows guest. These resources include the IDT, SSDT, DKOM list, kernel code sections, certain device drivers, among others. When a change is attempted in one of the guest's monitored resources, the CPU's hardware virtualization triggers a transfer of control ("VMEXIT") from the virtualized Windows guest environment to the security hypervisor for remediation. This approach benefits from strong isolation between the monitored environment and the security hypervisor and can also provide quick detection of improper accesses as they occur. However, DeepWatch requires placing a security monitoring hypervisor below the user's operating system which comes at a significant performance cost over a variety of operations beyond the security inspection required [74]. It would also require nested virtualization to secure a hypervisor which can incur a heavy performance impact.

Hypervisors have also been used to build more secure operating environments as opposed to being used for rootkit detection as in the case of Deep Defender. Security Visor [102] places a small hypervisor below the operating system to prevent unauthorized kernel code from executing. A key mechanism of Security Visor is its virtualization of the operating system's memory in order to intercept potentially malicious changes. The mechanism is rooted in the hypervisor as this has isolation from the kernel and would

not be compromised even upon a successful kernel attack. This mechanism is best suited for operating systems as placing a hypervisor under another hypervisor can introduce an additional performance impact onto the system. The performance overheads of Security Visor can be significant. Linux kernel builds and unzipping the kernel source took 219% and 140% longer, respectively, than when performed on a native Linux installation. The PostMark mail server benchmark showed an 86% performance decrease compared to native Linux. The authors note that application performance will be impacted based on the frequency of kernel calls and the amount of change in the application's working set. These overheads are driven by overheads in shadowing the CPU's GDT, LDT, and IDT registers along with the shadow page table. Advances in virtualization performance such as Enhanced Page Tables may have beneficial impacts on these overheads.

3.3.3 CPU Performance Counters

One of the challenges of identifying rootkits and other malware is that their authors can easily create a large number of variants that accomplish the same results but with variations in the implementation to avoid detection. Traditional software checkers that look for signatures to identify malicious code can be fooled by code that is implemented slightly differently. For this reason, researchers are investigating new methods that do not rely upon static software signatures but incorporate behavioral aspects. One promising approach that incorporates behavioral aspects is work done by Demme, Maycock et al. which leverages CPU performance counters to detect anomalous rootkit behavior [25]. CPU performance counters can record a variety of detailed processor metrics including number of instructions retired per second and a variety of cache statistics. These counters can be combined with machine learning techniques to identify anomalous behavior as running an application

that is infected can have a different performance counter profile than running the application without malicious code added in. The results show some promise in detecting malicious threads of execution based on this approach. An important challenge with this approach is that some infected applications can differ only slightly from the uninfected application which raises the possibilities of false positives.

3.3.4 TPM

The Trusted Computing Group (TCG) is an international standards group that develops specifications for technologies that "enable a safer computing environment across platforms and geographies [107]." One technology the TCG has developed is the Trusted Platform Module (TPM) which is a small hardware device that can perform hashing, encryption, and store secrets away from malicious code running on the host CPU. In 2004, Sailer, Zhang, et al, proposed a TPM-based mechanism that dynamically measured executable content such as applications, the kernel, and kernel modules on a running Linux system before they were invoked or used [100]. This allowed detection of unauthorized changes done by operating system rootkits. The mechanism supported three methods for invoking measurements: the file_mmap Linux Security Module hook to trigger a SHA1 measurement of the file, the addition of a measure function call in the kernel code path that relocates kernel modules in memory, and an addition to the `/etc/security/measure` interface to trigger a measurement on a given file descriptor.

As many files can be in use on a system at a given time, the mechanism reduced some of the overheads by avoiding the need to use the TPM on each file measurement, using a kernel cache of known measurements. The mechanism only extended the measurement to the TPM if it was not in the kernel cache (e.g. it had not been measured before.) A TPM extend operation

writes a hash to a special register (PCR/Platform Control Register) on the TPM device. Each hash is the result of a hash of a new value and the existing PCR value. The heaviest latencies in this approach result from the use of the TPM which is a relatively slow device. While a SHA1 hash could be accomplished in 4.21μs, the TPM extend operation increased latencies to 5430μs. The authors note, however, that TPM extend operations are relatively rare due to their optimizations so the ultimate performance impact from the TPM operations in this approach should be minimal.

3.3.5 Late Launch

Both Intel and AMD have developed security mechanisms in their CPUs that build upon the TPM, namely Intel TXT (Trusted Execution Technology) and AMD SVM (Secure Virtual Machine). A key feature of Intel TXT is that it can perform a measured launch of the operating system or hypervisor. This capability allows inspections of the boot loader as well as key system files and drivers before launching the operating system or hypervisor. If these files have changed, one of the options available to set in the launch policy is to reset the system, preventing further progress into a compromised environment. This provides protection at boot time; however, it is still possible for the system to become compromised after the system has been running for an extended period of time.

In 2008, McCune, Perrig et al. proposed Flicker which represented a novel usage of SVM and the TPM to execute a variety of applications in a context that was isolated from the host CPU. One supported application was a rootkit detector [75]. A primary benefit of CPU-based security mechanisms such as TXT and SVM is that they support a functionality called "Late Launch". This allows a secure launch at any time of a Virtual Machine Monitor or "Security Kernel at an arbitrary time with built-in protection against software-based

attacks." Thus upon the CPU issuing the instruction to launch the secure environment, DMA is disabled to the memory pages to be used by the secure environment, interrupts are disabled, and debugging access is disabled as well. In order to demonstrate that the secure environment was properly invoked, a measurement (hash) of the launched secure code is stored in a TPM register (PCR). With this mechanism, Flicker facilitates a secure launch of application code and can demonstrate that the code that was launched was the code that the administrator expected to run. Flicker's design limits the securely-launched code to ring 3 (user-mode) applications to prevent them from modifying the underlying operating system. While this provides protection against an errant or malicious application, it also reduces the visibility of a rootkit detector application running on Flicker.

The assurances that Flicker provides over the securely launched code do come at fairly high system impacts. The cost of the rootkit detector's hash of the OS kernel (22 ms) is dwarfed by the time required by the TPM to securely write the measurement ("quote") of the executed code and its arguments. The latter took between 331 and 972 ms depending on the TPM manufacturer. Beyond these impacts, Flicker also runs with the operating system "suspended and interrupts disabled" [75]. For this reason, the authors note that the user "will perceive a hang on the machine. Keyboard and mouse input during the Flicker session may be lost... The most significant risk to a system during a Flicker session is lost data in a transfer involving a block device, such as a hard-drive, CD-ROM drive, or USB flash drive" [75]. While Flicker does provide strong isolation from rootkits, there are several significant drawbacks: large performance impacts and an unsupported ability to run rootkit detectors at privilege levels greater than the operating system.

3.3.6 ARM TrustZone

The ARM processor commonly used in cell phones and tablets does not feature SMM, however, its TrustZone architecture has some similarities to SMM. TrustZone is a set of extensions to the ARM CPU that provide for a normal world and a secure world. Code running in the normal world is unable to access resources that are limited to code running in the secure world. The mechanism to transitioning between the normal and secure world is the Secure Mode Call instruction. Similar to SMM, the secure world has full access to host memory (normal world memory). The challenges of the TrustZone mechanism are that compromises in the normal world may be undetected if the secure world does not intercept malicious changes in system state. For example, the secure world is unable to intercept page fault exceptions and certain control instructions [8].

One enhancement to TrustZone is the TrustZone-based Real-time Kernel Protection (TZ-RKP) mechanism. TZ-RKP routes certain security-sensitive functions through the secure world for "inspection and approval before being executed" [8]. This has similarities and differences to common SMM usage. SMM can perform platform management tasks on behalf of less privileged operating system code, however, it is not tightly integrated into the operating system in the manner done by TZ-RKP. TZ-RKP inserts hooks into the operating system to call into the secure world to perform sensitive security operations (e.g. specifying location of memory translation tables and exception handlers) on behalf of the operating system in a more trust-worthy environment. TZ-RKP is presently deployed on Samsung Galaxy smartphones and tablets, including the Samsung Galaxy Note 3 and S5 phones.

A class of attacks that TZ-RKP can not detect are those in which the attacker tricks the kernel into modifying important data fields in its own

memory. These attacks could compromise control flow integrity in which the intended sequencing of kernel operations is changed. However, this would not result in compromise of the secure world as this remains in a separate non-compromised context and also state changes that require a call into the secure world still would not be possible. TZ-RKP assumes that the kernel can support a design in which pages are mapped exclusively as executable pages or write pages. From the performance perspective, TZ-RKP incurs overheads from 0.19% to 7.65% on a set of benchmarks surveyed by the designers. These degradations are influenced by frequent transitions to and from the secure world at the cost of 2,000 cycles for each transition. While 2,000 cycles on a 2.3 GHz CPU is not that large, there is additional work to be performed in the secure mode which would add to this cost, and with frequent occurrences, this has the potential to impose a noticeable performance impact. The designers "recommend that TZ-RKP's performance overhead should be carefully examined before it is implemented in a production environment" [8].

3.3.7 SMM-RIMMs

Firmware-based mechanisms leverage a special execution mode on the x86 CPU called System Management Mode. This mode provides "an alternate operating environment that can be used to monitor and manage various system resources for more efficient energy usage, to control system hardware, and/or to run proprietary code" [45]. AMD x86 CPUs also feature SMM [2] and Intel's Itanium CPU features a PMI that is similar in concept to an SMI [46]. SMM is designed such that neither privileged software nor applications can inspect its memory (SMRAM) or directly detect time spent in this mode. SMIs can occur for a variety of reasons including: reporting of hardware errors, thermal throttling, power capping, and system health checks [70]. SMIs can be synchronous via a CPU instruction or asynchronous from the chipset [26]. The

potential exists for an SMI to preempt time-sensitive code (e.g. code holding a global lock on one node in a cluster), resulting in delays well beyond what the software developer may have expected. The x86 architecture features a variety of different types of exceptions and interrupts. SMIs are unique in that they are a higher priority interrupt than Non-Maskable Interrupts (NMIs) and device interrupts. SMM has the benefit that other interrupts will not preempt it, but has the side effect that other device interrupts will only be handled after it has finished its work [7]. As SMM is broadly available on x86 CPUs, there is not additional hardware to be deployed to take advantage of it and developers can leverage it for new uses by modifying the BIOS.

From the perspective of RIMMs, SMM has several useful properties. SMM has the ability to preempt running host-side code and any malware in that region. Thus, even if host software is fully under the control of malware, one SMI can cause all of the CPUs to exit SMM into an alternate context. Thus, any malicious processing on any CPU thread is preempted for the duration that the CPU threads are in SMM. Once there, SMM has the necessary privilege to inspect and even modify host software, thus SMM has a high degree of privilege. SMM has broad visibility as once an SMI occurs, each interrupted CPU thread's context is saved in SMRAM where SMM can inspect it. In this way, it has straight-forward access to a very useful set of CPU registers that control host software execution including the CR3 register that controls paging, the IDTR that controls interrupt-handling, and the GDT that controls memory segmentation. SMM-RIMMs also benefit from strong isolation from host software. Even if an operating system or hypervisor is under full control of the adversary, SMM is still intact. Transitions into SMM are quicker than performing a TXT launch which can allow for more frequent transitions than could be accomplished with TXT-based mechanisms.

Several examples of SMM-RIMMs are HyperSentry [7], HyperCheck [113], and SPECTRE [119]. HyperSentry's developers created a mechanism to allow for "stealthy in-context integrity measurement of a running hypervisor (or any other highest privileged software)" [7]. HyperSentry relies upon a modified SMI handler that works in conjunction with a measurement agent that runs in the hypervisor. Each integrity measurement is triggered by an SMI that is generated by a server management device (BMC). Upon receiving an SMI, the CPU threads enter the SMI handler and the source of the SMIs determines whether processing will be handled by the HyperSentry SMI code or the standard SMI code. When the HyperSentry SMI code receives control, it measures the content of the measurement agent to ensure that it has not been compromised. Assuming the measurement agent is unmodified, HyperSentry sends one CPU thread to the measurement agent to perform the hypervisor measurement. While this measurement is in progress, the other CPU threads wait.

By leveraging a hypervisor-based measurement agent, HyperSentry avoids the semantic gap that results from trying to look in to the operating system or hypervisor from outside of that context. While SMM has the privileges to perform this examination, it does not have the knowledge to comprehend most of the data structures in use by the host software. This presents a quandary as RIMMs benefit from strong isolation from host software, yet, also need the ability to comprehend their key data structures. While developers could choose to make SMM aware of the set of data structures in use, this dramatically increases the amount of code that must be brought into the SMM-RIMM and would also necessitate updates when host-side data structures were revised. At the same time, simply placing a measurement agent in host software without a mechanism to check its integrity before usage, puts the

RIMM mechanism at risk although it solves the immediate problem of the semantic gap. For this reason, HyperSentry provides a very useful example of a solution that achieves the benefits of SMM protections and also bridges the semantic gap between SMM and host software.

HyperCheck is another SMM-RIMM that "aims to detect the in-memory, Ring-0 level (hypervisor or general OS) rootkits and rootkits in privileged domains of hypervisors" [7]. Unlike HyperSentry, HyperCheck places its measurement agent entirely in SMM and also relies upon a PCI-based network device. The SMM code performs measurements on CPU registers and the network card gathers the contents of memory for analysis. Like HyperSentry, HyperCheck is invoked with an SMI that gives control to the SMM-RIMM and begins the measurement process. Because HyperCheck does not have a measurement agent in the hypervisor, it relies upon statically compiled locations of code in the hypervisor symbol table for IDT table, hypercall table, and exception table locations. This makes it less dynamic but avoids the need to measure a hypervisor agent, instead relying upon SMM memory protections.

SPECTRE is a SMM-RIMM that examines hypervisors, operating systems, and user processes for certain attacks such as heap spray, heap overflow, and rootkit detection. One key feature of SPECTRE is that is resides solely in SMM and bridges the semantic gap by rebuilding the necessary semantic information for the operating system. For example, in order to comprehend the Kernel Processor Control Region (KPCR) in Windows, it relies upon the data to be present at a hard-coded virtual address (0xffddff000). The SMM code can use the CR3 register to walk the page tables to determine where in physical memory the hard-coded virtual address resides. Once the KPCR is located, the SMM code relies upon a pointer at offset 0x34 that points to

the KdVersionBlock. The code then checks offset 0x78 of this data structure to get to the start of linked lists of pointers to executive process structures. While this does avoid the need for an in-context measurement agent, it does present challenges of closely relying upon the current definitions of data structures. Future software changes could make changes that would break the assumptions built into this SMM-RIMM.

STM/PE [83] is a new approach that utilizes Intel's SMI Transfer Monitor (STM) to virtualize a measurement agent running in SMM. The STM is an SMM-based hypervisor that runs the SMI handler in a virtual machine with the goal of constraining its platform accesses to what is allowed by administrator policy [48]. STM/PE uses the Xen Hypervisor Integrity Monitor (XHIM) for its integrity measurements. This module is based off of the Linux Kernel Integrity Monitor (LKIM) [88]. As STM/PE utilizes the STM, it is also able to resolve SMM-RIMM Challenge 1 (C1-SMM-RIMM Privilege) similar to Vibhute's independent work [110] although with a different implementation.

3.4 Timeline of Approaches

Researchers have developed a variety of RIMMs using different software and hardware devices over the years. While the implementations are varied, there is a clear trend to leveraging hardware instead of earlier approaches that were purely software-based. Additionally, alternate approaches that endeavor to improve the security of operating systems and hypervisors represent alternate approaches to improving system security. Figure 3.1 depicts selected RIMMs and system security developments over time. One of the earliest hardware-based RIMMs, CoPilot helped begin the transition from pure software approaches to leveraging hardware for runtime integrity. This trend has continued with the chipset, TXT, SMM, CPU virtualization, memory modules,

System on a Chip, and TrustZone approaches. Table 3.1 compares selected approaches based on where they place their measurement agent, whether they have direct registers and memory access, their primary source of performance overhead and whether they occur periodically or are event-driven.

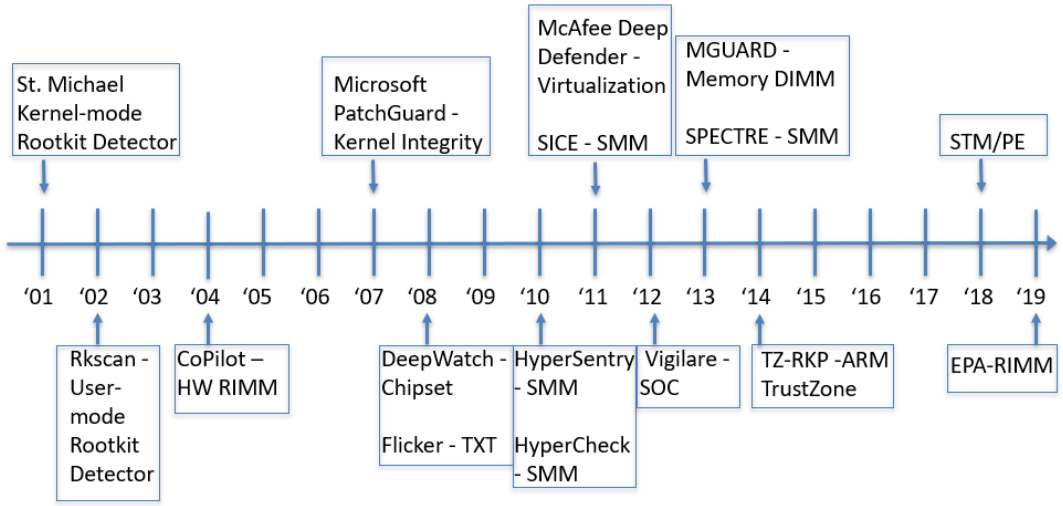

FIGURE 3.1: Timeline of RIMM Approaches from 2001-2019

TABLE 3.1: Comparison of Selected Runtime Integrity Monitor Approaches

Mechanism	**Location**	**Register Access**	**Memory Access**	**Performance Overhead**	**Periodic or Event-Driven**
PatchGuard	OS Code	Y	Y	OS Checking	Periodic
CoPilot	HW Device	N	Partial	Memory/PCI	Periodic
HyperSentry	SMM VMM	Y	Y	SMM transition	Periodic
HyperCheck	SMM	Y	Partial	SMM transition	Periodic
Deep Defender	VMM + OS	Y	Y	VMM transition	Event-Driven
Flicker	TPM	N	Y	TPM	Periodic
STM/PE	STM Guest	Y	Y	SMM transition	Periodic On-demand

3.5 Application Noise

The impact of SMIs relates to previous workload perturbation studies that examined the effect of noise from software heartbeats and system daemons [89],

hardware interrupts [10], and network interrupts [108]. This is relevant to our study since SMIs are the highest priority interrupt and take precedence over other interrupt sources. This is an extension to the form of noise called a detour that Beckman et al. describe that occurs when "an application is temporarily suspended to process an OS-interrupt" [10]. In the case of SMIs, however, the entire operating system or hypervisor and any running applications are temporarily suspended during SMI processing.

Ferreira et al. [30] found that noise's effect on an application may be reduced by absorption; but, the impact of noise can be amplified when it occurs at a performance-sensitive time. While noise can be a significant factor, its effect on the application may be reduced by absorption in which the full impact of noise is not reflected in the workload as it is hidden by other factors. An example of this is when MPI processes are in MPI_WAIT state, the full impact of noise is reduced since the processes are in a waiting state. The impact of noise can be amplified when it occurs at a performance-sensitive time such as MPI_Allreduce, MPI_Bcast, and MPI_Barrier [89]. These factors are relevant to SMIs as they could occur at performance-sensitive times which would amplify their effect or at times when their impact would not be noticed, e.g. while the application is blocked while waiting for the disk controller to fetch data.

4
Creation of Methodology for SMI Performance Measurement

At the outset of this work, there were no available SMM-RIMM implementations. This resulted in challenges in building a performance-efficient RIMM leveraging SMIs. SMIs are unlike other types of interrupts as they occur transparently to the operating system. This results in the operating system scheduler not being aware of their occurrence and analyzing their impacts is not trivial.

To better understand design trade-offs and design a performance-efficient SMM-RIMM, we needed a methodology to characterize the system impacts due to varying frequencies and durations of SMIs. The methodology would need to evaluate scenarios that represented the current state of the art with SMIs of 25-40ms duration similar to HyperCheck and HyperSentry along with shorter durations that corresponded to a time-sliced integrity measurement method. The methodology builds upon the observation that the time spent in SMM would be equivalent to the impact of an SMM-RIMM as both take time away from the running system. We needed a way to address potential challenges to this assumption, for example, CPU cache perturbation introduced by activities in SMM.

In Section 4.1, we describe the SMM-RIMM performance methodology requirements. As no existing performance methodology existed, we needed to define one. In Section 4.2, we describe related work on SMI detection which was the only related work pertaining to the performance methodology. In Section 4.3, we present an overview of the development of these methods. Section 4.4 describes our chipset-based method of generating a predictable SMI load. Section 4.5 describes our method of leveraging the existing SMIs on a platform to accomplish deterministic SMI loads. Section 4.6 presents

our modified BIOS methodology in which we added a mechanism to delay all CPUs inside SMM for a user-specified duration. Section 4.7 describes our use of EPA-RIMM as a method of providing insights into SMM-RIMM scheduling and system impacts. Section 4.8 summarizes our analysis of these different methods. Section 4.9 demonstrates how we validate that the generated SMIs take the expected amount of CPU cycles by analyzing impacts on predictable workloads. Section 4.10 shows how we generate the SMIs used by our various methods. In Section 4.11, we present our CPU cache and prefetching studies that give additional insights into the degree of determinism in measurement costs. Section 4.12 provides our conclusions over our performance measurement methodology and its implications for EPA-RIMM.

4.1 SMM-RIMM Performance Methodology Requirements

As the performance aspects of SMM-RIMMs have not yet been given a detailed study, a methodology to understand their performance impacts had not yet been created. This methodology would allow understanding the performance impact on applications running while inspections are occurring. It would also allow characterization of the operating system impacts resulting from the inspections. These understandings would help guide the creation of an SMM-RIMM without significant impacts on system performance, correctness, or stability.

In creating the SMM-RIMM performance methodology, we identified three key requirements. The first requirement, **Rquantify**, is the ability to quantify time spent in SMM (Section 4.1.1). If the amount of time spent in SMM cannot be measured, it is not possible to create a performance efficient SMM-RIMM. The second requirement, **Rcontrol**, is that the methodology

needs to be able to control the amount of time spent in SMM. If the time spent in SMM is uncontrollable, there is no ability to bound execution times (Section 4.1.2). The third requirement, **Rvalidate**, is that the performance methodology must be able to validate that the time spent in SMM matches expectations (Section 4.1.3). This provides assurance that SMI times are repeatable and deterministic.

4.1.1 Ability to quantify time spent in SMM - Rquantify

Time spent in SMM has a different property than in the operating system environment. In the latter, the operating system scheduler is able to schedule CPU threads for specific durations and preempt them to allow another process to run. With SMIs, the operating system is unaware that SMIs are occurring and not able to preempt their execution. It also does not feature a mechanism to quantify the time spent in SMM. In this section, we describe our method to quantify the time spent in SMM.

Step 1: **Calculate CPU cycles spent in SMM.** To allow quantification of time spent in SMM, we placed a starting CPU timestamp using the RDTSC (Read Timestamp Counter) just before the SMI was generated with an OUTB instruction. After the timestamp, the OUTB instruction is executed and the CPU threads will transition to SMM ("Transition to SMM"). While there, the work of the SMI is accomplished ("Process SMI") and the CPUs will return back to the operating system or VMM context ("Transition to OS/VMM"). When the CPUs return from the SMI, they OUTB instruction is complete and they will log the ending timestamp. Figure 4.1 depicts the flow. Next, we subtract the beginning timestamp value (*StartClock*) from the ending timestamp

value (*EndClock*) to obtain *SmiDelta* (the total number of CPU cycles spent in the SMI flow.) Equation 4.1 shows this calculation.

$$SmiDelta = EndClock - StartClock \tag{4.1}$$

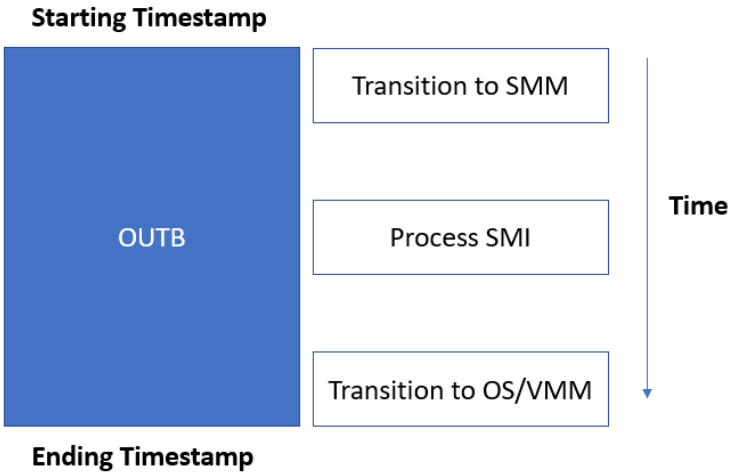

FIGURE 4.1: Timestamp method

Step 2: **Convert CPU cycles to a time measurement.** Counting time on modern CPUs has become more complicated with CPU frequency scaling features that vary the clock speed of the CPU to save power in times of reduced load. However, Intel has added a constant TSC feature [45] which keeps the rate at which the timestamp counter increments constant independent of the CPU clock frequency.

To calculate the SMI duration in μs (*SmiMicroseconds*), we take *SmiDelta* and divide by the CPU clock frequency (*ClockFreq*). We then multiply this by the number of μs in a second (*NumUsecsInSec*) which is 1,000,000. Equation 4.2 shows this calculation.

$$SmiMicroseconds = (SmiDelta/ClockFreq) * NumUsecsInSec \tag{4.2}$$

Step 3: **Calculate SMIs per second**. Beyond the time spent in a single SMI, the other key aspect that impacts performance is the number of SMIs per second. The total number of SMIs (*TotalSmis*) over a time interval (*SmisPerSec*) can be calculated by a script taking a starting reading of MSR_SMI_COUNT (*SmiCountStart*), sleeping for the desired duration, e.g. 300 seconds, *TimeDelta*), taking an ending reading of MSR_SMI_COUNT (*SmiCountEnd*), and subtracting *SmiCountStart* from *SmiCountEnd*, then dividing by *TimeDelta*. Equations 4.3 and 4.5 shows this calculation.

$$TotalSmis = (SmiCountEnd - SmiCountStart) \tag{4.3}$$

$$SmisPerSec = (TotalSmis)/TimeDelta \tag{4.4}$$

$$SmisPerSec = (TotalSmis)/Time \tag{4.5}$$

Once we have determined the number of SMIs per second and the cost per SMI, we can calculate the total number of microseconds spent in SMM in a given second. Equation 4.6 shows this calculation.

$$SmiCyclesPerSec = SmiMicroseconds * SmisPerSec \tag{4.6}$$

We can also calculate the percentage of CPU cycles spent in SMM (*PercentageSmiCycles*) by taking the *SmiCyclesPerSec* and divide it by

the number of CPU cycles per second (*ClockFreq*) as shown in Equation 4.7. This equation is useful for comparing the percentage of workload degradation to the percentage of time spent in SMM.

$$PercentageSmiCycles = SmiCyclesPerSec/ClockFreq \tag{4.7}$$

4.1.2 Ability to control time spent in SMM - Rcontrol

The performance methodology needs to address SMI duration and frequency scenarios that allow: 1. Understanding the performance impacts of the currently proposed approaches such as HyperSentry and 2. Supporting scenarios that would allow workload decomposition approaches to reduce the negative impacts of prolonged time in SMM.

We describe these scheduling approaches:

1. Non-Decomposed scheduling (Long duration and Infrequent SMI scheduling). This approach does not bound SMI processing times and can consume times in the order of 35-40ms for HyperSentry and HyperCheck

2. Decomposed scheduling (Shorter duration and Frequent SMI). This is our approach that decomposes large measurements into tasks that are designed to allow meeting the SMI latency guidelines.

Beyond scheduling, there are impacts that impact how much time is spent in SMM. These include factors such as CPU caching, CPU prefetchers, and power-savings methods. CPU caching can impact SMM performance and thus reduce or prolong time spent in SMM. CPU prefetchers can predictively fetch data into SMM to reduce memory access times. Power savings mechanisms can disable portions of the CPU to reduce power usage during idle times. As it takes time to transition out of these lower power states, the time required to enter SMM could be increased.

4.1.3 Ability to validate SMI load - Rvalidate

As time spent in SMM is not directly observable by the operating system and its tools, producing a methodology of properly accounting for the time becomes necessary. Validating that the expected SMM duration and frequency produced by the SMI scheduling approach accurately matches the expectations is required. Without a mechanism of validating these two factors, it is not possible to determine the impacts on applications or operating system. Additionally, one concern that we wanted to address with our methodology was whether the SMI activities performed by the chosen SMI would perturb the system beyond the amount of CPU cycles taken away.

4.2 Related Work

The two primary existing tools that can detect the time taken by a single SMI (*SmiDelta*) and the number of SMIs over a time period (*SmisPerSec*) are Jon Masters' "simple SMI detector [71]" and "BIOSBITS [106]". The former work detects loss of operating system control by hogging all of the CPU(s) for configurable time intervals, looking to see if something stole time...". BIOSBITS has similar functionality and can "detect system management interrupts by watching for large gaps in time between successive values of the time-stamp counter." These approaches can be helpful to determine SMI costs but they rely upon monopolizing the CPU in order to detect when control is lost, which makes them unsuitable for performance measurement as this technique does not allow also running an application benchmark. One additional capability is an SMI counter on recent Intel CPUs called MSR_SMI_COUNT and it increments with every SMI that has occurred [45]. This counter can be sampled over time to determine if SMIs are occurring and their frequency.

For SMI generation, an SMI can typically be generated from a device driver by writing a value to IO port 0xB2. However, this does not provide a large amount of measurement flexibility as the SMI duration depends on the BIOS codebase and system particulars. While the capability of generating different SMIs existed via writes of different values to IO port 0XB2, it had not been used for SMI generation to be equivalent to the durations of SMM RIMMs.

4.3 Measurement Methodology Creation

At the outset of this work, we did not have any existing SMM-RIMMs to examine. We did also not have access to BIOS code that would enable us to build our own SMM-RIMM and measure its impacts on the system. For this reason, we settled upon an approach that would inject SMIs that were already enabled by the BIOS manufacturer but not turned on at runtime. Our steps were as follows:

1. We examined the Intel chipset documentation [44] to identify potential sources of SMIs that we could enable. We determined that there were a number of SMIs that could be triggered by writing to different chipset registers. This approach would allow us to generate a high rate of short SMIs, which would accurately model a decomposed SMI scheduling method. However, it did not allow us to inject SMIs that would be equivalent to a Non-Decomposed scheduling method.

2. To address the limitation of the chipset SMI methodology, we investigated trying to trigger other SMIs that were enabled on the system that were accessible via a device driver that would write a value to IO Port 0xB2. Depending on the value written to the port, a different SMI would fire. As each SMI had a pre-defined purpose, they each took a

unique amount of time. At this stage, we were able to generate SMIs of long durations and schedule them to repeat periodically, just as an SMM-RIMM would do. However, we were concerned about side-effects of running unknown SMIs. This uncertainty encouraged us to develop our SMI validation approach to examine whether the SMI duration directly correlated to its impact on predictable CPU-bound applications.

3. At this stage, we had a breakthrough as we got access and approval to use a development system and add our own SMIs with specific durations. We implemented a busy-wait capability in SMM to consume the desired amount of time. At this stage, we had a high degree of control over SMI times and could generate them according to the frequencies we needed. The only lacking element was that a busy-wait driver was not performing the types of actions an SMM-RIMM would perform. A key limitation of this approach was that the firmware was not open-source and we could not share or publish detailed information about it.

4. In 2014, Intel released the Minnowboard single-board computer with open-source UEFI firmware. This provided everything that we needed to develop our own SMM-RIMM, measure its performance, and release our prototype as open-source for the research community.

4.4 Technique 1: Chipset SMIs

Examining the various SMI generation sources in the chipset documentation, we identified two potential SMI generators: PERIODIC_SMI and the SWSMI_TMR (Software SMI timer) and created a Linux device driver to enable these options. The periodic SMI hardware supported generation of one SMI every [8, 16, 32, 64] seconds which we determined was too light an SMI

load for our purposes given its infrequent triggering and short duration that was within SMI latency guidelines. The SWSMI_TMR approach supported a high frequency of SMIs per second. While these were shorter SMIs that were within SMI latency guidelines, they well-represented a time-sliced approach of SMM-RIMM scheduling.

TABLE 4.1: Chipset SMI Generation

Mechanism	**Chipset Register 1**	**Chipset Register 2**	**Measured Results**
Periodic SMI	SMI_EN, Periodic SMI Enable=1	GEN_PMCON_1 bits 1:0: 00 = 64 seconds 01 = 32 seconds 10 = 16 seconds 11 = 8 seconds	One SMI every 8,16,32,64 seconds
SW SMI Timer	SMI_EN, SW SMI Timer Enable=1	GEN_PMCON_3 bits 7:6: 00 = 1.5ms $\pm$ 0.6ms 01 = 16 ms $\pm$ 4ms 10 = 32 ms $\pm$ 4ms 11 = 64 ms $\pm$ 4ms	One SMI every 2,16,32,62 ms

Table 4.1 provides additional detail on the chipset register bits set to enable these SMIs. We selected the SWSMI_TMR feature and planned experiments to characterize the amount of time spent in SMM at the varying frequencies that the SWSMI_TMR mechanism supported (*SmiCyclesPerSec*). As the mechanism has a little variability in the amount of SMIs generated (e.g. "1.5 ms $\pm$ 0.6 ms"), we first performed an experiment to determine the SMI rate (*SmisPerSec*). We created a Linux shell script to count the number of SMIs over a 300 second interval (*SmisPerSec*) with TimeDelta set to 300 seconds. We set the SWSMI_TMR SMI generation frequency by adjusting bits 7:6 in the GEN_PMCON_3 register, choosing the '00' value that selected the "1.5 ms $\pm$ 0.6 ms" duration and then turned on SWSMI_TMR SMI generation by setting bit 6 in SMI_EN. With SMI generation enabled, we ran our shell script that took an initial sample of the MSR_SMI_COUNT (*SmiCountStart*)

to determine the starting number of SMIs, slept for 300 seconds, and then took an ending sample of MSR_SMI_COUNT (*SmiCountEnd*). We determined that the mechanism was resulting in 500 SMIs/second (*SmisPerSec= 500*). We repeated this process for the other three supported frequencies several times to ensure that the achieved results were stable over time.

Now knowing the SMI frequency, we needed to determine the total amount of CPU time taken away by the SMIs which would represent the time spent in an SMM-RIMM (*SmiCyclesPerSec*). To determine this, we applied the techniques of BIOSBITS [106] and the "Simple SMI Detector [71].” We added functionality to our device driver to calculate this by doing the following steps:

Step 1: Turn on the SWSMI_TMR with the chosen SMI generation frequency.

Step 2: Allow the user to write a value to our driver’s supported proc file system interface to trigger the logic below in Step 3.

Step 3: Upon receiving the trigger, set the driver to busy wait in a tight loop, storing CPU time-stamp counter results and the value of the MSR_SMI_COUNT in a memory buffer.

Step 4: Allow the user to stop the busy waiting and print out the buffer’s contents.

Step 5: Looks for large gaps between successive CPU time-stamp counter results (e.g. > 100000 CPU cycles) that could represent SMIs and cross-check with the MSR_SMI_COUNT value to verify that it showed the counter increasing.

Step 6: Average the durations for each SMI and determine the number of CPU cycles spent per SMI.

With knowledge of the number of SMIs per second (*SmisPerSec*), the cost per SMI (*SmiMicroseconds*), and the CPU frequency (*ClockFreq*), we had the necessary information to calculate the percentage of CPU cycles spent in SMM *PercentageSmiCycles*).

One benefit of the chipset approach is that SMI generation occurs completely out-of-band of the operating system which removes one source of potential variability. Additionally, for an SMM-RIMM, hardware generation of SMIs is preferable to software-based mechanisms as the latter have a dependency on the (untrusted) operating system scheduler and kernel code. The SWSMI_TMR approach is quite representative of the Decomposed SMI scheduling approach in which longer-running operations are split into a larger number of shorter-running tasks. The drawback with the SWSMI_TMR approach was that it did not provide an ability to specify arbitrary SMI frequencies (could not vary *SmisPerSec*) or generate longer SMIs than 0.11ms (*SmiMicroseconds*) on our system. Also, the feature is not supported by all motherboard BIOS implementations as it triggered a hang on some systems.

4.5 Technique 2: Blackbox SMI Generation

The primary drawback with the chipset-based SMI generation approach was that it did not allow long SMIs (e.g. *SmiMicroseconds* $\geq$ 35-40 ms) that would more closely match the system preemptions of HyperSentry and other SMM-RIMMs. As we also needed to measure the impact of longer SMI preemptions, we needed to complement the SWSMI_TMR technique with an approach that would allow for longer time to be spent in SMM. As were still unable at this time to modify the BIOS source, we began looking for ways to generate longer SMIs.

Besides hardware SMI generation, there is another way to generate SMIs

by writing values to a special IO Port called APM_CNT on Intel [45] and "SMI Command Port" on AMD [2]. The port number is typically 0xB2. We added functionality to our device driver to generate write arbitrary values to the APM_CNT port and take a timestamp before the IO write instruction (*StartClock*) and after it (*EndClock*) to determine the SMI duration, *SmiDelta*). With this technique, we measured values from 0 to 0xFF. While some values hung the system, other values reliably generated longer times in SMM ranging from 1 to 1,061 ms on the Dell PowerEdge R410 server. The 35 ms duration, in particular, was closely in the range of the durations that HyperCheck and HyperSentry consumed. Figure 4.2 shows the Blackbox SMI results.

However, our key concern with this approach is that without knowing more about what the SMI was actually doing, we could not rule out the possibility of performance side effects. For example, if the SMI were to adjust the CPU frequency, we would be introducing a side effect into our measurements. For this reason, we utilized the technique discussed in Section 4.9 to provide assurance that the Blackbox SMIs were not impacting performance in a way beyond taking away the expected amount of time in SMM.

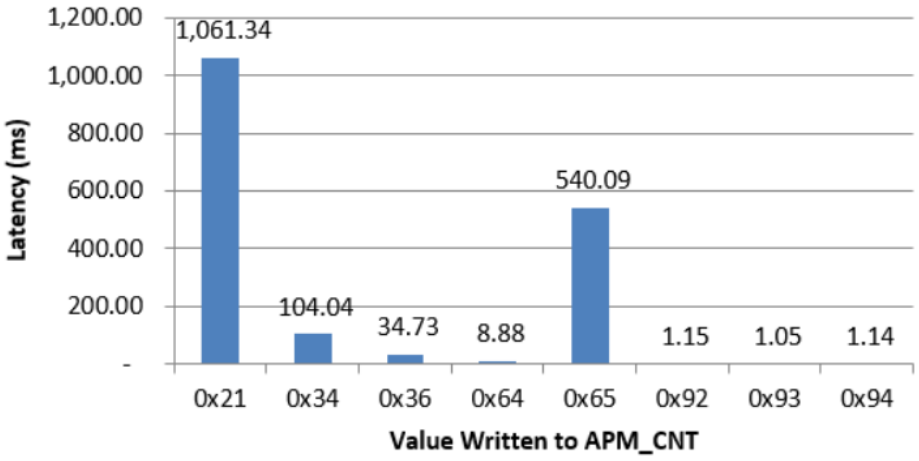

FIGURE 4.2: Long SMIs on Dell PowerEdge R410

4.6 Technique 3: Modified BIOS

While techniques 1 and 2 allowed us to measure time-sliced and longer SMI approaches, we still did not have precise control over our SMI durations. Both approaches limited us to the existing SMI interfaces on the system. For this reason, we obtained access to a development system in which we could recompile the BIOS and add in our own SMI delays. This option is generally not available to end-users on most systems. For the Modified BIOS methodology, we modified our SMI handler to allow a user-configurable amount of delay. In this approach we added twelve values that could be written to the APM_CNT port to generate varying levels of SMI delays: (in ms) 1.43, 5, 10, 20, 50, 99, 495, 990, 5k, 10k, 20k, 64k. When the SMI handler received control, it would delay in a loop for the specified amount of CPU cycles before returning control to the host software. In this way, all CPU threads left the host software and stayed in SMM for the specified amount of time.

The benefits of this approach were that it allowed us to precise calibrate the amount of time to be spent in SMM in a way that was not possible with the chipset and Black Box SMI scenarios. The primary drawback is that that this approach simply busy-waited in SMM without performing any of the actual SMM-RIMM operations. Additionally, at the time, this approach was limited to those with ability to modify the BIOS source code which makes it harder for the researchers without BIOS source to reproduce the results.

4.7 Technique 4: EPA-RIMM

With the release of the Minnowboard Max, a researcher with this platform could replicate our Modified BIOS technique. To improve upon our Modified BIOS technique, we created our own SMM-RIMM allowing precise controls

over SMI durations, frequencies, and measurement operations. Unlike the previous approaches which took time away in SMM to behave like an SMM-RIMM, this approach actually performed actual SMM RIMM operations such as hashing of host-side resources and accessing the saved register state from the SMRAM Save State Map. With this approach, we enabled the SMI handler to handle a new value written to the APM_CNT port that triggered a measurement request. Along with this value, our device driver (Ring 0 Manager) provided the address of a command structure that EPA-RIMM should operate on. As we did with the black box and modified BIOS techniques, we measured the amount of time required to process various measurement operations trigger by SMIs. With the EPA-RIMM test system, we achieved our strongest measurement technique as we were able to generate actual measurement requests involving hashing and registers, along with characterizing their behavior.

4.8 Technique Comparison

With four SMI generation techniques to draw upon, it is possible to test SMI impacts on a variety of hardware. However, not every technique is testable on every given platform. For example, the ability to recompile or modify the BIOS is not possible on many production systems. Or a given system tested with a Blackbox SMI technique may not have SMIs of the desired duration. In Table 4.2, we summarize and compare the four techniques to show whether they support time-sliced RIMM operation, longer SMIs, whether BIOS source is required, compatibility issues, benefits, drawbacks, and when the technique would be useful. Table 4.3 shows the systems we used for the development of these techniques. Ultimately, for RIMM developers, EPA-RIMM surpasses the capabilities of the other approaches as it performs actual SMM-RIMM

operations with fine-grain controls. The SMI generation methodologies could also be applied for other purposes including evaluating impacts of latency on network devices. Figure 4.3 provides a flow chart that allows selecting the appropriate SMI measurement technique.

TABLE 4.2: SMI Generation Technique Comparison

Technique	**Time-sliced?**	**SMIs >1ms?**	**BIOS Source Req?**	**Benefits**	**Drawbacks**	**When to use**
1. Chipset SWSMI _TMR	Y	N	N	Out of band, no BIOS source, time-sliced	Availability, limited SMI durations/ frequencies	Time-sliced
2. Blackbox SMI	Y	Y	N	Long SMIs, broadly available	Could impact performance, limited SMI durations	Long SMIs w/o BIOS source
3. Modified BIOS	Y	Y	Y	Arbitrary duration	Requires BIOS source	When BIOS source is avail-able
4. EPA RIMM	Y	Y	Y	SMM-RIMM	Requires BIOS source	When BIOS source is avail-able

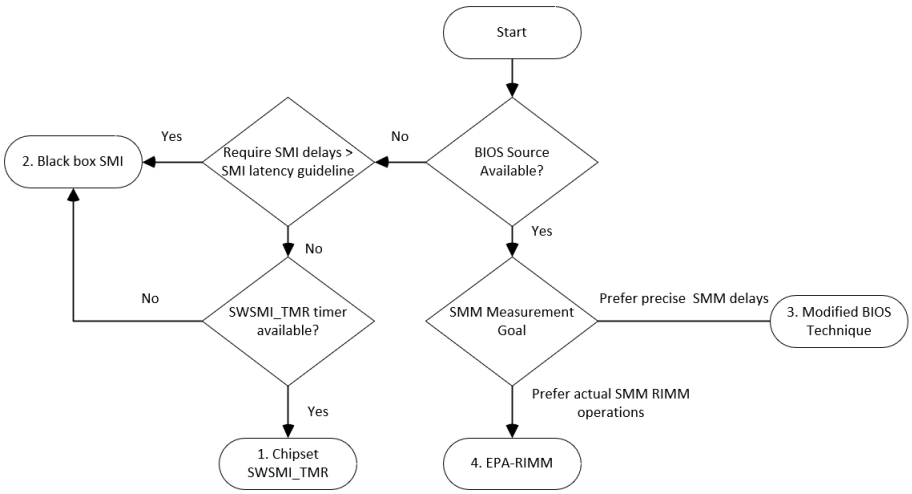

FIGURE 4.3: SMI Measurement Technique Considerations

TABLE 4.3: SMI Generation Techniques Testing

Technique	**Tested on**	**Out of Band?**
Chipset	Intel DQ67SW	Y
Blackbox SMI	Intel DQ67SW/Dell PowerEdge R410	N
Modified BIOS	Intel Nehalem Dev system	N
EPA-RIMM	Minnowboard Max	Possible

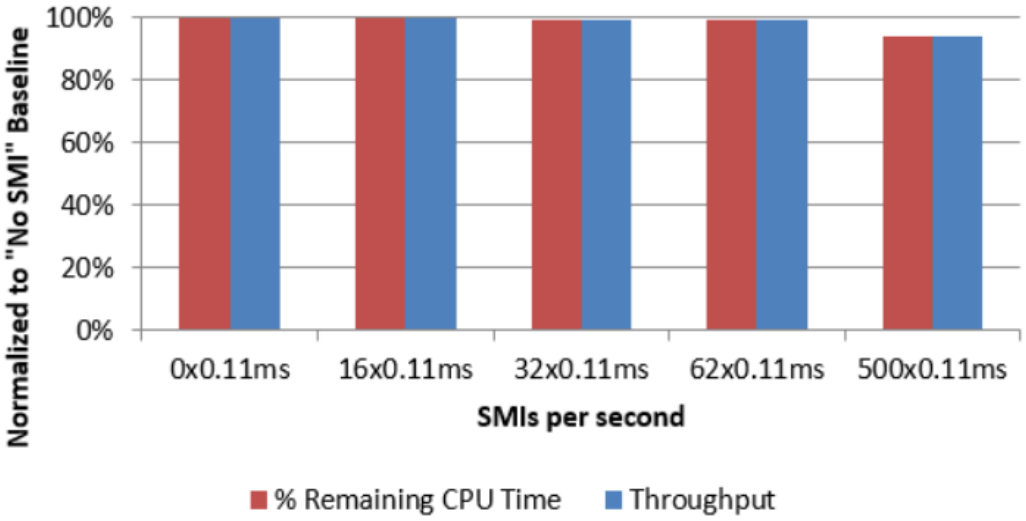

FIGURE 4.4: Chipset SMI Evaluation

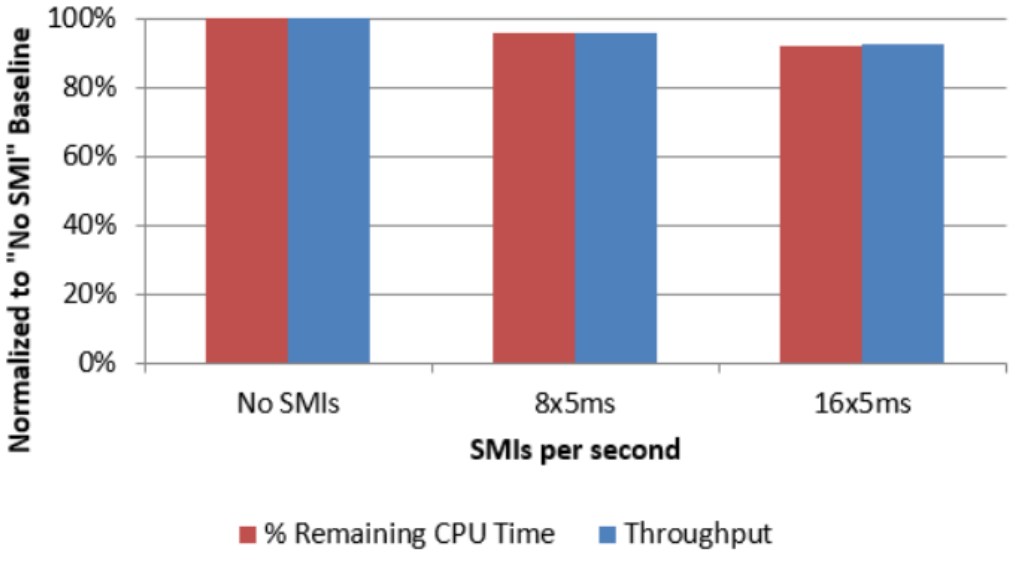

FIGURE 4.5: Blackbox SMI Evaluation

4.9 Validating the SMI Load

For each approach, it was necessary to quantify the SMI impact. As described in Sections 4.4, 4.5, 4.6, 4.7, we were able to determine the duration of the SMIs we generated. As we controlled the amount of SMIs generated per second and knew the CPU frequency, we knew how many CPU cycles we were consuming in SMM and the percentage of CPU cycles remaining to the applications (100% - *PercentageSmiCycles*). At this point, we could perform performance validation to ensure that our calculations were correct. For this purpose, we selected two CPU-intensive microbenchmarks: OpenSSL [86] and Distributed.net's RC5-72 [95]. OpenSSL provides a built-in benchmark utility to measure how many SHA hashing operations can be performed in an interval of time. RC5-72 is a compute-intensive workload that brute-forces cryptographic keys.

We established performance baselines by running the workloads without our injected SMIs. We then injected our SMIs and measured the resulting throughput. We compared the baseline throughput versus the throughput with injected SMIs and determined how closely it matched (100% - *PercentageSmiCycles*) impact. As we expected, for these heavy computational workloads, the time spent in SMM came directly out of the application's throughput as shown in Figures 4.4 and 4.5. The nature of these computational workloads showed that they were unable to maintain their baseline throughput when the CPU cycles they needed for computation were taken away. This presented a useful property for validating the amount of time taken by our various SMI generation techniques. Note that not all workloads have this property as we show in Section 5.2.2 that some workloads are able to hide a portion of the SMI overheads when SMIs occur during portions where the workload is waiting.

4.10 SMI Generation

To turn on or generate each of these SMIs, we needed to develop device drivers for each tested operating system or hypervisor. The Blackbox SMI, Modified BIOS SMIs, and EPA RIMM measurement requests used an OUT CPU instruction executed from Ring 0. The chipset SWSMI_TMR methodology also required Ring 0 privileges as we needed to adjust chipset registers. We developed device drivers for: Xen 4.1.2, Linux (Ubuntu 12/Centos 6), and Windows Server 2012, to trigger SMI delays. For Linux/Xen we used the kernel work queues to schedule our software SMI once a second and also adjust the necessary chipset bits for the SWSMI_TMR method. For Windows Server 2012, we used the kernel function IoStartTimer to schedule one SMI/second. Our test setup additionally allows us to generate a single SMI on demand.

4.11 Task Provisioning

To meet EPA-RIMM's bound on maximum time spent in a single SMI session, accurate estimates over Task costs are essential. SMM's lack of preemption removes one source of non-determinism over Task cost. However, preemption is not the only source of performance nondeterminism. To examine these two impacts on Task cost estimates, we devised targeted experiments analyzing the cache impacts. These experiments help establish an accurate upper bound for Task cost.

The measurement setup used a Minnowboard Turbot with 1.46 GHz CPUs, 2 GB RAM, Ubuntu 14.04.5, Linux kernel 4.11.0.

4.11.1 Cache and Prefetcher Impact Measurement Study

4.11.1.1 Measurement Design

We devised an experiment to examine Task cost variations due to CPU caching mechanisms. The experiment consisted of three scenarios designed to gauge the impact of CPU caching and prefetching performance optimizations.

For each scenario, we crafted a sequence of Bins containing fourteen 4K hash input-sized Tasks. (Fourteen Tasks is the largest number of Tasks that EPA-RIMM currently supports in a Bin.) We disabled CPU power-saving mechanisms which would add another variable to this experiment.

Three measurement scenarios:

1. Identical virtual address: This scenario consists of repeated hash operations to the same virtual address. It is designed to evaluate the ideal measurement case where the memory to be hashed is present in the cache.

2. Sequential virtual address: This scenario consists of a sequence of incrementing virtual addresses to measure. It is designed to evaluate whether the CPU prefetcher feature is able to result in increased hashing performance due to speculatively reading in memory pages before they are needed.

3. Random virtual address: This scenario consists of potentially a worst-case measurement in which data is neither cached or potentially benefitting from prefetching.

Table 4.4 provides the first Bin of the sequence. Subsequent Bins continue the pattern.

TABLE 4.4: Sample Bin

Task #	Identical	Sequential	Random
1	ffffffff8100a000	ffffffff81000000	ffffffff81559000
2	ffffffff8100a000	ffffffff81001000	ffffffff8162f000
3	ffffffff8100a000	ffffffff81002000	ffffffff81531000
4	ffffffff8100a000	ffffffff81003000	ffffffff814d9000
5	ffffffff8100a000	ffffffff81004000	ffffffff814e6000
6	ffffffff8100a000	ffffffff81005000	ffffffff817a1000
7	ffffffff8100a000	ffffffff81006000	ffffffff813ba000
8	ffffffff8100a000	ffffffff81007000	ffffffff816eb000
9	ffffffff8100a000	ffffffff81008000	ffffffff81314000
10	ffffffff8100a000	ffffffff81009000	ffffffff81767000
11	ffffffff8100a000	ffffffff8100a000	ffffffff81295000
12	ffffffff8100a000	ffffffff8100b000	ffffffff81380000
13	ffffffff8100a000	ffffffff8100c000	ffffffff817b2000
14	ffffffff8100a000	ffffffff8100d000	ffffffff81763000

4.11.1.2 Identical Addresses

Examining the variation between 100 Bins in the identical memory address hashing scenario shows that the hash costs appear quite regular with minimal variation. Figure 4.6 shows the measurement results. Table 4.5 shows the statistical summary. We observe that the average cost (0.1352ms) is quite close to the minimum cost (0.1351ms) with only a standard deviation of 0.000098ms.

Figure 4.7 provides data showing the first eight Bins. The data shows a pattern in which the first Task in the Bin experiences a higher cost than subsequent Tasks in the Bin. This indicates that the first Task in the Bin experiences a cache miss. Once the data is cached for the subsequent Tasks in the Bin, they achieve lower measurement costs.

TABLE 4.5: Identical Hash Scenario Cost Analysis

	Identical
	ms
Average	0.1352
Min	0.1351
Max	0.1360
Median	0.1352
StdDev	0.000098

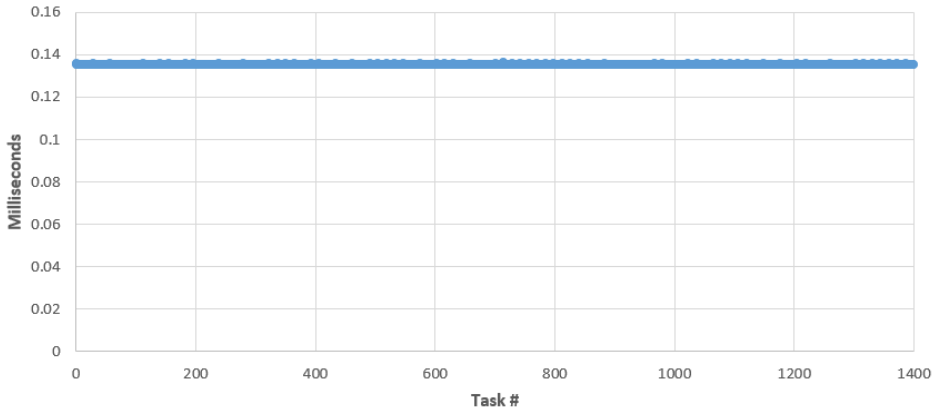

FIGURE 4.6: Identical Address 100 Bins

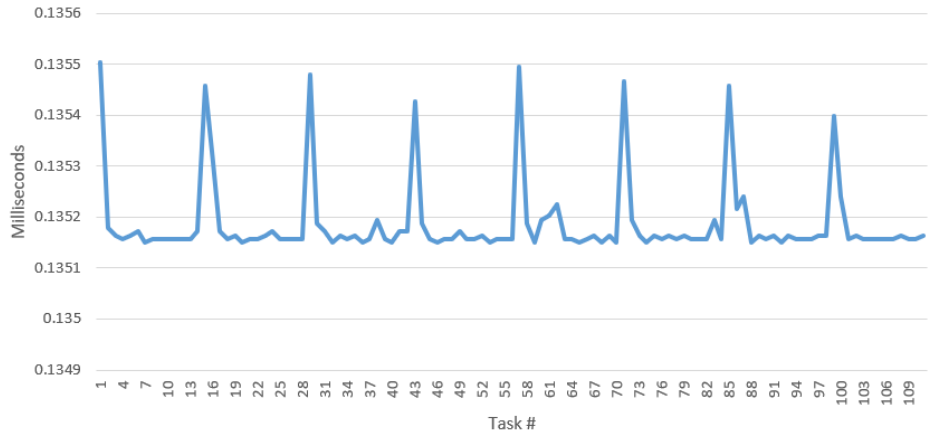

FIGURE 4.7: Identical Address 8 Bins

4.11.1.3 Sequential Addresses

The sequential address scenario measurement results as shown in Figure 4.8 show that the scenario has costs that do not significantly vary from measurement to measurement. However, examining a smaller section of data (first

100 Bins) as shown in Figure 4.9 shows that the costs do not have the same pattern as the identical measurement scenario. The first Task in the Bin is not necessarily the most costly Task. Table 4.6 shows the statistical summary.

TABLE 4.6: Sequential Hash Scenario Cost Analysis

	Sequential
	ms
Average	0.1353
Min	0.1352
Max	0.1361
Median	0.1353
StdDev	0.000088

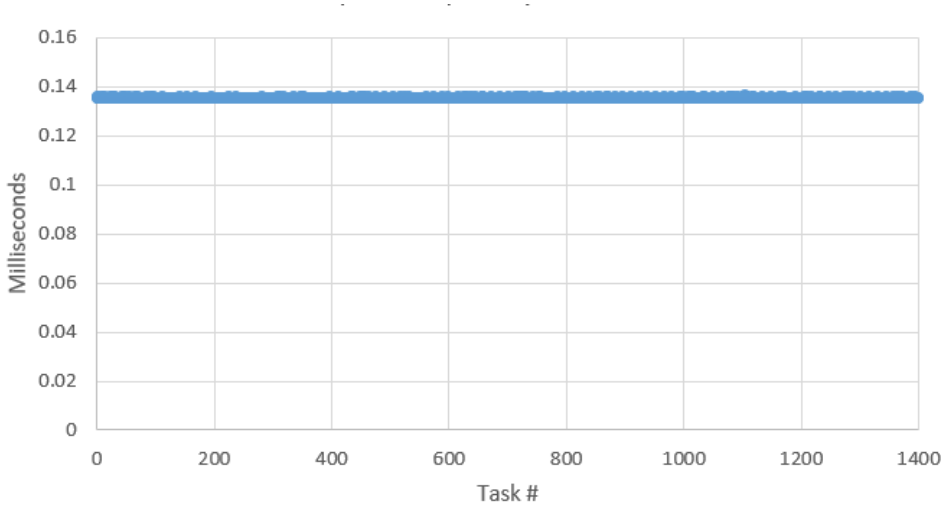

FIGURE 4.8: Sequential Address 100 Bins

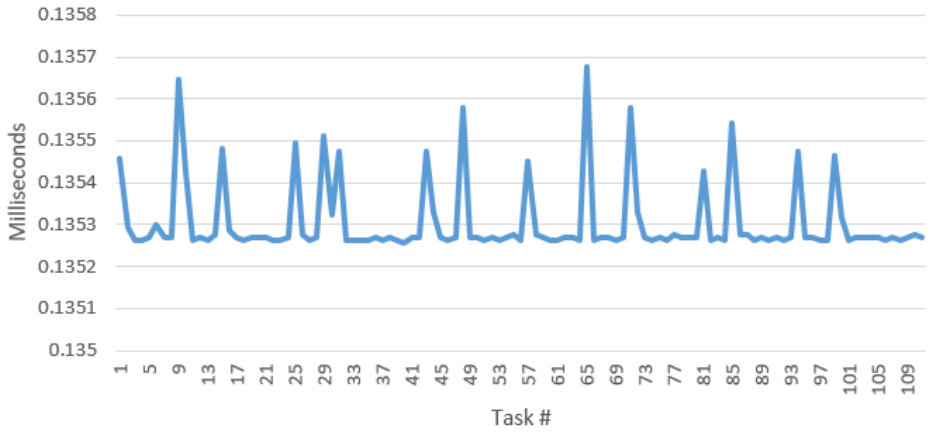

FIGURE 4.9: Sequential Address 8 Bins

4.11.1.4 Random Addresses

The random address scenario does not have obvious variation when viewing 100 Bins of data as shown in Figure 4.10. However, when examining a smaller sub-section of the first eight Bins as shown in Figure 4.11, we observe that the Task costs vary without a clear pattern. Thus, the caching effects do not benefit these measurements and also the prefetcher is not able to measurably improve the results. Table 4.7 shows the statistical summary.

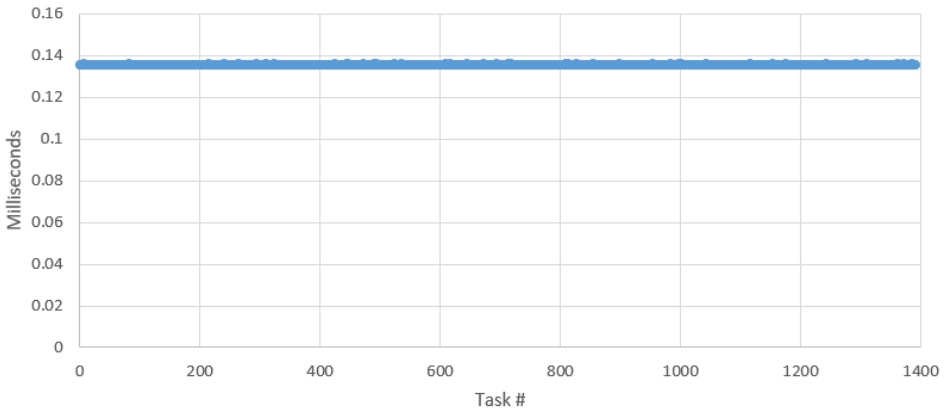

FIGURE 4.10: Random Address 100 Bins

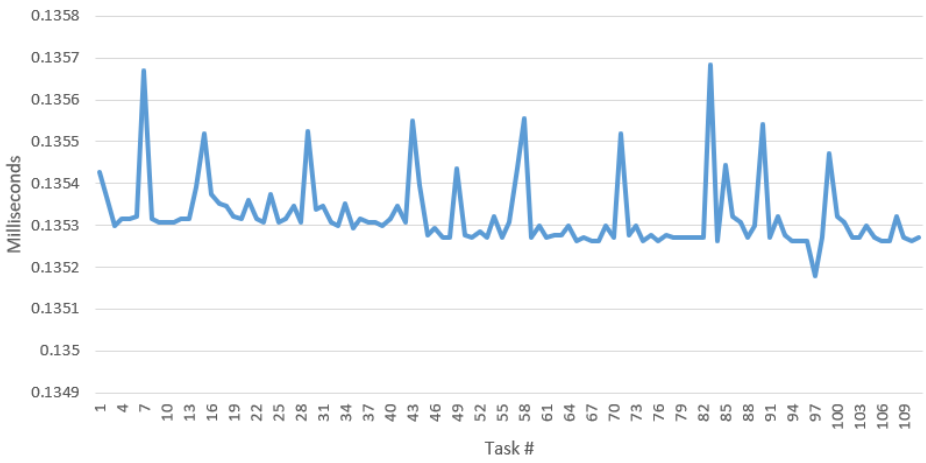

FIGURE 4.11: Random Address 8 Bins

TABLE 4.7: Random Hash Scenario Cost Analysis

	Random
	ms
Average	0.1353
Min	0.1352
Max	0.1361
Median	0.1353
StdDev	0.000100

4.11.1.5 Analysis

Figure 4.12 shows the aggregate results of the Identical, Sequential, and Random measurements. We observe that the identical measurements had the lowest costs compared to the other scenarios. We also see that this scenario has predictable cost spikes on the first Task of every Bin which indicates that the address does not fall into the CPU cache. However, the identical scenario is not a likely EPA-RIMM measurement scenario as EPA-RIMM would not be frequently measuring a single address repeatedly. The Sequential scenario does not benefit from CPU caching as the measured addresses vary. The prefetcher does also not appear to help as the average, min, max, and median times are equivalent to the random measurement cost, as shown in Table 4.8. In a typical EPA-RIMM measurement scenario, the most likely scenarios

are sequential or random, depending on the Backend Manager scheduling logic. This study quantifies the expected performance of these two scheduling approaches for memory measurements, showing that ultimately the amount of deviation is not significant for EPA-RIMM performance.

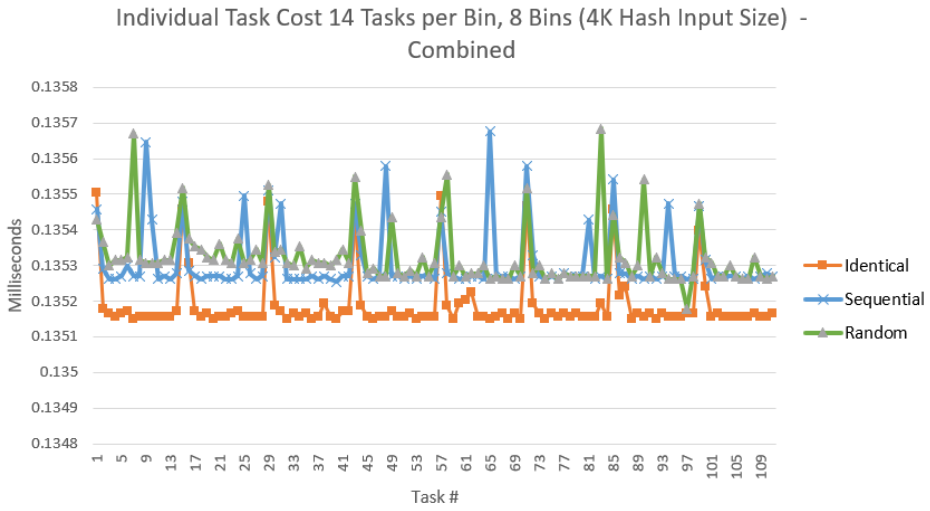

FIGURE 4.12: Combined 4K Hash Input Size Data, 8 Bins

TABLE 4.8: Cost analysis

	Identical	Sequential	Random
Average	0.1352	0.1353	0.1353
Min	0.1351	0.1352	0.1352
Max	0.1360	0.1361	0.1361
Median	0.1352	0.1353	0.1353
StdDev	0.000098	0.000088	0.000100

4.12 Conclusion

At the outset of this research, we identified a clear disconnect between the prolonged duration of SMI-based measurements from proposed SMM-RIMMs and the SMI latency guidelines. We believed that it may be possible to reduce the amount of time spent in SMM by applying decomposition to SMM-RIMM measurements to process them in fine-grained portions over a larger number of measurements. These smaller measurements could be scheduled at an

increased frequency. However, it was unclear what system impact that this shorter but more frequent measurement approach would have.

Moreover, there were no established methodologies to compare this alternate scheduling approach to the current state of the art. To resolve this challenge, we developed methodologies for generating SMIs of varying durations and frequencies to analyze their resulting impact. Our methodology is built upon three requirements: *Rquantify*, *Rcontrol*, and *Rvalidate*. Once able to quantify the time spent in SMM, we demonstrated controlling the amount of time spent in SMM (*Rcontrol*) by four SMI scheduling techniques. We then validated the time spent in SMM (*Rvalidate*) by comparing the quantified time in SMM against the expected degradation for two CPU-intensive workloads.

5
SMI Preemption Performance Study

In this chapter, we characterize the impacts of SMIs on the system by using our SMI measurement methodology. In our performance study, we varied the durations and frequencies of SMIs to show the resulting impacts from different SMI scheduling approaches. In Section 5.1, we examine the system impacts of this time spent in SMM and we cover the resulting impacts on applications in Section 5.2. We summarize the SMI latency study in Section 5.3.

5.1 System-level Effects

We begin in Section 5.1.1 by examining timing assumptions in the kernel and device drivers. We show the symptoms of spending an excessive amount of time in SMM in Section 5.1.2. We then examine the impacts of SMIs on timer interrupts and the impact on CPU power C-states in Section 5.1.3. We cover the impacts of SMIs on process accounting in Section 5.1.4. We summarize the system-level effects in Section 5.1.5.

5.1.1 Timing Expectations in Code

The Linux kernel source code contains assumptions about SMI durations in several places. For example, the function that calibrates the CPU's TSC during boot native_calibrate_tsc, uses the tsc_read_refs function which has special handling of SMI disturbances. tsc_read_refs checks two close reads of the CPU's timestamp counter to ensure that they are less than the declared SMI_THRESHOLD=50000 (CPU clocks) to avoid a scenario where an SMI occurs between the two reads. If the system cannot obtain two close reads of the TSC of a duration less than the SMI_THRESHOLD, it will try up to five times before returning [55]. Prolonged or inopportune SMIs could result in a situation where the TSC could not be used as the clocksource for timing due

to an inability to properly calibrate it. Other clocksource calibration sections of the Linux kernel feature similar concerns over the impact of an SMI hitting during calibration including functions pit_calibrate_tsc and hpet_next_event.

USB audio relies upon careful synchronization to keep the audio playback in sync. To better establish an upper bound for SMI durations, we developed a measurement to study the impact of prolonged SMIs on a USB audio speaker device. For this experiment, we used a pair of Logitech S-150 USB speakers and a system running Centos 6.0 with a Linux 3.7.1 kernel. We booted into the GUI and began playing a streaming audio file from YouTube. While playing the audio file, we generated progressively longer SMIs using our modified BIOS mechanism while checking the system log via the 'dmesg' command after each SMI completed.

We observed a number of warnings with SMIs up to the 1000 ms (1 second) range. At a SMI duration of 1000 ms, the audio completely stopped and the driver reported an error in the system log. Table 5.1 shows these results. The warnings we saw resulted from the snd_pcm_delay function which defines the playback delay as "the overall latency from the write call to the final DAC [94]." The code provides a warning when the delay estimate is off by more than 2 ms. In this measurement, we find another example of software with built-in timing assumptions that could be significantly altered by longer SMM-RIMM measurements. We also note that the SMI durations described in the HyperSentry and HyperCheck papers are in the range to cause warnings from the ALSA sound sub-system.

5.1.2 Symptoms of Excessive Time Spent in SMM

During development of our SMM-RIMM development system, we encountered situations where we spent too long in a single SMI session. For example,

TABLE 5.1: USB Audio Sensitivity to Prolonged SMI Delays

SMM time (ms)	**Warning**
1.43	ALSA sound/usb/pcm.c:1213 delay: estimated 144, actual 0
5-999	ALSA sound/usb/pcm.c:1213 delay: estimated [336 to 384], actual 0
1000	ALSA sound/usb/endpoint.c:391 cannot submit urb (err = -27)

with extensive serial output and a large hash operation, one of our preemptions inadvertently lasted 247 seconds. The Linux operating system appeared frozen during this duration, however, at the completion of the SMI, the system continued operating, albeit with several warnings and errors. These included: warnings from the Read Copy Update mechanism, warnings about an unstable clocksource, a hardware interrupt timeout, and a disk I/O error. Clearly this is far too long to spend in SMM, however, it is indicative of the types of errors that can occur if excessive time is spent in a single SMI session. It also demonstrates that even if a user is not using the system, exhaustive checking within a single SMI can easily overwhelm timeout expectations in the kernel and device drivers.

```
INFO: rcu_sched self-detected stall on CPU {
INFO: rcu_sched self-detected stall on CPU { 1}
0}  (t=61790 jiffies g=545203 c=545202 q=0)
(t=61790 jiffies g=545203 c=545202 q=0)
. . .
Clocksource tsc unstable (delta = 243722605708 ns)
mmc2: Timeout waiting for hardware interrupt.
. . .
mmcblk0: timed out sending r/w cmd command, card status 0x900
```

```
end\_request: I/O error, dev mmcblk0, sector 90111920
EXT4-fs warning (device mmcblk0p2): ext4_end_bio:317:
   I/O error -5 writing to inode 3019625 (offset 0
   size 0 starting block 11263991)
Buffer I/O error on device mmcblk0p2, logical block 11132662
Switched to clocksource acpi_pm
```

5.1.3 Timer Interrupt Effects

We originally began our investigation of the system effects of SMIs by focusing on the Linux timer interrupt. Our rationale was that this interrupt effectively drove a wide variety of system tasks ranging from process accounting to setting timers. We were very interested to examine the effects of SMIs on this critical kernel functionality as SMIs would take precedence over the timer interrupts.

5.1.3.1 Timer Interrupt Background

Traditionally many important scheduling and statistical operations in the Linux kernel happened on a regular timer tick interval, e.g. [100, 250, 300, 1000] times a second. For power savings reasons and reduced virtualization overheads, the "tickless kernel" option has been added, allowing the kernel to remain idle longer by avoiding unnecessary wake-ups. If the next scheduled timer event would occur after the next periodic timer tick, the kernel would reprogram "the per-CPU clock event device to this future event" allowing the CPU to remain idle longer [34]. In both traditional and tickless operation, our inspection of the Linux 3.1.4 kernel showed that once the kernel wakes, it runs several key functions in do_timer which update the kernel's internal clock count (jiffy) and wall clock time, and calculate the load on the system. (Refer to Figure 5.1.) Then it calls update_process_times which charges time

to executing processes, runs high resolution timers and raises SoftIRQs for local timers, checks if the system is in a quiet state for RCU callbacks, does printk statements, runs IRQ work, calls scheduler_tick and then runs timers that are due [72]. The scheduler_tick function performs several important tasks including updating scheduler timestamp data, updating timestamps for processes on the run queue, updating CPU load statistics based on the run queue, invoking the scheduler, updating performance events for the Linux Performance Event subsystem, determining if a CPU is idle at the clock tick, and load balancing tasks between CPU run queues. Intel technical documentation notes "All interrupts normally handled by the operating system are disabled upon entry into SMM [45]." This presents the possibility for an SMI to perturb timer interrupts and consequently impact the important scheduling operations in scheduler_tick as a side effect.

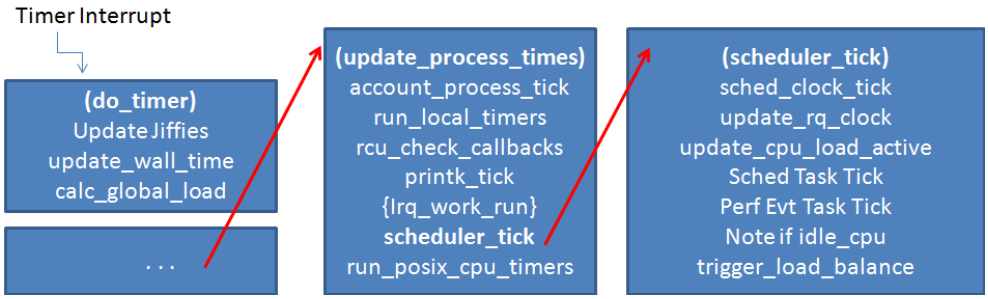

FIGURE 5.1: Timer Interrupt Code Flow

5.1.3.2 Kernel Instrumentation

To investigate the degree to which SMIs preempted timer interrupts, we instrumented the Linux kernel do_timer and scheduler_tick functions. For do_timer, we logged a trace point just after the timer interrupt occurs, recording the total number of SMIs processed ("SMI count" obtained via a CPU MSR read of MSR_SMI_COUNT) and the time of the entrance to the function from RDTSC. For scheduler_tick, we logged the CPU number, the SMI count, and

the timestamp from RDTSC. We extracted our traces from the kernel with the SystemTap utility [93]. In post-processing, we calculated the deltas between successive handlings of the timer ticks. Our regular timer tick scenario has a timer tick every millisecond. We generated SMIs using the chipset timer for the short but frequent scenarios and the Blackbox SMI method for the hybrid and long SMI scenarios. Our test system was an Intel DQ67SW board running native Centos and the 3.1.4 Linux kernel. Because the timer interrupt takes precedence over executing code, whether the CPU is idle or busy does not impact the regularity of the regular timer ticks. For this reason, we depict only the idle CPU data in this section. After establishing a baseline with no regular SMIs, we measure the effect of short but frequent SMIs using the Chipset-based SMI generation. Following this, we utilize a Blackbox SMI scenario of a batch of eight 5ms SMIs, once a second to represent an SMM-RIMM that takes 40ms per second to do integrity measurements using a time-sliced approach.

5.1.3.3 Timer Interrupt Results: Non-virtualized Linux

To analyze the data, we narrow our focus to the deltas between successive invocations of scheduler_tick to highlight SMI-caused delays. Numerous short SMIs cause jitter in the timer interrupt handling. Since SMIs take precedence over timer interrupts, the deltas between successive timer interrupts depart from the expected 1ms. Deltas greater than 1ms occur due to an SMI firing when a timer interrupt would have taken place. The delay in timer interrupt handling results in the greater than 1 ms delta, that in turn results in the next timer interrupt occurring after less than 1ms.

Table 5.2 shows a small sample of the jitter in the handling of timer interrupts. This effect eventually dissipates, but occurs again as the timer interrupt and SMI occurrences coincide. Even when regular SMIs are short, they can happen to occur at precisely when the timer interrupt fires, resulting in a

period of irregular timer interrupts for the short but frequent SMI scenario. Figure 5.2 depicts this effect.

For the Blackbox SMI scenario of a batch of eight 5ms SMIs a second, when the batch concludes, execution returns back to the operating system until another SMI occurs. In this scenario and a longer Blackbox SMI scenario with a 104ms SMI, the privileged software suffers significant portions of time where no forward progress can be made. These results (as shown in Figure 5.3) show that both long and short SMIs can preempt the timer interrupt with different patterns. The short but frequent scenario caused periods of jitter in timer interrupt handling. The long SMI scenarios showed that user and kernel tasks are completely frozen for extended periods of time and a number of timer ticks were missed.

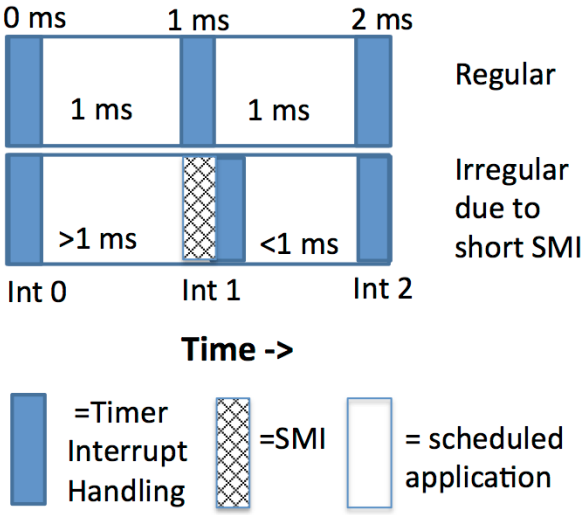

FIGURE 5.2: SMI Preemption of Timer Interrupt Handling

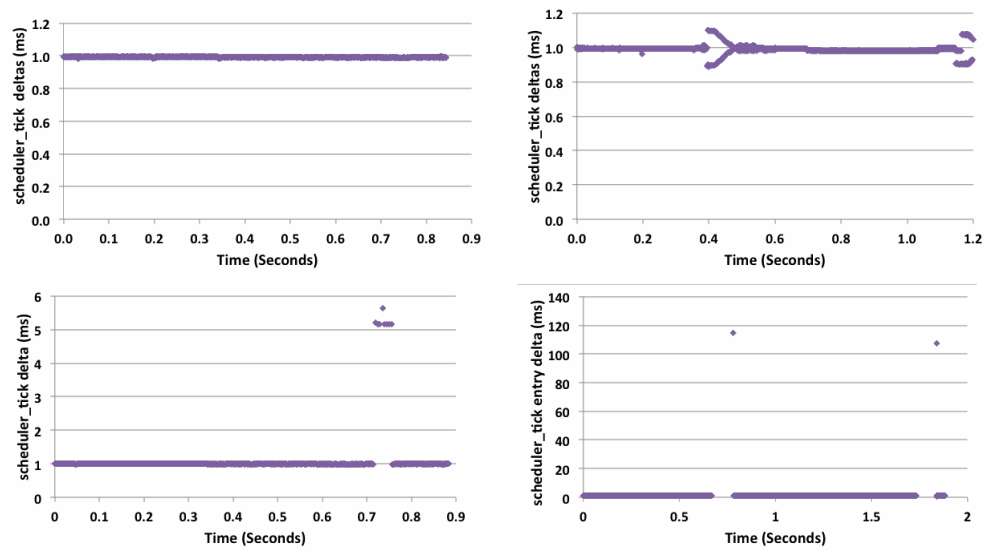

FIGURE 5.3: Native OS measurements with regular timer ticks and idle CPU
(a) (top left) Baseline (No SMIs)
(b) (top right) 0.11 ms SMI (500/sec)
(c) (bottom left) 5 ms SMI (8/sec)
(d) (bottom right) 104 ms SMI (1/sec)

TABLE 5.2: SMI Occurrences and Timer Interrupts

CPU#	SMI Count	Delta (ms)	Location	Notes
	39,089	0.00	do_timer	
0	39,090	0.08	scheduler_tick	
1	39,090	0.00	scheduler_tick	
	39,090	0.92	do_timer	**<1ms**
2	39,090	0.00	scheduler_tick	
0	39,090	0.00	scheduler_tick	
5	39,090	0.00	scheduler_tick	
7	39,090	0.00	scheduler_tick	
	39,091	1.07	do_timer	**>1ms**
6	39,091	0.00	scheduler_tick	
0	39,091	0.00	scheduler_tick	
7	39,091	0.00	scheduler_tick	
	39,091	0.92	do_timer	**<1ms**

5.1.3.4 Timer Interrupt Results: Xen Virtualization

To examine the effects of SMIs on timer interrupts in a virtualized environment, we repeated the measurements with a Xen HVM ("Hardware Virtualized Machine") Linux guest running under Xen 4.1.2. The results show that running a virtualized guest introduces a small degree of jitter in the regularity of the handling of timer interrupts (Figure 5.4(a), and adding SMIs perturbs the regularity further as shown in Figure 5.4(b)). For groups of long SMIs (e.g. groups of eight 5 ms SMIs), the guest can experience a significantly longer loss of control which coalesces multiple preemptions into one longer one (Figure 5.4(c) and 5.4(d)). For example, the Xen HVM guest experiences prolonged losses of control that exceed the 5 ms SMI in the range of 10 and 26 ms. We suspect that these increased delays are the effect of SMIs acting upon the virtual machine manager's scheduler which is resulting in the virtual machine not handling the interrupt for a longer period of time and amplifying the impact of longer SMIs in virtual environments.

5.1.3.5 Timer Interrupt and Turbostat Results: Tickless Linux Kernel

When the CPU is busy, the tickless kernel behaves like the regular timer tick, since no ticks are "skipped." During idle periods, however, the tickless kernel can experience large gaps between successive entries into the scheduler_tick function (e.g. up to around 200ms based on our measurements). Therefore, we focus here on the idle CPU case. We expect regular SMI activity to subvert the tickless kernel's energy savings, by waking up the CPU to enter SMM. To test this, we gathered data on the processor C-state utilizations using Turbostat [12]. Turbostat produces a log of what percentage of time the processor threads were in a given C-state. We started Turbostat, let the system

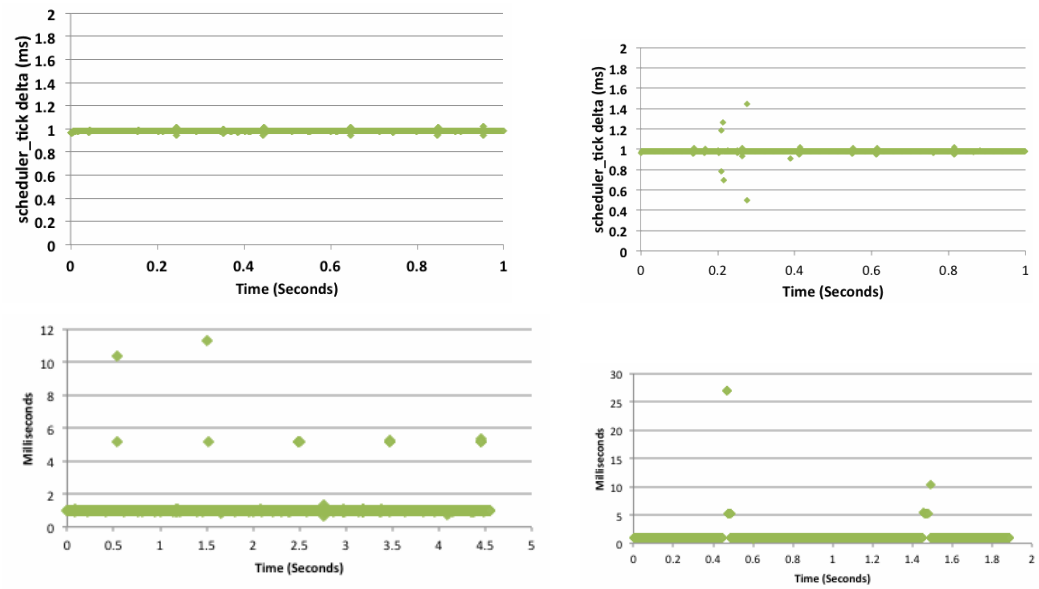

FIGURE 5.4: Virtualized measurements with regular timer ticks
(a) (top left) Baseline (No SMIs), Idle CPU
(b) (top right) 0.11 ms SMI (16/sec), Busy CPU
(c) (bottom left) 5 ms SMI (3/sec), Busy CPU
(d) (bottom right) 5 ms SMI (8/sec), Busy CPU

idle for several seconds, then enabled SMIs, waited a few seconds, disabled SMIs, and ended Turbostat.

In Figure 5.5(a), we show the baseline case for the tickless kernel without SMIs. The timing of the scheduler_tick entries varies widely as the kernel avoids unnecessary wake-ups to achieve power savings. Figure 5.5(b) shows the results for 500 SMIs/second. It is not readily apparent from the graph if a timer interrupt has been delayed or the kernel was simply idle for a long period of time. To look more closely, we must examine the raw trace data (see Table 5.3.) This shows that the kernel sleeps through the SMI activity as indicated by the increasing SMI count during long periods of kernel idleness. The tickless kernel adaptive timer mechanism is unaware of SMIs and while the kernel is idle, the CPUs transition in and out of SMM processing SMIs.

Fortunately, the Linux kernel (since version 2.6.19) has a mechanism to avoid missing jiffy updates due to lost timer ticks by determining how many

timer ticks were missed (ticks) and incrementing the jiffy count accordingly in do_timer. Without such a mechanism, jiffy updates would be lost. The results of our instrumentation (Table 5.4) show that the do_timer function increments the ticks value after receiving control following an SMI. When an SMI preempts the kernel for five ms, the kernel determines that five timer ticks were missed and sets the ticks value accordingly and adds that value to the jiffies count. When our group of eight SMIs concludes, our instrumentation shows the SMI count staying steady and the ticks value returning to one as the SMIs subside. The kernel remained idle through the SMIs, however the CPU was actively processing SMIs. If we limited our analysis to our kernel instrumentation, we would miss a large amount of activity on the system. The kernel instrumentation correctly indicates that there were long periods of idle in the kernel which traditionally would correlate to the CPUs ability to transition into deeper sleep states. However, with SMM-RIMMs, regular SMIs are also occurring which would keep the CPU active. Figures 5.6 and 5.7 show that SMIs bring the CPU out of the lowest power C6 state and into the higher power-consuming C0 and C1 states. The short but frequent scenario results in more time spent in higher power C-states than the hybrid scenario that has longer SMIs.

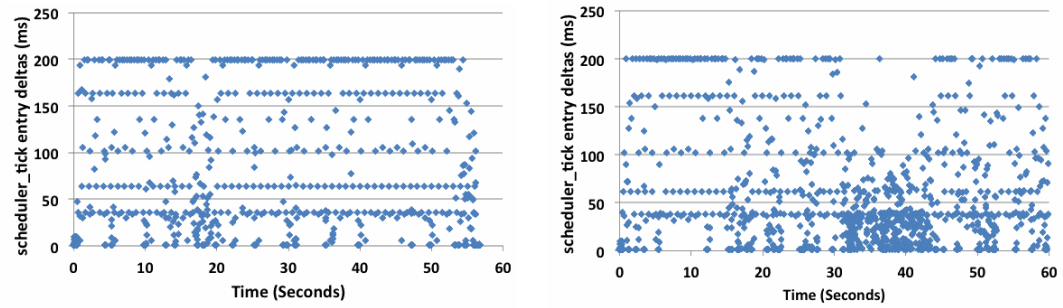

FIGURE 5.5: Virtualized measurements with tickless kernel, idle CPU
(a) (left) Baseline (No SMIs)
(b) (right) 0.11 ms SMI (500/sec)

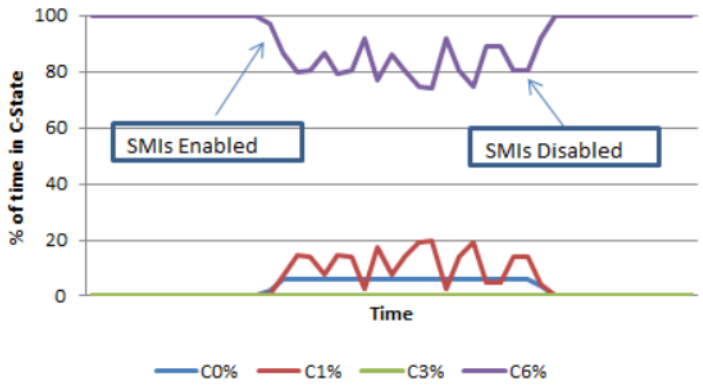

FIGURE 5.6: 500x0.11ms SMIs/second

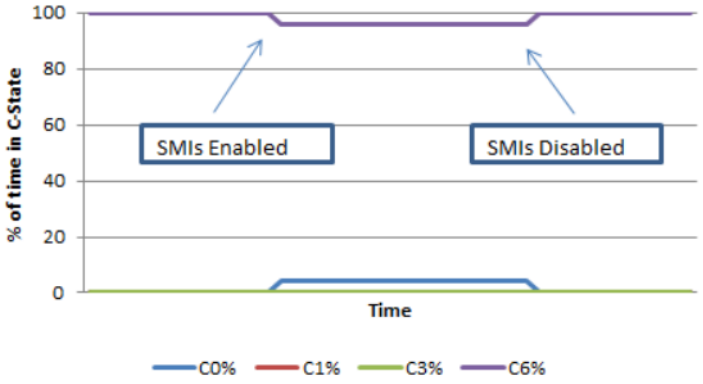

FIGURE 5.7: 8x5ms SMIs/second

TABLE 5.3: Tickless Kernel and 500 SMIs/second

Trace Point#	SMI Count	scheduler_tick delta (ms)
0	23,351	40
1	23,382	62
2	23,433	102

5.1.3.6 Timer Tick Conclusions

Our examination of disruptions to the regularity of the scheduler_tick shows several important effects. In some cases with short but frequent scheduling, SMIs can resonate with the timer interrupt resulting in extended periods of time where the timer interrupt handling may occur late relative to a regular time tick. This may result in timer interrupt handlers closer together or

TABLE 5.4: do_Timer Ticks Mechanism, the trace after the ellipsis begins to recover from the batch of SMis

Trace Point#	SMI Count	Ticks	Delta (ms)
0	19,082	1	1.00
1	19,083	5	5.21
2	19,084	5	5.15
...	...	...	...
3	19,090	1	0.63
4	19,090	1	1.00

further apart than traditionally done. Additionally, with the longer SMI scheduling option, SMIs that exceed the length of the timer interrupt will cause timer interrupts to be missed. However, the kernel can keep its internal jiffy count accurate. With long SMIs, there can be long periods of time between entries into the process scheduling function. Virtualized environments may experience longer delays as multiple shorter delays coalesce into longer delays. Applications may experience longer wait times since the OS scheduler cannot run.

In the case of an idle tickless kernel, determining if a timer interrupt was delayed or lost due to an SMI is not as straightforward. Our results show that the kernel remained idle while SMIs were occurring which is expected since the kernel is unaware of the loss of control due to SMIs. The C-state analysis showed that while the kernel was idle, the CPU's power utilization was affected by the SMI activity. The short but frequent SMI scheduling scenario resulted in the CPU running in higher power C-states due to frequent wakeups from SMIs that circumvent the power-saving processor modes.

5.1.4 Process Accounting

In the course of our OpenSSL measurements on a system using the 2.6.32 Linux kernel, we noticed an unexpected phenomenon: When we increased the duration of the SMIs using our modified BIOS, the reported throughput

did not decrease correspondingly but instead remained constant as shown in Figure 5.8. Achieving the same performance regardless of whether SMIs were occurring was not a reasonable result. We also noticed that OpenSSL's reported computation time decreased as we generated longer SMIs. Analyzing the method of calculating the workload throughput showed that it reported throughput in bytes processed per second by determining the number of computations performed and the length of time required. The OpenSSL benchmark set up a signal (SIGALRM) for three seconds in the future and performed computations until the signal arrived. When we ran the workload with a 2.6.32 kernel and 100 ms SMIs, the kernel reported that the computations took 2.74 seconds. When no SMIs occurred, the kernel reported the time as 3 seconds, although the reported throughput paradoxically remained constant to the non-SMI case.

Our measurements using a more recent kernel (3.7.6) showed different behavior that closely matched our calculations over the amount of expected throughput decline. This configuration reported that the OpenSSL benchmark was computing for approximately the full 3 seconds both when 100 ms SMIs were enabled and when they were disabled and showed degraded throughput with SMIs. Figure 5.9 depicts the scaling of times billed to the application for varying durations of SMIs for the two kernel versions.

To root cause the discrepancy, we instrumented the two Linux kernels to log the flow of time-keeping data used by the times and accompanying functions. We also added a trace point in the OpenSSL application to capture the time when the signal handler function was called in the application and two trace points before and after the computations began in the benchmark. Our examination of OpenSSL showed that it used the Linux kernel's times function which reports the amount of user time, system time, child user time,

child system time used by a process.

We started SystemTap [93] to monitor key variables in the kernel functions responsible for the reported process time statistics:

1. do_sys_times
2. thread_group_times
3. thread_group_cputime
4. task_sched_runtime
5. do_task_delta_exec
6. scale_utime

We then started an OpenSSL benchmark run using "openssl speed sha512", with SystemTap. This test allowed us to compare the reported amount of time billed to the process with the trace points gathered in the application using the CPU's TSC.

The results show that the SIGALRM signal was received after three seconds in both kernel versions. For the 2.6.32 kernel, this highlighted the discrepancy between the amount of application time as measured by the TSC and the kernel. One kernel function explained the discrepancy: scale_utime. This is used to reduce over or under-counting of user or system time due to the point in time when the user or system task was actually interrupted. The code scales the operating system timer tick-based values against the scheduler's record of total runtime. With a 100 ms SMI per second, this function increased the billed user time by 10% which compensated for the loss of time spent in SMM, while for the no SMI scenario, the user time was not scaled accordingly. Table 5.5 shows the scaling calculation for the first three second OpenSSL measurement. As the 2.6.32 kernel did not call this function in the

do_sys_times code path, the user time was not adjusted to include the 10% of time spent in SMM leaving the billed process times lower in the 2.6.32 kernel.

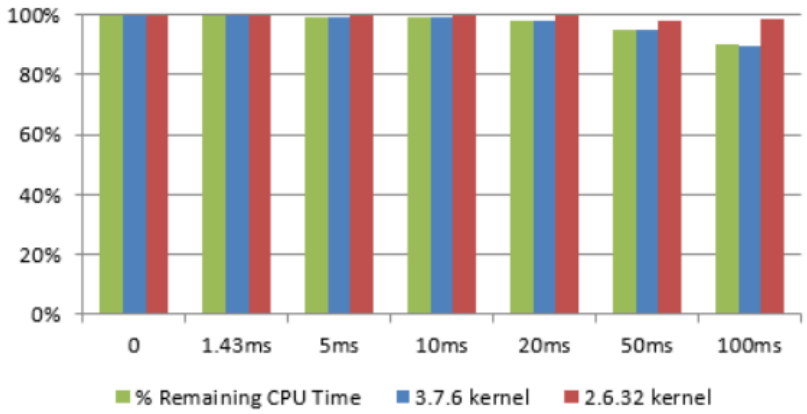

FIGURE 5.8: Throughput Scaling

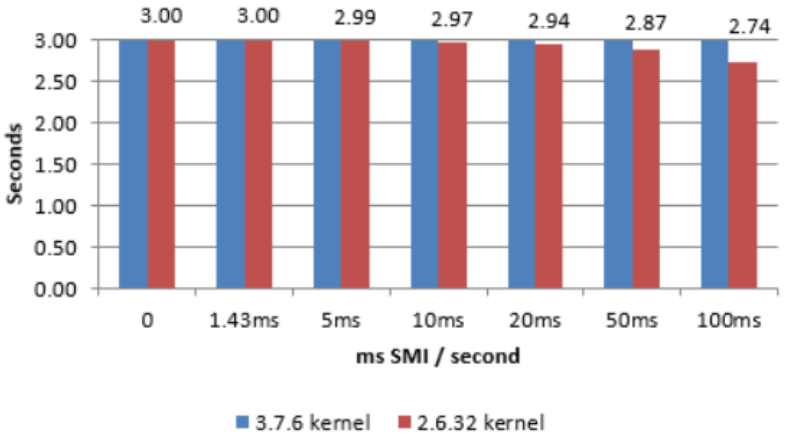

FIGURE 5.9: Billed Seconds

5.1.5 System-Level Effects Summary

Our results raise the question of how the operating system should account for process times when there is prolonged SMI activity on the system: include any SMI times with the billed process time using the times mechanism, or leave this time out of the billed amount? There are benefits and drawbacks to both approaches. Reporting time inclusive of SMI times has the drawback of charging applications for time spent outside of their process which could

TABLE 5.5: User Time Scaling In Scale_utime 3.7.6 kernel

Variable	**Notes**	**Scaling No SMIs**	**Scaling 100 ms SMI 1 per sec**
utime	Unscaled User Time	3,003	2,708
rtime	Scheduler's sum_exec_time	3,008	3,002
total	User + System	3,009	2,709
[Scaled user time]	(rtime * utime) / total	3,002	3,000

penalize some applications more than others depending on when the SMIs happened to occur. In our study, this resulted in all three seconds being attributed via the times mechanism to a process without discarding the portion of the time spent in SMM. The exclusion of SMI times in process time accounting can lead to discrepancies as well. In the case of OpenSSL running on the 2.6.32 kernel, the workload concluded after three seconds based on the CPU's TSC, however the process only believed that it had used 2.74 seconds when 100 ms SMIs were active. When SMIs were infrequent and had short durations, their effect on process accounting could essentially be overlooked. For environments that are sensitive to accurate billing of time to users such as cloud providers, new mechanisms are required to more accurately account for the amount of time consumed by long SMIs. However, resolving the fundamental issues in process time accounting will require kernel changes and possible SMM-RIMM involvement.

Our analysis of system level SMM effects shows several negative impacts from prolonged SMM time. While certain sections of the Linux kernel have special handling for SMI occurrences, other sections could have differing behavior upon experiencing prolonged SMI durations. Software advances such as tickless kernels, while implemented for other reasons, increase the

tolerance of SMM preemptions. Our detailed results demonstrate that systems can spend longer in SMM than current guidelines, however, there are problems that arise at durations below those contemplated for SMM-RIMMs. We showed that SMIs cause periods of timer interrupt jitter in the short but frequent scenario and extended periods of delays for the longer SMI scenarios. These impacts delay handling of timer interrupts and postpone work on all cores until the SMI completes. Additionally, in an environment where power savings are of increasing importance, SMM-RIMMs would bring a reduction in the amount of time CPUs can remain in low power states. In an extreme SMI preemption, we showed a device driver that failed because it interpreted the delay as unresponsive hardware.

5.2 Application Effects

Because even slight delays can have a perceptible impact on applications, we designed a study to investigate the impact of prolonged SMIs on several types of workloads. The correlation of application and noise granularity [10] is quite relevant to the SMI-based perturbation investigation as SMIs could be long or short, frequent or infrequent, occur regularly or irregularly.

5.2.1 Kernel Compilation

Linux kernel compilation involves several key aspects of platform performance including CPU operations, disk I/O, and memory accesses. We used Xen 4.2.1 with a Centos 6.3 Domain 0 and a Centos 6.3 HVM guest with one virtual CPU and two GB of RAM. The results in Figure 5.10 show increases in total compilation time that very closely match the level of SMM preemption. For example, taking 10% of the CPU cycles away (100 ms SMI scenario) resulted in a 10.8% increase in the duration of the kernel compilation.

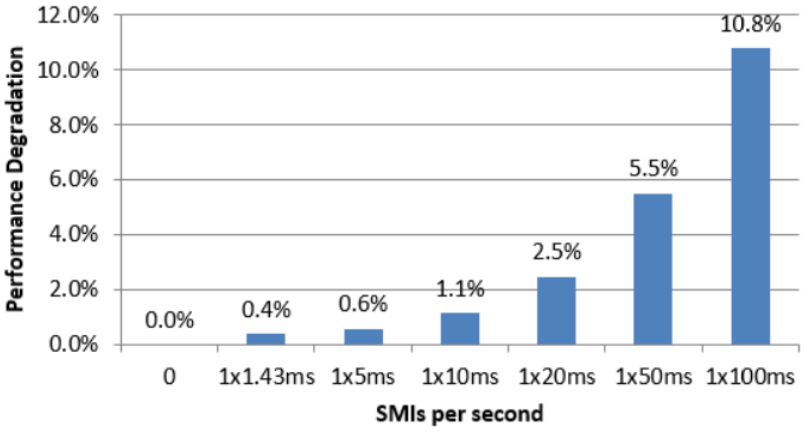

FIGURE 5.10: Kernel Compilation Performance for Linux/Xen

5.2.2 Microbenchmarks

To examine system performance impacts on a broader set of system activities, we ran two sets of benchmarks, one for Xen's Domain 0 (Xen 4.1.2) and one for a Centos 6.0 hardware-virtualized guest. We compared how throughput scaled against the baseline for varying levels of SMIs using our modified BIOS technique. For our workloads, we used RC5-72 [95], a compute-intensive workload that brute-forces cryptographic keys (tested on Domain 0 only); Netperf 2.5 for TCP transmit [85] using a gigabit Ethernet device; and XDD for 128KB sequential disk reads using an Intel X25-M SSD [115].

The left-most bar in Figure 5.11, Figure 5.12, and Figure 5.13 shows the percentage of CPU time available to the system after subtracting the amount of time spent in SMM per second. The individual benchmarks all experienced throughput degradations that closely match the amount of CPU cycles taken away for SMIs. With these workloads and long SMIs, SMM latency cannot be hidden by the application as it comes at the cost of performing I/O operations or computations.

By comparison, short but frequent SMI scheduling can maintain baseline throughput for some workloads even as the amount of available CPU time

decreases. SMI usages that are able to interleave SMIs with I/O processing may be able to avoid the full penalty of the SMI by processing their SMM work in multiple smaller units. These results indicate that a time-sliced SMM-RIMM can allow workloads that do not exclusively perform computations to avoid a portion of the performance degradation that would otherwise occur for (non-heavy compute) applications.

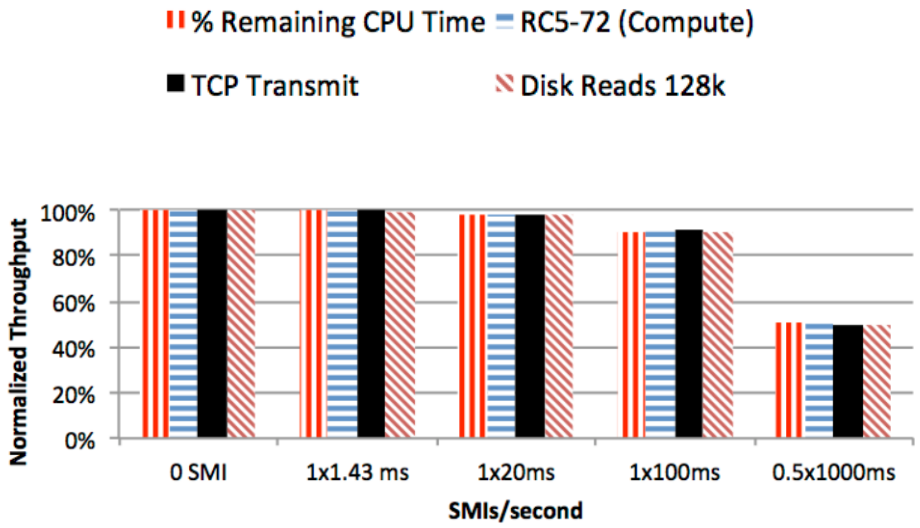

FIGURE 5.11: Xen Dom0, Long SMIs

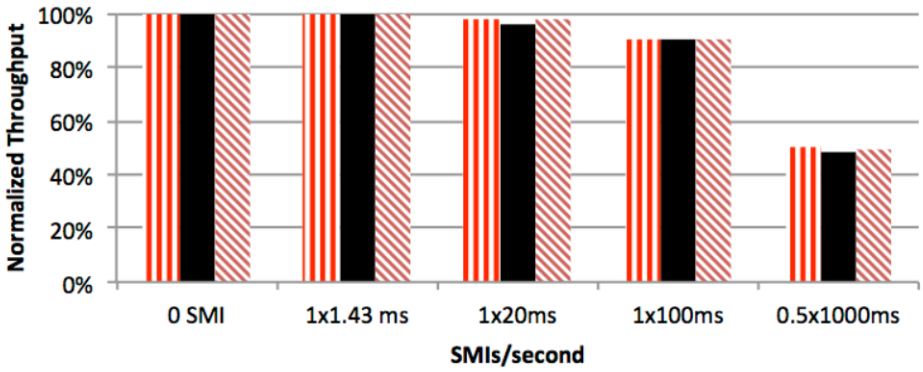

FIGURE 5.12: Xen VT Guest I/O, Long SMIs

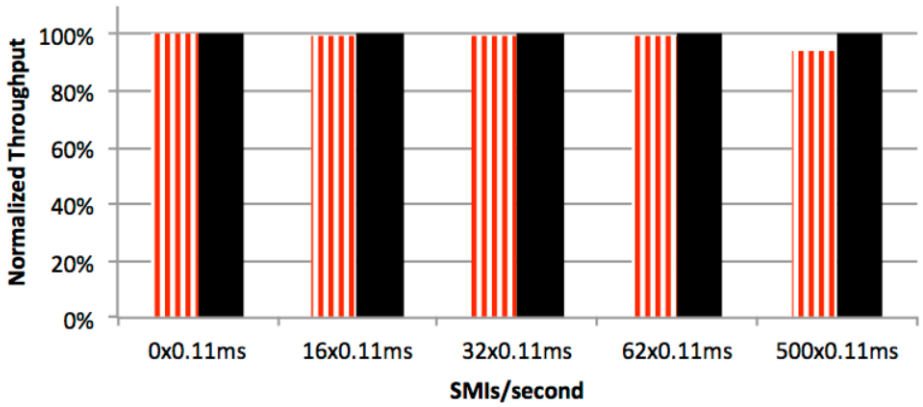

FIGURE 5.13: Xen VT Guest Short SMIs

5.2.3 Latency-sensitive Application

As the USB testing indicated, latency-sensitive applications can be problematic for SMM-RIMMs. To investigate this further we used Windows Server 2012 and the Unreal Tournament 3 benchmark utility (UTBench) to measure game frame rates. We used our modified BIOS technique for these tests and show the results in Table 5.6.

Although the average frame rates were above 50 fps for all durations but the 495 ms SMI, at SMI durations of 20 ms and higher the frame rates dipped below 30 frames per second, which is in the range of the user's perception. The finer-grained analysis shows that 20 ms delays only dropped below 30 frames per second 0.92% of the time which we did not notice subjectively however at 50ms delays, the system achieved below 30 frames per second 5.99% of the time which was visually apparent. This latency-sensitive application showed clear sensitivities between the 20 and 50 ms SMI durations.

TABLE 5.6: Unreal Tournament 3 Frame Rate Binning

MS SMI	0-5	5-10	10 - 15	15 - 20	20 - 25	25 - 30	30 - 35	35 - 40	40 - 45	45 - 50	50 - 55	55 - 60	60+
0	0	0	0	0	0	0	0	0	0	0	0	0	100
1.43	0	0	0	0	0	0	0	0	0.01	0	0	0.08	99.91
5	0	0	0	0	0	0	0	0	0	0.09	0.16	0.15	99.60
10	0	0	0	0	0	0	0	0.07	0.87	0.12	0.14	0.02	98.79
20	0	0	0	0	0	**0.92**	1.59	0.37	0.17	0.05	0	0	96.90
50	0	0	0	**5.99**	0	0	0	0	0	0	0.01	0.01	94.00
100	0	**10.83**	0	0	0	0	0	0	0	0	0	0	89.17
495	**50.24**	0	0.04	0.03	0.02	0.02	0	0	0	0	0.01	0	49.64

5.2.4 Application Conclusions

Our results show that application level impacts from SMM time vary based on the characteristics of the application as well as SMI scheduling. Some usages (e.g. compilation) experience degraded throughput while others such as Unreal Tournament and audio playback, are particularly sensitive to long duration SMIs as the user experience is severely degraded. Some applications (like TCP transmit in Figure 5.13) are able to hide a portion of the SMI delays as these applications would still need to wait for other operations to complete. We note that the latency sensitive applications we examined suffered user-perceptible impacts at some of the SMI durations proposed for SMM-RIMMs.

The measurement study provided insights into whether available head-room exists to allow increasing the SMI delays slightly above the latency guideline. The ability to support modest increases in SMI duration over the guideline forms a useful tuning knob to allow additional measurement or longer measurements to be performed under time of attack. This approach is analogous to the turbo feature on modern x86 CPUs that allows one or more CPU cores to briefly run at a higher CPU frequency to accomplish more work while available thermal budget exists [66]. This analysis provides a measurement-baseline for an empirical bound on SMI latency, that provides

a practical upper bound on SMI latency in which no significant issues have been observed. We term this bound, $\textbf{LimitSmi}_{\textbf{Empirical}}$ in contrast to the BITS guideline [106] which we term $\textbf{LimitSMI}_{\textbf{BITS}}$.

5.3 Conclusions

This measurement study gave us the initial performance feasibility analysis to better understand the tradeoffs between single long SMI preemptions utilized by contemporary SMM-RIMMs and the time-sliced approach that we propose. Figure 5.14 summarizes the results. The use of SMM-RIMMs causes host software and applications to experience unexpected preemptions, however, our performance measurements show that keeping these preemptions to a shorter degree can greatly reduce negative impacts. For example, this can avoid breaking timing assumptions in the operating system that impact correctness and reduce the impact on latency-sensitive applications.

There are also signs that operating systems can be more tolerant to lost timer ticks than in previous years as the Linux kernel can recover from the effects of lost ticks. Our results suggest that RIMM developers should carefully consider how system preemptions should be accounted for, particularly in cloud environments where users pay a financial cost for CPU time used.

Impacts on power utilization will depend on the level of CPU activity. If the CPUs are idle, measurement SMIs would bring the CPUs out of more idle states. However, on active servers, the CPUs will likely already be in less idle modes which should not significantly impact the consumption.

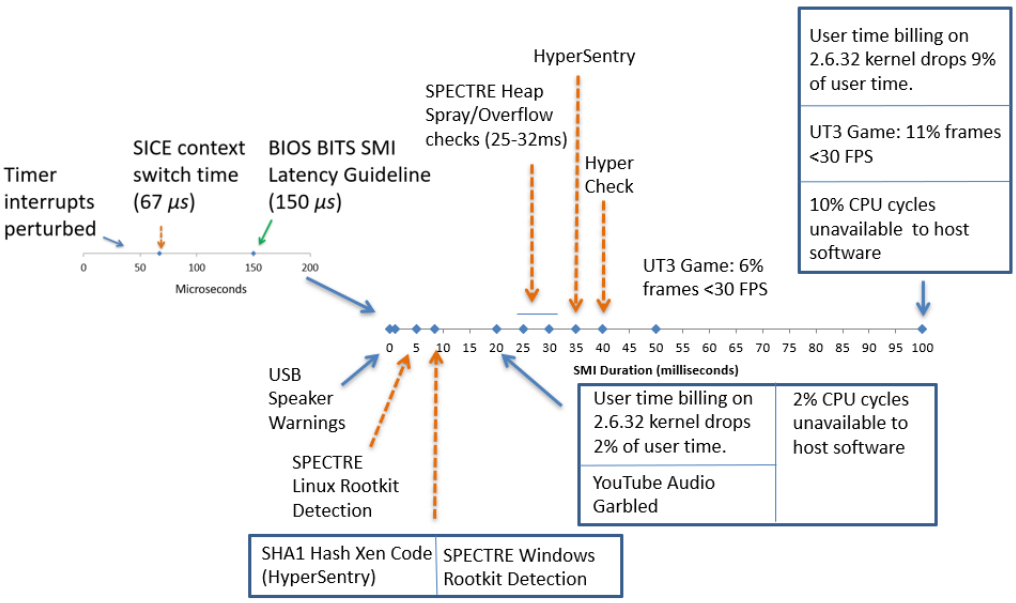

FIGURE 5.14: SMI Performance Impact Summary

6
EPA-RIMM Design Requirements

As we research building a performance-aware and extensible SMM-RIMM, we constrain our approach based on eight design requirements. An early SMM-RIMM, HyperSentry, provided five key design requirements for their approach addressing protection of the measurement agent and its results. These requirements [7] (Stealthy Invocation, Verifiable Behavior, Deterministic Execution, In-context Privileged Measurement, and Attestable Output) represent a useful starting foundation for SMM-RIMMs, however, we identify three key gaps: Extensible measurements, performance efficiency, and the need to constrain the measurement agent. The lack of extensible measurements reduces the overall effectiveness of the SMM-RIMM approach as attackers could readily shift their tactics to compromising unmonitored resources. It also could result in the SMM-RIMM being unable to respond to new attacks which significantly reduces the value of the approach. For the performance efficiency gap, there are several implications of a lack of focus on the amount of work to be taken in SMM. First, dramatically exceeding expectations over time spent in SMM can result in system stability and correctness issues based on our performance studies [23], rendering the approach ill-suited for practical use. Second, the set of integrity measurements to be performed may grow over time in response to new threats and accomplishing all measurements in a single SMM session quickly becomes infeasible. In recent years, concerns over the power of SMM have grown and efforts arisen to constrain its resource access. Thus, an SMM-RIMM should now be constructed with the principle of least privilege. This new requirement prescribes providing the measurement agent with the minimum access to perform its inspections without growing the agent's access unnecessarily.

EPA-RIMM supports the five original design requirements with modifications due to architectural differences (Sections 6.1 through Section 6.5) and contributes three new requirements addressing extensibility, performance, and a constrained measurement agent in Section 6.6, Section 6.7, and Section 6.8. We provide our design requirement conclusions in Section 6.9.

6.1 Requirement 1 - Stealthy Invocation

SMM-RIMMs endeavor to catch malicious code by surprise. Thus, a key design requirement is that an attacker in the monitored operating system or hypervisor should not be able to detect that a measurement is about to take place. If the measurements of an SMM-RIMM were to be detected by a compromised hypervisor, it could be possible for malicious code in the hypervisor to hide attack traces before an integrity measurement occurred. HyperSentry implements a "Stealthy Invocation" mechanism to cover this requirement [7]. Utilizing an SMI to trigger a measurement can aid the element of surprise, particularly if the SMI generation is done from hardware (e.g. chipset or out-of-band controller) without awareness of host software. However, care still must be taken that malicious software is not able to derive when a measurement is scheduled to occur or observe indications that an SMI is pending, as discussed in Section 3.3.7. In general, SMM aids the stealthy invocation requirement as control transfers to SMM can be accomplished via asynchronous methods (e.g. hardware-generated SMIs) that do not require the host CPU to trigger the measurement.

6.2 Requirement 2 - Verifiable Behavior

The verifiable behavior requirement prescribes that the "code base of the measurement agent, along with its input data, should be measured and verified before it is invoked [7]." As HyperSentry utilizes a measurement agent

in the hypervisor to gain the benefit of deeper insight into the hypervisor state, the measurement agent is at a significant degree of risk as it resides in the same privilege level as the code that it is monitoring. Thus, HyperSentry's agent integrity must be verified before it is invoked to ensure that it has not been compromised. Here, HyperSentry and EPA-RIMM differ in their trust assumptions. HyperSentry places their measurement agent in an untrusted location, e.g. a potentially malicious hypervisor to simplify the task of getting insights into the currently operating hypervisor state. EPA-RIMM places the measurement agent within a hardware-protected region, SMRAM, to better armor the measurement agent and uses a provisioning phase to direct the runtime measurements. While SMRAM has been compromised before [53, 14, 15, 20], it has a higher bar to compromise than hypervisor code due to additional hardware protections and higher attack complexity.

6.3 Requirement 3 - Deterministic Execution

HyperSentry defines a "Deterministic Execution" requirement in which the measurement agent should not be "changeable nor interruptible" after it is invoked [7]. This property is supported by the use of SMM as its processing is not interruptible and hardware protections such as BIOS flash protections and SMM Range Registers (SMRR) protect SMM code from being changed. HyperSentry measurements have a two-step process in which entry into trusted SMM code first occurs, the measurement agent in the hypervisor is measured (e.g. as described in Section 3.3.7) and then control transfers to the measurement agent in an uninterruptible manner. Contemporary computer systems support varying degrees of privilege as previously shown in Figure 6.1 with SMM as the most privileged level compared to hypervisors,

operating system kernels, and applications. This provides a "strong isolation [5]" between the RIMM and the monitored environment which prevents changing or interrupting the measurement agent. Having strong isolation is beneficial from the security perspective but poses challenges for the ability of the RIMM to understand the internal workings of the less-privileged layer that it is monitoring.

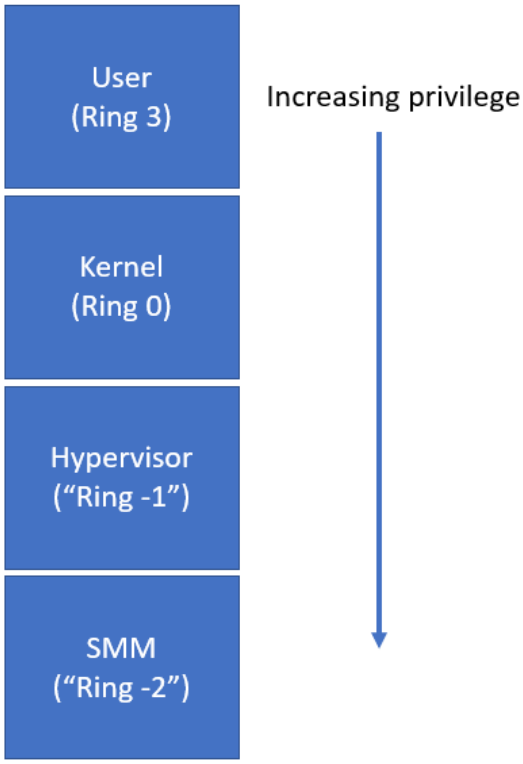

FIGURE 6.1: x86 Privilege Levels

6.4 Requirement 4 - In-Context Privileged Measurement

HyperSentry notes that for In-Context Privileged Measurement that the measurement agent should be privileged "and in the right context to access the hypervisor's code and data, and to gain full access to the CPU's state [7]." While an SMM-RIMM has the privilege to look deeply within the monitored

environment, the challenge becomes how to interpret what it observes, as SMM does not natively understand internal operating system or hypervisor data structures. SMM would not know whether the host software is Windows 8, Windows 10, Linux, or Xen. Compounding the challenge, operating systems or hypervisors utilize internal data structures that differ from each other and vary between revisions. For example, Xen features a "Grant Table" data structure that controls which memory pages a virtual machine is allowed to map. As this data structure is Xen-specific, SMM is not aware of its existence. SMM is also not aware that Xen supports two different versions of the Grant Table data structure [97] but would need to understand the layout of the appropriate version in order to properly measure it.

Previous SMM-RIMMs have dealt with this challenge in several different ways. SPECTRE built in operating system-specific knowledge into their SMM measurement agent [119]. To identify physical memory addresses for SMM to measure, SPECTRE utilized an observation that in Linux and Windows, virtual addresses above addresses 0xC0000000 and 0x80000000, respectively are considered kernel space. To find the physical memory address, an offset is subtracted from the virtual address. SPECTRE employs a similar method to find the location of the Kernel Processor Control Region (KPCR) which controls a sequence of data structures that control processes and threads in order to measure it. The authors note that this data structure resides at virtual address 0xFFDFF000 and rely upon known memory offsets to parse the data structure.

The issue with this approach is that it assumes a kernel data structure layout that can vary depending on version. Also kernel address space randomization (KASLR) breaks these assumptions as code can be located at different addresses each boot. Thus, this method of hard-coding addresses

and offsets is brittle and would require BIOS updates to update SPECTRE's assumptions. HyperSentry resolved this tension by placing their measurement agent within the hypervisor. This allows the agent to directly leverage hypervisor data structures and layouts in its measurements. EPA-RIMM keeps the measurement agent within SMM and utilizes information gleaned from a provisioning phase to direct the measurements. Changes in layout and offsets can be accounted for in the data structures that the EPA-RIMM measurement agent receives.

6.5 Requirement 5 - Attestable Output

It is important that the output of an SMM-RIMM be trustable. If the SMM-RIMM were to output a result but host software were to tamper with the output, attacks would go unnoticed. For this reason, it is essential for an SMM-RIMM to output its results such that they can be verified at the receiving end. This includes using signing, encryption, and message authentication checks. These mechanisms would provide the ability to detect malicious tampering with in-transit results. Similarly, if malicious host software were to prevent the SMM-RIMM from outputting any data at all, this could be detectable by the RIMM recipient not receiving any of the expected reports.

HyperSentry builds trust into the RIMM's output by leveraging a public and private key pair to be used in verifying that the output was not forged. This mechanism places the private key in SMM's protected SMRAM region and extends the public key to a PCR (register) on the system's TPM. Before transmitting its results, the SMM-RIMM signs the measurements with its private key. A remote user can choose whether to accept the results based on whether the data was signed by HyperSentry's private key. Additionally, the use of a nonce can guarantee freshness of results to avoid replay attacks

in which a previous valid but stale measurement is provided. With this mechanism, assurance over the integrity of the data can be provided by the RIMM.

6.6 New Requirement 6 - Extensible Measurements

Current SMM-RIMMs lack the ability to vary the amount of runtime checking based on performance needs or changes in the current threat environment which reduces effectiveness. Thus, we propose a new requirement for SMM-RIMMs prescribing the need for extensible measurements. Providing a programmatic API allows a new set of measurements to be pushed out to the SMM-RIMMs as new checks are designed. This capability allows the RIMM's checks to be updated as often as desired without needing to replace firmware code, allowing for a more adaptive and effective mechanism. The benefit of SMM-RIMMs can be greatly enhanced with communication between the SMM-RIMM instances. When attacks are detected on one SMM-RIMM instance, this information can be shared with other participating SMM-RIMMs to help prevent attacks from spreading. It can also enable more targeted inspections by focusing SMM-RIMMs on platform resources that have been compromised on one of the instances.

6.7 New Requirement 7 - Performance-aware

An ideal RIMM design would provide quick detection of attacks at a performance overhead that is imperceptible to the user. The more quickly an attack is detected, the more useful the RIMM mechanism is. The intensity of RIMM inspections has two key variables: measurement frequency and duration. A very frequent scheduling of a RIMM could have a low or a high system impact depending on measurement duration. Similarly, an inspection with a longer duration may be less impactful to the user if it were to only run once a

day. Thus, these two parameters provide tuning knobs that can be calibrated based on differing tolerances. There are practical upper limits, though, to how long the CPU threads can remain in SMM in a single session as some software has specific timing expectations which can impact correctness. Significant performance impacts can occur with prolonged check durations. Therefore, it becomes infeasible to consider approaches that conduct extensive checks within one atomic measurement session and greatly exceed the established SMI latency guidelines.

Given this limitation, it is necessary to consider time-slicing RIMM operations such that they can occur more frequently and consume less time in a single RIMM measurement session. This time-sliced approach is the most promising mechanism that can accomplish complex integrity measurements over a longer period of time with fewer negative impacts.

6.8 New Requirement 8 - Constrained Measurement Agent

SMM code has traditionally been granted access to all of memory and registers. SMM-RIMMs have adopted this design model and had access to all system state to perform their measurements. However, in response to growing concerns over attacks on SMM, efforts have arisen to reduce the accesses of SMM code on contemporary platforms. The Intel STM provides one such mechanism that allows policies to be created to remove a set of accesses from the SMI handler. Given current industry trends, we propose the use of a constrained measurement agent as a requirement for SMM-RIMMs.

6.9 Conclusions

SMM-RIMMs were originally formulated in a time when awareness of the performance impacts of SMM were not understood and granting SMM full access to the system's resources was common-place. Additionally, the concept

of extending the set of measured resources was not conceived. Our design requirements address these key limitations to enable SMM-RIMMs in the contemporary environments, while supporting the previously formulated requirements of stealthy invocations, verifiable behavior, deterministic execution, In-Content privileged measurement, and attestable output. We add three new requirements: 1. Extensible measurements to better vary the set of monitored resources during inspections, 2. Performance-awareness to balance the need for checking with system performance constraints, 3. Constrained measurement agent to reduce the risk of an SMM-based measurement agent compromising the system.

7
Architecture

In this section, we provide an overview of EPA-RIMM's architecture. There are three key abstractions in EPA-RIMM: Checks, Tasks, and Bins. A Check (Section 7.1) is a description of an integrity measurement with parameters to guide its execution. If a Check detects an unacceptable result (e.g. a changed result in a presumed static resource), EPA-RIMM generates an alert. At runtime, Checks are decomposed into partial resource measurements called Tasks (Section 7.2), to meet expectations for SMI latency. Tasks are scheduled in Bins (Section 7.3) to bound the work performed in one SMI session. Bin size is defined as the sum of the execution times of the Tasks it contains.

The EPA-RIMM software architecture comprises the Diagnosis Manager (Section 7.4), the Backend Manager (Section 7.5), the Oracle (Section 7.6), the Host Communications Manager (Section 7.7) and the Inspector (Section 7.8). Section 7.9 provides our security analysis of the EPA-RIMM architecture. We provide conclusions on the EPA-RIMM architecture in Section 7.10.

7.1 EPA-RIMM Checks

EPA-RIMM's Checks consist of measurements over resource types including memory and registers. EPA-RIMM's architecture envisions the ability to add a variety of new Checks in response to emerging attacks.

Each Check contains a command along with its arguments and a priority. Checks allow the Administrator to specify particular measurements over sets of memory regions, Control Registers, and Model-Specific Registers (MSRs). Sample Checks include the: "Kernel Code Section Range", "Hypervisor Code Section Range" Check that measures the host software kernel code sections to identify code injections, the "Interrupt Descriptor Table" Check that measures

the contents of the Interrupt Descriptor Table (IDT), the "Interrupt Descriptor Table Register" (IDTR) Check that verifies that the IDTR register value has not changed, the "Global Descriptor Table" (GDT) Check that measures the Global Descriptor Table to determine if it has changed. Other Checks measure specific MSRs or CPU control registers, for example, to determine if the Write Protect bit on CR0 was cleared or the Supervisor Mode Execution Protection (SMEP) on CR4 were disabled by a rootkit. Figure 7.1 shows several sample Checks.

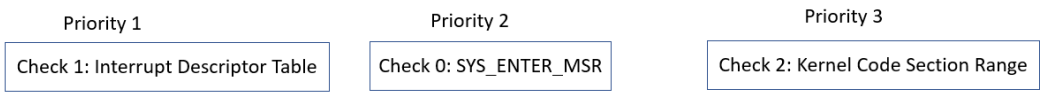

FIGURE 7.1: Checks

7.1.1 Check Definition

The EPA-RIMM Administrator specifies Checks from the Check definition template. Table 7.1 shows the fields supported by Checks and their decomposed expression, Tasks. Each Check and Task have an ID# for reference. The *Command* is an instruction to the Inspector that specifies the category of measurement to perform, e.g. Measure Virtual Memory, Measure Physical Memory, Measure Register, or Measure MSR. The *Operand* field allows specifying which Register or MSR should be measured (for Register or MSR measurements.) The *Memory Address* field directs the SMM Inspector to begin a memory measurement at the specified Memory Address and the *Length* fields specifies the length of the measurement. The *Priority* field provides guidance over when the measurement should be scheduled, with higher-priority measurements performed before lower-priority measurements. The *Hash* field provides the hash result for the measurement based on provisioning.

TABLE 7.1: Check and Task Descriptions

Check	**Task**	**Description**
Id#		Unique Id
Command		Measurement
Operand		Command Arg
Memory Address		Starting Address
Length		Measurement Size
Priority		
	Hash	Value to Compare

7.1.2 Measurement Commands

EPA-RIMM supports several measurement commands to identify the presence of rootkits including: Measure Memory Range, Sample Memory Range, Measure Control Registers, Measure Model-Specific Registers.

7.1.2.1 Command: Measure Memory Range

The *Measure Memory Range* command allows the Administrator to completely measure the specified memory range. The goal of this check is to detect unexpected changes in presumed static memory regions, for example: kernel or hypervisor code sections, read-only data structures, or any memory region which is not expected to change.

This command supports the following options:

1. Virtual or physical memory range: Start address

2. Measurement size (number of bytes)

7.1.2.2 Command: Sample Memory Range

The *Sample Memory Range* command measures a statistical portion of a memory range. This allows leveraging spatial locality for measurements. In

contrast to the *Memory Range* command, the *Memory Range Sampling* command performs sampling over a memory range. The command supports measurement densities from 1% to 100% of the full range. This Check has the potential of reducing the overall number of EPA-RIMM measurements. We gratefully acknowledge the contributions of Bruce Irvin who provided the initial vision for this Check.

This command supports the following options:

1. Virtual or physical memory range: Start address
2. Measurement size (number of bytes)
3. Measurement Density (range from 1% to 100%)
4. Sampling algorithm (random)

7.1.2.3 Command: Measure Control Registers

The *Measure Control Register* command is designed to detect changes that rootkits may make to these privileged registers, for example, turning off write protection for kernel code. This command supports the following options over the supported set of CPU Control Registers:

1. Control Register: CR0, CR3, CR4
2. Measurement size (4 or 8 bytes)

7.1.2.4 Command: Measure Model-Specific Registers (MSRs)

This command supports the following options over supported MSRs:

1. MSR Index
2. Measurement size (4 or 8 bytes)

7.2 Tasks

EPA-RIMM Tasks are a key component for limiting the amount of time spent in a single SMM session. Checks are decomposed into Tasks with the goal of accomplishing them over a period of time. This approach performs less work in a single SMI session, however, allows completing the entire measurement over a sequence of measurements. Each Task contains a priority that is inherited from the Check and also a cost. Figure 7.2 shows the decomposition of several Checks into Tasks.

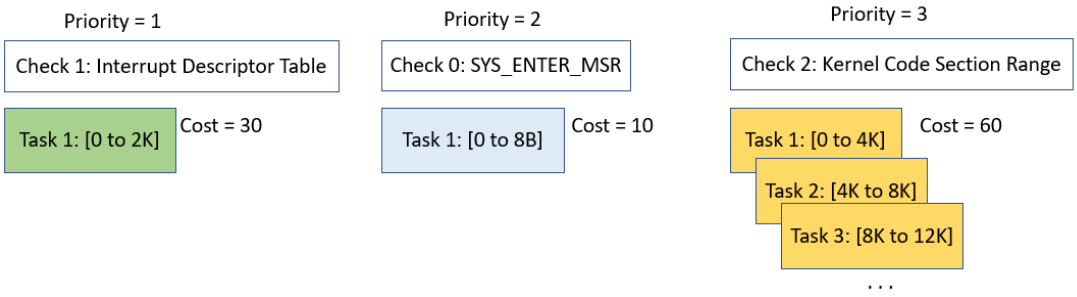

FIGURE 7.2: Tasks

7.3 Bins

Bins are collections of one or more Tasks for processing in a single SMI session. In Figure 7.3, the Bin cost is set to a maximum of 100μs which allows three Tasks.

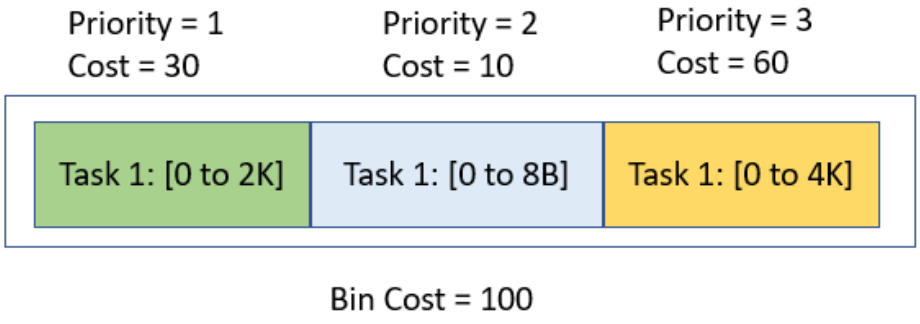

FIGURE 7.3: Bin

7.4 Diagnosis Manager

The Diagnosis Manager (DM) orchestrates the runtime integrity measurements on a separate node from the monitored node. It initiates the Checks, and interprets the measurement results. A single DM may be responsible for one or more monitored nodes. The DM sends and receives information about attack discoveries from across the EPA-RIMM framework to help guide detection on other monitored nodes. This allows dynamically adjusting the priority of Checks to search for detected issues on other nodes.

7.4.1 DM Provisioning

The DM is initialized with a set of specific Checks. The currently supported EPA-RIMM commands and their parameters are: Register (CR0, CR3[1], CR4, IDTR, GDTR), Mem (Address, Length), and MSR (MSR Number). Checks that measure large memory regions could exceed desired SMI session times and need decomposition. To determine a suitable granularity, EPA-RIMM measures the cost per byte of various hash sizes during the provisioning phase, then uses this data with the Check's Decomposition Target to fine-tune the amount of work performed in a task. Checks involving Control Registers or MSRs cannot be decomposed and thus consist of a single Task.

7.4.2 DM Runtime

The DM sends Checks to the Backend Manager. Each Check returns a result of unchanged or changed, indicating whether the measurement matches the comparison value. Changed results cause the DM to raise an alert. EPA-RIMM's provisioning phase and Check data structures provided the fundamental basis of directing the measurement operations. By basing the

[1] CR3 is dynamic and can change.

measurements on provisioned data from the monitored node and allowing measurement specification from the DM, we remove the requirement to put this logic into SMM code itself. This also guides the Inspector's comprehension of host software which resolves the semantic gap between SMM and host software.

7.4.3 Measurement Triggers

EPA-RIMM measurement triggers provide a means of reducing the amount of measurements required to determine if a given security hypothesis is true or false. We provide a description over how to specify measurement triggers in Section 7.4.3.1 and describe two measurement triggers in Section 7.4.3.2 and Section 7.4.3.3.

7.4.3.1 Specifying Measurement Triggers

Measurement triggers evaluate security hypothesis in a procedural manner to reduce the number of measurements required to evaluate the hypothesis, for example, performing lighter-weight measurements before heavier ones. This allows detecting compromises with reduced SMM overheads. A trigger may have two types of actions:

1. Independent Action: This action does not have a dependency on the previous action.

2. Dependent Action: This action is only run if the previous action returns a changed result.

A trigger with only independent actions is indicative of security inspections that require checking all avenues of compromise for a given resource. For example, an IDT rootkit could be accomplished by either a change of the IDTR or one of the IDT entries. As one change is not dependent on the other,

both resources must be checked. However, to allow for earliest detection at minimal measurement cost, the IDTR register measurement is performed first as a detection of a change would provide an alert at minimal cost.

Triggers with only dependent actions imply a logical ordering of successive measurements that lead to an advanced security diagnosis. For example, with a kernel code compromise with completely persistent changes, the CR0 write-protect bit must be disabled before the kernel code can be modified. However, a detection of a changed CR0 register does not guarantee that the code sections were modified and thus, the code sections must be measured to complete the diagnosis.

7.4.3.2 Example Measurement Trigger: Kernel Code Sections Unchanged - Persistent CR0 and kernel code changes

HypKernelFunctionsTampering hypothesizes that signs of kernel code tampering are evident. Evaluation of the hypothesis begins with a light-weight measurement. If this measurement indicates a changed result, then the hypothesis has been verified without involving additional measurements. However, if the measurement does not indicate a changed measurement, then further measurements must be done. As an example, the following steps allow evaluation of this hypothesis:

1. Control Register 0 Write Protect bit changed.

   If True, signs of tampering are present and no further measurements are required to evaluate this hypothesis.
   If False, proceed to next measurement.

2. Kernel code sections are changed. This measurement requires hashing all of the kernel code sections in smaller Task sizes which is a more

expensive operation than the Control Register measurement.

If True, signs of tampering are present and no further measurements are required to evaluate the hypothesis.
If False, there are no additional enabled measurements and the hypothesis is evaluated as False.

Figure 7.4 shows the flow.

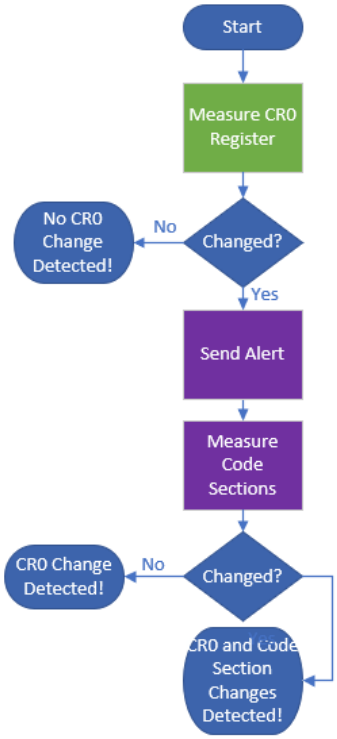

FIGURE 7.4: Persistent Kernel Code Section and CR0 Trigger. Purple boxes are dependent actions.

7.4.3.3 Interrupt Descriptor Table Unchanged

Figure 7.5 provides the flow for the IDT trigger. This trigger begins with a light-weight measurement of the IDT register (IDTR). If this measurement indicates a change, the Trigger completes without incurring the cost of the IDT measurement.

However, if the IDTR check indicates no change, it is still possible that the attacker has modified the IDT itself. As the IDT can be modified without changing the IDTR, this action is independent of the previous action. If the IDT table has changed, the trigger completes with a change detected, otherwise, it completes with no change detected.

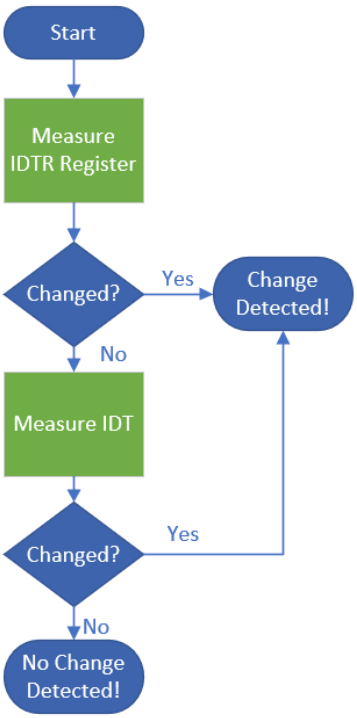

FIGURE 7.5: Interrupt Descriptor Table Trigger. Green boxes are independent actions.

7.5 Backend Manager

The Backend Manager (BEM) manages the performance aspects of EPA-RIMM and provides measurement requests to the monitored systems. It resides on a separate node from the monitored node. It receives Checks from the Diagnosis Manager and decomposes them into smaller Tasks to avoid prolonged SMM session times. The granularity of the decomposition is specified by the Decomposition Target parameter. (See Table 7.2.) The BEM schedules Tasks

by filling Bins based on a target Bin size. It signs, creates a MAC (Message Authentication Code), encrypts each Bin, and then provides it to the Host Communications Manager which interfaces with the SMM-based Inspector. The BEM waits to receive the Inspector's Results back. It decrypts the Results and checks the signature and MAC to ensure that they came from the proper Inspector and were not tampered with in-transit. The BEM merges results of all the Tasks for each Check into a single Result and sends it to the DM.

Check decomposition reduces system impacts, allowing larger, more frequent, and less predictable measurements. This approach trades atomic measurements for partial results over time. Scrubbing attacks are challenges for SMM-based runtime integrity mechanisms [7], and frequent measurements may reduce transient malware's time window to operate. Malware that installed itself and later tried to remove itself could be detected via EPA-RIMM re-measurements. Check decomposition involves an overall efficiency and space trade-off as smaller amounts of work processed per SMI result in more SMIs in total. Additionally, a larger number of Tasks requires more storage space on the BEM. Thus, there is a trade-off between SMM latency and overall measurement efficiency.

TABLE 7.2: Decomposition and Bin Size Parameters

Value	**Configured at**	**Applies to...**
Target Bin Size	Runtime	Current Bin Size
Max Bin Size	Provisioning	Bin Size Limit
Decomp. Target	Provisioning	Task Granularity

7.5.1 BEM Provisioning

The BEM is provisioned with the appropriate keys for signing and encryption. The BEM's Task performance estimations are set based on an initial performance measurement so that the EPA-RIMM can begin runtime operation

with an appropriately sized amount of work in a Bin. Since there is no preemption of SMM code, platform-specific performance prediction is accurate. Initial measurements that will be re-checked over time can be gathered in the provisioning phase in an offline environment (preferred) or upon the first measurement of a resource. Storing the measurement hashes in the BEM avoids scalability issues related to limited SMRAM.

7.5.2 BEM Runtime

Because the system experiences an overhead transitioning to SMM and back, minimizing the number of SMIs is a consideration. Since each SMI transfers one Bin, efficiently filling the Bin reduces the number of SMIs and consequently the amount of time spent transitioning to and from SMM. We use a priority queue for Bin packing, so Tasks are not scheduled in strict priority order; a lower priority Task might be selected to "fill" a Bin in place of larger higher priority Tasks. Priorities are adaptive; Tasks are assigned an initial priority based on their parent Check, but priorities change at runtime, for example, with aging. The BEM may increase or decrease the Bin size within set bounds. The BEM may also increase or decrease the SMI frequency upon direction from the DM.

7.6 Oracle

The Oracle is responsible for maintaining the provisioned hashes. It resides on a separate node from the monitored node. This allows abstracting the machine-specific parameters from the BEM. At runtime, the BEM will request the provisioned hashes from the Oracle to provide to the Inspector to allow comparing the current measurement against the provisioned hash.

7.7 Host Communications Manager

The Host Communications Manager (HCM) resides within the monitored system and provides an interface between the Inspector and the BEM. As SMM does not feature a dedicated network stack, an interface needs to be enabled for communication with it. The HCM mechanism should be out-of-band of the OS, such as the BMC (Baseboard Management Controller) [7]. In-band mechanisms (e.g. Ring 3 application and kernel module) should not be used as they are vulnerable to malware. For example, if a HCM process were to be killed by malware, the measurement would stop. While the BEM could detect a lack of response and trigger an alert, there is a subtler attack that is possible. Malicious code could recognize that a measurement request is imminent, clean its traces, and then let the measurement proceed. The HCM receives Bins from the BEM and provides them to the Inspector when it triggers its operation via an SMI. The HCM also receives Results from the Inspector which it returns to the BEM.

7.8 Inspector

The Inspector performs the measurements from SMM, noting differences compared to the comparison measurement. The Inspector is compiled into the BIOS and is initiated via an SMI. It has the ability to view the interrupted host-side CPU register state, MSR values, and allowable regions of the host-side memory space. The Inspector also monitors the measurement cost in terms of time and returns the cost to the BEM so that it can adaptively tune the Bin size. The Inspector checks the Bin's Message Authentication Code (MAC) creates a new MAC for the outgoing Results to ensure in-transit integrity.

7.8.1 Inspector Provisioning

The Inspector must be provisioned with encryption and signing keys for its communications with the BEM. EPA-RIMM does not prescribe a particular key provisioning method and leaves the implementation up to the implementer. EPA-RIMM is compatible with TPM-based key provisioning or a Diffie-Hellman key exchange using public keys embedded in the firmware.

7.8.2 Inspector Runtime

The Inspector will be invoked by an SMI that specifies the Bin for the Inspector. The Inspector returns the Results as shown in Table 7.3.

TABLE 7.3: Results Description

Results Entry	**Source**
Check ID#	BEM
Task ID#	BEM
Result	Inspector
Measured Hash	Inspector
Measurement Cost	Inspector
Inspector Signature	Inspector
Results Integrity Measurement	Inspector

7.8.3 Complete Architecture Flow

Figure 7.6 provides a complete example of the EPA-RIMM architecture. In this example, we show three Checks: A, B, and C with different costs and priorities. The Diagnosis Manager sends these Checks to the Backend Manager which decomposes them into Tasks using a Task Decomposition Target of 75μs and places them into a priority queue. The Backend Manager then forms Bins with no more than 100μs. The Backend Manager then signs and encrypts the Bins and provides to the Host Communications Manager which passes the Bin to the Inspector via an SMI. The Inspector processes the Tasks in the Bin and observes a difference in Check A - Task 0 which it returns back through

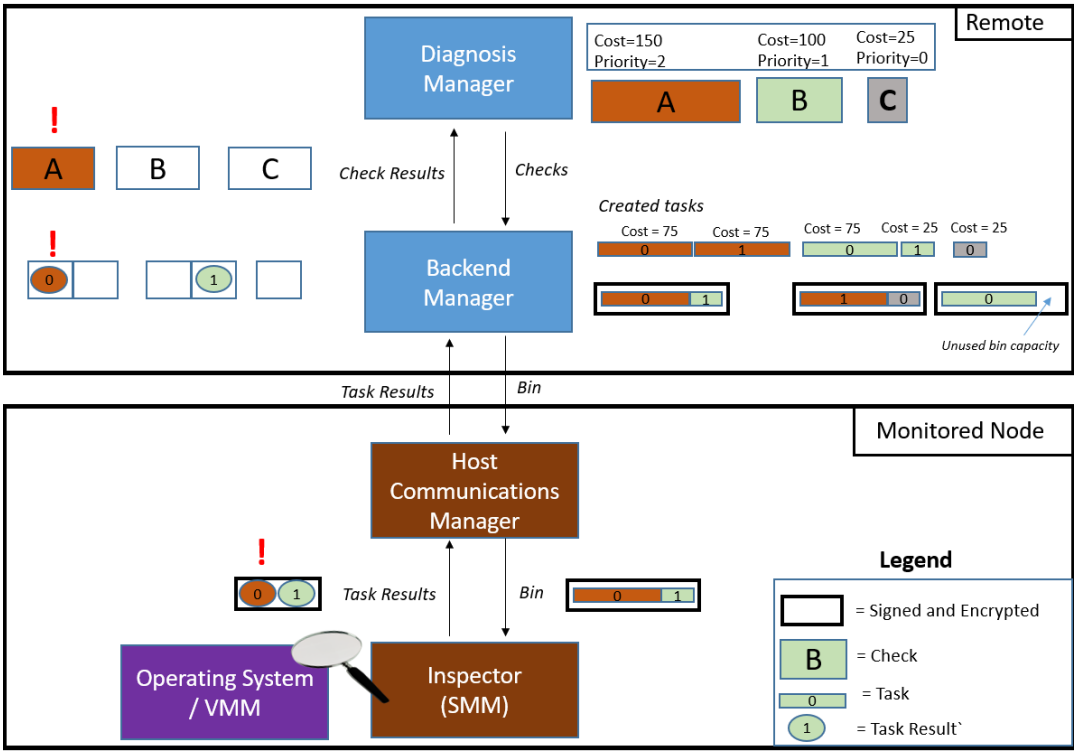

FIGURE 7.6: A complete example of EPA-RIMM's active monitoring phase. In this example, the same Bin is provided to all monitored nodes, but in a heterogeneous environment the Bins and the hash costs could differ between nodes. We show the BEM and the Inspector as residing on separate machines, but there is no requirement for this separation.

the Host Communications Manager to the Backend Manager. The Backend Manager consolidates the results and provides to the Diagnosis Manager.

7.9 Security Analysis

In this section, we describe our assumptions and analyze potential threats against EPA-RIMM. We consider attacks against SMM and EPA-RIMM components and on requests and results, side channels, initial measurements, infrastructure compromise and denial of service, crypto/signing attacks, and transient evasion attacks.

Research Question 1: How to design a more secure measurement agent?

7.9.1 Assumptions

We assume that initial measurements can be gathered during a provisioning process and that OS updates that change monitored resources trigger a re-provisioning. We also assume that SMRAM is well-protected and leverages available hardware protections including SPI protections over the BIOS chip and proper SMRR configuration. The CHIPSEC tool can be used to verify proper platform SMM configuration [68]. EPA-RIMM also assumes the presence of an out-of-band network interface to allow communication of measurements requests and results. We assume the out-of-band interface is not malicious. EPA-RIMM targets scenarios where an attacker has gained control over the operating system or hypervisor at runtime. This can include code injection into these privileged layers.

7.9.2 Inspector

The Inspector, residing within SMM, may be targeted by the attacker. One potential attack is a confused deputy attack in which the attacker attempts to trick the Inspector into overwriting SMRAM memory or other privileged memory. The Inspector should check input buffers to ensure that they do not reside within the SMRR. The Inspector should also communicate directly with the BEM and not write data into OS-controlled memory. Attacks on EPA-RIMM's Inspector could attempt to forge a measurement request from a malicious BEM in order to gain additional insights into operation of the system. For this reason, it is important that the Inspector and BEM properly authenticate with each other. The Inspector and BEM also use encryption for their communications to prevent eavesdropping or tampering. The nonce prevents replay attacks in which previous measurements are passed off as current measurements. The Inspector should not return more information

than is required to determine an unwanted change has occurred. By returning hashes instead of actual measurement values, the Inspector helps limit its potential usefulness as a side channel.

7.9.3 Initial Measurements and EPA-RIMM launch

Initial measurements should be provisioned in an offline environment to avoid compromised values appearing as pristine values. In homogeneous environments with identical kernel and OS versions, it may be possible to gather an initial measurement on a representative node for comparison other identically-configured nodes. New kernel versions that come with an operating system update would require re-provisioning due to new memory layouts. Once the host software launches with a trusted boot, EPA-RIMM can begin servicing measurement requests over the out-of-band HCM interface.

7.9.4 Infrastructure Compromise and Denial of Service

If a Denial of Service were to affect the DM or BEM, the flow of measurement requests would slow or cease. Monitoring of the flow of EPA-RIMM measurements would be necessary to identify this type of attack. If the DM were to be compromised, it would be possible to misdirect EPA-RIMM to monitor an unrelated set of resources while an attack executes or share false reports of attack detection. For this reason, the EPA-RIMM Administrator should monitor and investigate the threat intelligence exchanged by EPA-RIMM and also audit the Checks that are being performed for unexpected changes. A compromise of the BEM could expose the hash database. However, this is of limited use as it contains hashes instead of memory contents.

A denial of service attack against the measurement agent cannot be completely avoided. For scenarios where improper Bins are provided, processing time can be minimized by aborting processing of the Bin once a problem with decryption or signature is identified. Administrators or supervisor software could monitor the number of SMIs processed by reading the MSR_SMI_COUNT or an STM-based SMI counter periodically to determine if a high rate of SMIs is occurring. For the scenario in which the attacker can construct valid Bins, the Inspector is provisioned with a maximum limit for memory measurements to avoid spending excessive time in an SMI session. The EPA-RIMM administrator specifies the limit based on benchmarks for the system that produce a hash cost per byte metric. This will not completely thwart a denial of service attack but will reduce a degree of its effects. The measurement agent could also consult the MSR_SMI_COUNT and if it detects a higher rate of SMIs than allowed by provisioning, could abort further Bin processing, returning errors to the Backend Manager which would trigger an alert.

There are two key aspects to address for minimizing SMM-based denial of service attacks: 1. Duration of SMI, 2. Frequency of SMIs. For EPA-RIMM, SMI duration is primarily determined by the measurement size. An attacker who succeeds in requesting an overly large measurement could consume large amounts of time in SMM, preventing the system from performing other work. Likewise, an attacker who specifies a flood of measurements could also cause the system to spend significant amounts of time in SMM. Note: while EPA-RIMM can take safeguards against being used as a denial of service mechanism, SMM-based denial of service attacks are always present as any attacker with Ring 0 privileges can generate an arbitrary SMI.

To address the duration issue, the EPA-RIMM provisioning phase performs hash measurements starting from powers of 2^{10} until the measurement results exceed $\mathbf{LimitSmi_{Empirical}}$. The measurement agent records the maximum hash size that was below
$\mathbf{LimitSmi_{Empirical}}$ and rejects measurements with lengths that exceed this.

EPA-RIMM addressed the SMI flood attack by allowing a maximum number of inspections per minute. The measurement agent increments a counter with each measurement request and checks the counter value against the maximum-allowed value specified during provisioning. Measurement request frequencies that exceed the allowable maximum value are rejected.

Attackers who try to invoke false positives by changing resources on a monitored node to trigger a flood of alerts, would succeed in triggering these alerts but would not evade notice that monitored resources had been successfully changed.

7.9.5 Transient Evasion Techniques

All snapshot-based periodic inspections have the potential to miss attack detection if signs of the attack were not present at the measurement interval. EPA-RIMM, unlike other SMM RIMMs, can be used to measure more frequently, in smaller portions to reduce the amount of time between periodic measurements. Additionally, varying the set of measured items dynamically at runtime leads to less predictable measurements which complicates the attacker's task.

7.9.6 Stealth

It is difficult for SMM RIMMs to be completely stealthy and EPA-RIMM is no exception. A motivated attacker could leverage timing information to ascertain losses of control. The developers of a stealthy SMM-based debugger,

MALT, note that while they were able to adjust various system timers in SMM to hide their operation, a dedicated attacker could send an encrypted message to a remote timing server to get accurate sense of time [118]. For these reasons, while SMM-RIMMs operate independently from host software, complete stealth appears infeasible.

7.9.7 Host-side Memory Visibility

Most SMM-RIMMs grant privileges beyond what is absolutely required by their measurement agent. By utilizing the SMI Transfer Monitor (STM), the principle of least privilege could be applied to the measurement agent [110, 84]. The STM is a thin SMM-based hypervisor that virtualizes SMI handlers in their own virtual machine and applies a protection policy over this virtual machine to constrain their accesses to platform resources.

We implemented a set of STM policies to restrict the measurement agent's access to the allowed set of resources. As the measurement agent does not need to modify host memory but only read the memory for the purpose of hashing, host memory writes are prohibited to the measurement agent. To allow the measurement agent to return results via a memory write, the EPA-RIMM architecture can leverage the UEFI Communications Buffer which provides a BIOS-reserved memory region for communications [116]. EPA-RIMM does not need to modify MSRs or IO Ports and can thus read-only access to measured resources allows the measurement agent to have a suitable level of access. EPA-RIMM requires host memory read access where it would otherwise not be required. Thus, there is a necessary trade-off in the goals of: 1. Restricting SMI handler access to host resources and 2. Providing SMM-based runtime integrity measurement. SMM policies that provide increased rootkit detection at minimal impact to SMM security represent an overall improvement in system security.

7.9.8 KASLR

The kernel address space layout randomization (KASLR) feature would require special handling with EPA-RIMM measurements as the kernel addresses would be randomized. One option for supporting KASLR would be to generate new provisioned values upon initial boot. This allows KASLR to be enabled in a method that is compatible with EPA-RIMM. At present, the future of KASLR is unclear [36]. There have been several recent attacks on the KASLR feature using page faults, prefetch, Intel TSX, and Branch Target Buffers and several mitigations proposed [41, 37, 51, 28].

7.9.9 Spectre/Meltdown

In recent months, the Spectre and Meltdown attacks [64, 57] have received significant coverage. Some of these attacks are applicable to SMM. Intel has provided guidance about software remediations for the attacks [43] and also has released CPUs with upgraded hardware-based fixes. The Bounds Check Bypass (CVE-2017-5753) uses speculative execution following branch instructions. For this vulnerability, adding LFENCE instructions before a bounds check can mitigate the attack. Intel notes that "Overapplication of LFENCE can compromise performance [43]. As a goal of EPA-RIMM is to bound the time spent in SMM, this mitigation reduces the available amount of SMM time for EPA-RIMM processing. Thus, a careful consideration of where LFENCE operations may be required would be necessary to mitigate the side-channel attack and maintain acceptable EPA-RIMM performance. Another attack, Branch Target Injection, leverages indirect branch predictors to control which instructions are speculatively executed after a near indirect branch predictor. Intel has announced two remediations, the first of which provides control over which indirect branch speculations are allowed. The second

creates a "retpoline" which substitutes near jump and call instructions with an alternate code sequence that invokes a direct call. This approach may result in a reduced performance overhead for the mitigation. EPA-RIMM developers should analyze the impact of these two remediations on the Inspector to determine the appropriate balance between side-channel protections and Inspector performance.

7.9.10 Attacks on Measurement Agent Communications

EPA-RIMM achieved protections for confidentiality, authenticity, and integrity by means of encrypted, signed, and authenticated communications. Encryption over the communications protects against malicious inspection of the contents as each message is not able to be parsed by an attacker without the proper key.

Falsified communications can be remedied by signing measurements and results to ensure they originated from the proper entities. While EPA-RIMM does not prescribe a particular signature mechanism, RSA provides a compatible implementation. RSA cryptography leverages public and private keys to encrypt messages for a specific target. During provisioning, the measurement agent conveys its public key to the Backend Manager and receives the Backend Manager's public key. Each entity is responsible for maintaining the secrecy of their private keys. The measurement agent stores its private key in SMRAM which is unavailable to code in the operating system or hypervisor. The Backend Manager may store its key in the Trusted Platform Module (TPM), a hardware-protected chip that is commonly used to store keys. The Backend Manager uses its private key to encrypt messages for a particular measurement agent and the measurement agent utilizes its private key to decrypt the message from the Backend Manager. The reverse flow of measurement results uses the same mechanism.

EPA-RIMM addressed the possibility of spoofed messages using a signature check. Messages that are received without a proper signature can be rejected by either the measurement agent or the Backend Manager. Listing 7.1 shows detection by the measurement agent of a spoofed sender.

LISTING 7.1: Signature example

```
Correct signature: 4d414e4147455231323334...
Spoofed signature: 55444333221100AABBCCDD...

ERROR: BEM signature mismatch!
```

To protect against a communications tampering attack, EPA-RIMM leverages an HMAC over measurement requests and results to obtain confidence over message integrity [58, 11]. HMAC uses an encryption key and hash algorithm to verify the data integrity and authentication of a message. The data integrity check can detect a change in the measurement request or results after initial transmission of the message.

While encryption provides protection against an attacker being able to discover the contents of a measurement request or result, it does not provide full protection against an attacker that modifies a portion of the data, e.g. the message integrity. For this reason, an HMAC provides the ability to detect message tampering. Listing 7.2 shows an example of an HMAC being constructed over the contents of an unmodified communication and then rechecked after an attacker modifies part of the message. The tampering results in a different HMAC which the measurement agent can then be aware of.

LISTING 7.2: HMAC example

```
HMAC over unmodified message:
```

```
7b8dd1535678ce49728292a6c582e702
07e4844bcf954b7685fbe48771837486

HMAC over modified message:
83368892920c4148cbc7281b35b8c24b
24a87185c957ffb513401463fd004f8d

"ERROR: HMAC differs!"
```

To address replay attacks where an attacker tries to replay copies of previous measurement requests or results, a nonce provides the ability to sequence communications such that each message has a unique sequence number and attempts to pass off old messages as new messages can be detected. EPA-RIMM's communications include a nonce to ensure measurement liveness.

For each Bin that the Backend Manager creates, it generates a unique nonce and a unique identifier for the Bin. It associates the nonce with the Bin identifier in its internal memory. If the Backend Manager were to receive a Result for a given Bin with an improper nonce, it would discard the Result and would raise an alert that it received an incorrect nonce value, which may be indicative of an attack on EPA-RIMM.

EPA-RIMM leverages encryption over measurement requests and results to protect against an adversary who endeavors to view the contents of these messages. Applying cryptography incurs performance costs within a limited time budget in SMM, which necessitates selecting performance-efficient cryptography mechanisms. EPA-RIMM does not prescribe a specific cryptography implementation, however, there are implications based on which method is selected. One approach is to use public and private key cryptography to establish a secure channel in which an AES symmetric key can be exchanged.

Encryption provides a critical capability in hiding the contents of EPA-RIMM measurements from an attacker. Listing 7.3 shows the contents of an intercepted measurement result from the measurement agent. The fields are not useful to the attacker as the attacker cannot derive the results of the EPA-RIMM measurement.

LISTING 7.3: Encrypted measurement result

```
IV                      0x1672d3b32e7 0x1672d3b3342
Command                 027fe03e3445e1be9
Operand                 0x82fc469e9d38c16a
Virtual Address         0x9d3a395f7820d12e
Phys Address            0x39355208988e5bf9
Length                  0x9e2d8af570319ead
Result                  0x3af930fbf4075f0f
Nonce                   0xe3222c1fdb601ca0
Cost                    5d133a485c3e5cca
Task UUID               3b245fa1173c6740
Reserved1               46094808c2bc3445
Hash:                   3b4da65d 8bdc39ed 220d60fa ...
Manager Signature:      ff01ccd7c82a64a37e3ef9ca672 ...
Inspector Signature:    ff9e49affc8468859e78abbced2 ...
HMAC                    6fdb27f61196ec55a931589a645 ...
```

7.9.11 Use of EPA-RIMM as a side channel

EPA-RIMM limits the amount of useful data to an attacker by reporting hashes instead of raw values from the system. This prevents attackers from observing

register values, memory contents, and MSR values that might increase an attacker's understanding of the actual system state.

EPA-RIMM measurements consist of a flow of hash values for resources measured. An attacker who succeeds in compromising the Backend Manager and endeavors to learn useful measurement details from the system being monitored would only have access to hash values from the system, as shown in Figure 7.7. As hashes consist of a mathematical one-way operation that does not allow reconstructing the initial values, attackers will not gain useful information from the data.

Sample# **Hash**

1. 2312041994 1522703948 2003520127 2771094464 1207174385 4098032895 3882206773 3549063283
2. 4290169166 2059379603 1595788773 2391455268 0577867196 3279052461 3559606428 3335050934
3. 3229603466 2536425739 1494895723 3128019217 4058557060 0197835174 1107849095 1027491922
4. 0273522843 3567702509 1330620092 3090294126 0638415955 1374511360 2345956605 0117388516
5. 2802769891 0413381580 0595099389 1447580119 2937619112 3015307390 0881392582 1863494457

FIGURE 7.7: Hash samples

7.10 Conclusions

The EPA-RIMM architecture provides a means of meeting the design requirements described in Section 6. EPA-RIMM allows the specification of Checks which the Backend Manager decomposes into Tasks to bound the amount of work performed in a single SMI session. Bins represent the complete set of work to be performed by a CPU thread in a single SMI session.

To resolve the SMM-RIMM semantic gap and remove the need to store context in SMM, we identified a set of measurement primitives to detect a variety of known rootkit and ransomware techniques, namely: virtual and physical memory ranges, CPU control registers, and MSRs. These fundamental building blocks for integrity measurements are able to detect a variety of

contemporary host software rootkits and ransomwares. For future extensibility, the API can support new primitives by specifying the new parameters and adding the corresponding measurement code in the Inspector. This can all be accomplished without requiring broader changes in the EPA-RIMM architecture.

Measurement Triggers provide the means to perform advanced security diagnosis in a logical method by performing light-weight measurements prior to heavier measurements. Dependent trigger actions provide a means of avoiding heavier measurements until lighter-weight measurements demonstrate the need for this additional work. By applying the use of measurement triggers, we employed a targeted approach towards analyzing potential attack scenarios which reduced the amount of checking required. This allowed evaluation of security hypothesis with reduced measurement cost.

Beyond limiting the time spent in a single SMI session, we needed to address the overall performance impacts of repeated SMM-RIMM measurements to lessen the system impact. We investigated whether a hypotheses model developed in the performance analysis field (Paradyn) can be applied to EPA-RIMM's security inspections [80]. In this model, lighter-weight measurements are performed first to evaluate whether a potential hypothesis can explain performance problems. If lighter-weight measurements indicate a potential performance problem, then more invasive and costlier measurements are performed. This technique had not yet been applied to SMM-RIMMs. SMM-RIMMs need to balance the amount of inspections they perform with their impact on the system. In the ideal case, each security-sensitive resource could be checked at every CPU clock tick, presenting no opportunity for an attacker to escape detection. However, this would result in a system that was unusable for practical purposes. Thus, the SMM-RIMM needs to provide

effective detection at a reasonable performance trade-off. We investigated whether this approach can also be used in EPA-RIMM to provide effective rootkit detection at reduced performance cost.

The security of the EPA-RIMM architecture is very important as vulnerabilities in its SMM code would present significant concern. A key method for dealing with this concern is to run the Inspector inside a virtual machine to constrain its accesses [110]. This allows applying the principle of least privilege to the measurement agent. By storing hashes instead of raw values, EPA-RIMM reduces the amount of actionable information an attacker can glean from the measurements. The EPA-RIMM architecture provides a usable example of an SMM-based RIMM that balances performance, the principle of least privilege, and effectiveness which is unique among other SMM-RIMMs.

8
EPA-RIMM Prototype

In this section, we describe our EPA-RIMM prototype including its various software modules. We leverage open-hardware platforms with open-source UEFI implementations for this prototype. We cover attack detection with the prototype and its impact on application performance.

8.1 Prototype Overview

To test our design, we developed a prototype which implements the necessary portion of the EPA-RIMM architecture. Our prototype systems do not have an out-of-band communication mechanism, therefore we demonstrate the functionality using an in-band mechanism. This is sufficient for the research prototype since it does not share the security requirements of a production system. We implemented four separate modules: the BEM, an in-band HCM (consisting of the "Frontend Manager" (FEM) and "Ring 0 Manager" (R0M) modules), and the Inspector. The BEM runs on a network server while the other components reside on the monitored system. The initial prototype does not implement the Diagnosis Manager although an updated software stack adds a basic version of this component. The prototype is available at: https://github.com/PPerfLab/EPARIMM-Release.[1]

8.1.1 Hardware

We enabled our prototype on two open-hardware systems: The Minnowboard Turbot and the UP Squared ("UP2") board. Both boards leverage the x86 CPU architecture, feature open-source UEFI firmware, and provide support flashing modified firmware. Both boards have less computational resources

[1]The BEM used in this dissertation is available for internal testing and benchmarking, however, is not released in this github. An updated BEM has been released.

than a typical server platform that EPA-RIMM would be running on. The Turbot features a dual core Intel Atom e3826 processor with a base clock of 1.46 GHz and 2 GB RAM [49]. The UP2 board has a more recent Intel Pentium N4200 CPU with a base clock of 1.1 GHz and 8 GB of RAM. Both boards feature onboard Ethernet and support an attached solid-state disk (SSD). We use Ubuntu 14.04 64bit with a 4.11 kernel on these systems.

8.1.2 Firmware

We modified the SMI handler in the respective UEFI source for the Turbot and UP2 boards by creating a DXE_SMM_DRIVER which registers a new software SMI using the EFI_SMM_SW_DISPATCH_PROTOCOL. We integrated OpenSSL 1.1.0e support into the Inspector for SHA256 hashing for measurements, AES256-CBC for encryption, and HMAC SHA256 for integrity. We configure the SMM page tables to allow the necessary access to host memory to allow memory measurements, receiving measurement requests, and sending measurement results.

8.2 Prototype Modules

8.2.1 BEM

Our BEM is the primary interface to running EPA-RIMM measurements on the prototype. We created a script to identify relevant sections of the kernel code to measure, e.g. code sections and read-only data sections and create fixed-size Tasks for the BEM. We also provide an interface for the user to configure which control registers and which MSRs to measure.

Once the Tasks are entered into the BEM, it establishes a priority queue for the measurement Tasks and groups them into Bins based on a first-fit algorithm. The BEM sets up a network socket connection with a set of FEMs (one per monitored system). As each monitored system may process Tasks

at different rates, the number of Tasks that fit in a given Bin size may vary across systems.

The BEM supports a wide variety of testing options:

1. Run Duration Options: Run forever, run for X Bins, run for X priority queue refills, re-run previously measurements tasks (yes or no), send Bins for a timed duration.

2. Bin transmission parameters: Random delay between Bins (yes or no), maximum random delay between Bins, duration of delay between Bins.

3. Bin Encryption enabled (yes or no), Result decryption enabled (yes or no)

4. HMAC creation enabled (yes or no), HMAC comparison enabled (yes or no)

5. Measurement targets: Measure [Memory, Control Registers, MSRs] (yes or no), Measure IDTR and IDT (yes or no), sample memory range enabled (yes or no), sample granularity (bytes), sample density (percentage)

6. Maximum Bin size

8.2.2 HCM

We implemented the HCM prototype with a FEM and Ring 0 Manager. The FEM receives the signed and encrypted Bin from the BEM and writes it to a /proc interface that is registered by the Ring 0 Manager. After each Bin has been processed, the FEM retrieves the Results from the /proc interface and send them to the BEM. The Ring 0 Manager (Linux kernel module) receives the Bin from the FEM via the write to the /proc interface. The Ring 0 Manager

then records the virtual memory location of the Bin in a CPU register. After this, it triggers the measurement SMI by writing a pre-arranged value to port 0xB2.

8.2.3 Inspector

The Inspector registers an SMI that will receive measurement requests. Upon receiving the SMI, the Inspector will locate the Bin in memory, converting the virtual memory address of the Bin to physical. The Inspector checks to ensure that the Bin is not within SMRAM memory to avoid overwriting SMRAM memory. The Inspector decrypts the Bin, checks the signature, verifies the provided HMAC, and performs the specified measurements. To prepare the results, the Inspector generates an HMAC over their contents, and then encrypts and signs the Results data. It then copies the Results out to the Ring 0 Manager.

For the provisioning phase, the Inspector computes hash values for the specified operation (MSR, CPU register, or memory region) and writes the hash value back into the Results data structure. For subsequent measurements, the Inspector receives the initial hash value from the BEM for comparison.

The Inspector gathers performance metrics on each measurement to determine the cost of hash, encryption, HMAC, among its other operations. This allows the BEM to refine its cost estimates over time with a rolling average to more precisely bound the amount of time spent in SMM.

The Inspector also incorporates an automatically-generated MSR whitelist which lists the acceptable set of MSRs to monitor. This whitelist prevents the specification of non-supported MSRs on the CPU which would trigger an exception.

8.3 Attack Detection Using the Prototype

In this section, we show the Checks that can detect the rootkit and ransomware techniques described in Table 1.1. As we do not have access to the actual rootkits and ransomwares, we implemented compromises of the types of resources that they changed.

1. IDT Hooking:

   IDTR: Check command = Measure Register
   Operand = IDTR.

   IDT: Check command = Measure Virtual Memory
   Address = IDT address (obtained from provisioned IDTR)

2. CR4.SMEP Disable: Check command = Measure Register
   Operand = CR4

3. Kernel Code Injection: We first determine the kernel code sections from /proc/kallsyms.

   ffffffff81000000 T _stext
   ...
   ffffffff81c031d1 T _etext

   The resulting Check is:

   Command = Measure Virtual Memory
   Address = 0xffffffff81000000

Length = 0xc031d1.

4. System Call Hooking: We determine the location of the System Call Table from the /proc/kallsyms:

   ffffffff81e00220 R sys_call_table

   ffffffff81e01340 r

   Check command = Measure Virtual Memory

   Address = 0xffffffff81e00220

   Length = 0x1220

5. Xen Code Injection and Xen Malicious Exception Handler: Similar to Linux, Xen produces a file (xen-syms) that maps kernel symbols to virtual addresses. The range can be determined from the beginning (_stext) to the end, e.g. _einittext.

   ffff82d080200000 T _stext

   ...

   ffff82d08063b64d T _einittext

   Check command = Measure Virtual Memory

   Address = 0xffff82d080200000

   Length = 0x43B64D

8.3.1 Transient Attack Detection

In this section, we analyze EPA-RIMM's ability to detect transient attacks in Section 8.3.1. While transient attacks are not the target of EPA-RIMM, we examine EPA-RIMM's ability to detect transient attacks.

A common attack on SMM-RIMMs is called a scrubbing (or evasion) attack [81, 112]. If attackers can predict when a measurement is about to be

performed, the attackers can clean up traces of their attack such that when the measurement later occurs, the traces of the attack are no longer present. After the measurement, the attackers can again place their attack hooks.

We considered the following scenario in which an attacker places their IDTR attack for a short sub-second duration (either 0.1 or 0.5 secs) before removing it. The attacker waits a randomized duration (between 0 and 30 seconds) between transient attacks and attempts them repeatedly over an extended period of time. EPA-RIMM is loaded with an IDTR Check that it will schedule according to varying frequencies described in Table 8.1. Figures 8.1, 8.2, 8.3, and 8.4 show the detailed inspection reports for scenarios 1-4, respectively, with green depicting a measurement with no change detected and red indicating a successful detection of compromise.

The results show successful detection of all attack scenarios, although in Scenario 3, only four of 1610 attacks were detected. However, even one successful detection would raise an administrator alert. In practice, there would likely be additional EPA-RIMM measurements on the queue which would reduce the frequency of the IDTR measurement which would reduce the detection percentage. An attacker, though, could not guarantee a complete lack of detection as they cannot rule out a measurement occurring while the attack was placed.

As EPA-RIMM focuses on persistent rootkits, it does not completely address the transient attack detection issue common to all SMM-RIMMs that perform periodic measurements. However, it can address a variety of transient attack scenarios. The chance of transient attack detection is determined by the frequency of measurements, whether the changed resource is present in the set of enabled measurements, and measurement processing rate. Our results show that EPA-RIMM can detect some instances of transient attacks.

TABLE 8.1: Transient Attack Detection

Scenario	AWait(s)	HPlace(s)	MFreq(1 per Ns)	TAttacks	TAttackD
1	0..30	0.5	10	1127	**51**
2	0..30	0.1	10	3398	**37**
3	0..30	0.1	30	1610	**4**
4	0..30	0.1	1..10 (Random)	1693	**52**

Legend
AWait: Attacker wait randomized threshold (secs)
HPlace: Hack placement duration (secs)
MFreq: EPA-RIMM Measurement Frequency (1 per N secs)
TAttacks: Number of transient attacks placed
TAttackD: Number of transient attacks detected

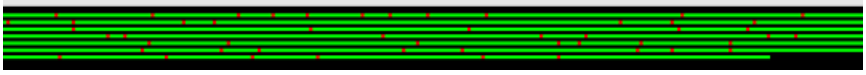

FIGURE 8.1: 0.5s compromise placement (HPlace)
1 measurement per 10 secs (MFreq)

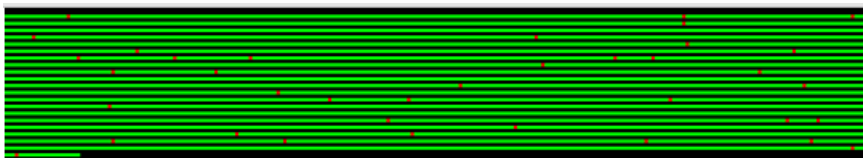

FIGURE 8.2: 0.1s compromise placement (HPlace)
1 measurement per 10 secs (MFreq)

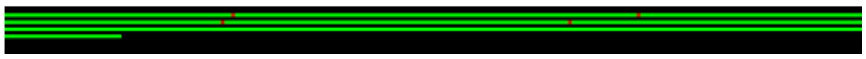

FIGURE 8.3: 0.1s compromise placement (HPlace)
1 measurement per 30 secs (MFreq)

FIGURE 8.4: 0.1s compromise placement (HPlace)
1 measurement per 1 to 10 secs (randomized) (MFreq)

8.4 Impacts on Application Performance

To measure EPA-RIMM's impact on application performance, we compared benchmark performance on the Minnowboard Turbot for four scenarios:

Baseline (no SMIs); Light with two 0.5KB memory hashes per second, Medium with four 0.5KB memory hashes per second; and eight 0.5KB memory hashes per second. We ran the Phoronix Cachebench, pybench, C-Ray, and ffmpeg benchmarks [91] and compared the throughput achieved against the no-measurement baseline. (See Figures 8.5 and 8.6.) Cachebench exercises the memory and cache - we selected the read operation; Pybench shows the system's python performance; C-Ray performs multi-threaded CPU floating point operations; and ffmpeg performs multi-threaded audio/video encoding. We observed performance degradation roughly proportional to the amount of CPU cycles spent in SMM for the single-threaded Cachebench and pybench workloads. The multi-threaded workloads, C-Ray and ffmpeg, show greater performance degradation which indicates the cumulative impact of loss of CPU cycles across a larger number of CPUs.

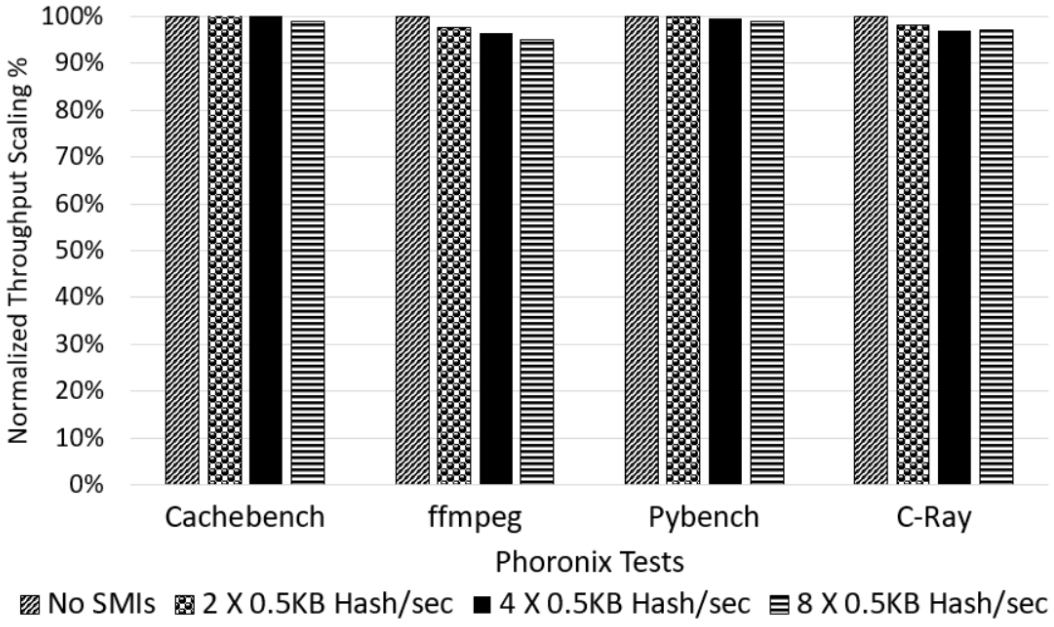

FIGURE 8.5: Application Impacts Linux

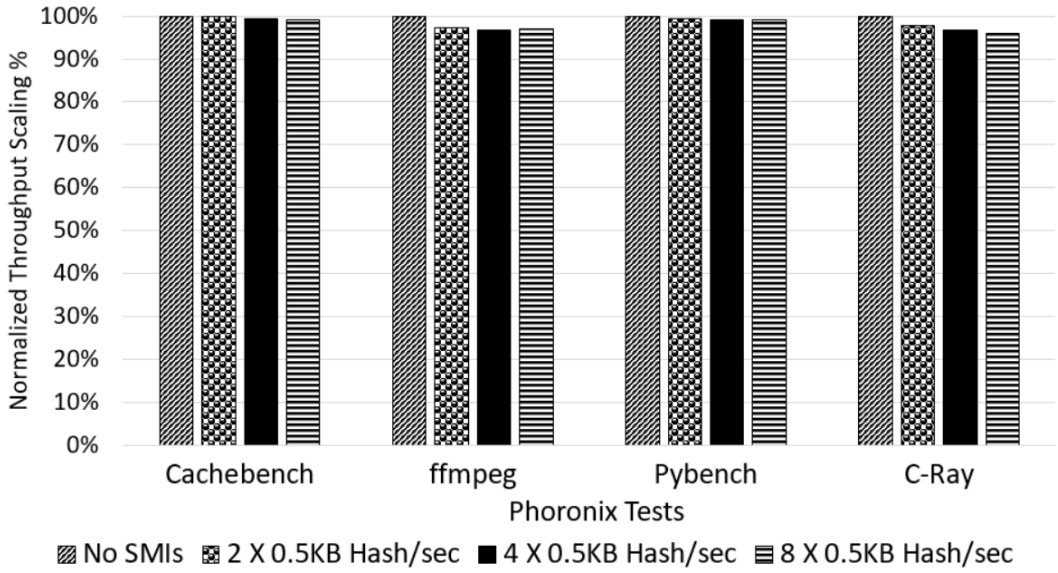

FIGURE 8.6: Application Impacts Xen

8.5 Discussion

The EPA-RIMM prototype demonstrates a functional UEFI-based EPA-RIMM measurement on two commodity open-hardware platforms: Minnowboard Turbot and UP2 boards. EPA-RIMM has also been ported by John Fastabend to use coreboot firmware [24]. With new open-hardware platforms that support coreboot [60, 96], this presents additional deployment options. Our open-source release of the EPA-RIMM software stack represents the first publicly-available SMM-RIMM prototype. Our prototype demonstrates detection of attacks involving changes to CPU control registers, kernel and hypervisor code injection, as well as the possibility of detecting some transient malware.

To evaluate EPA-RIMM's ability to provide the SMM measurement agent with enough context to detect rootkit and ransomware techniques, we developed simulated attacks that performed the same resource compromises as done in recent examples of these types of malware. EPA-RIMM detected these compromises by leveraging its provisioning phase and measurement API.

The BEM provides a wide variety of test options that enabled a wide variety of test scenarios, including measurements over the cost of cryptography operations, SMI processing costs, and varying Bin and Task sizes. The Inspector supports the flexible measurement description API and allows rootkit detection without building state into SMRAM. The HCM provides an example of the necessary communications between the Inspector and the BEM, serving as a pass-through for encrypted data flowing between these two endpoints.

The resulting application impacts of EPA-RIMM demonstrate that the resulting performance is dependent on the size and frequency of the EPA-RIMM measurements, representing a useful tuning knob. If EPA-RIMM administrators want to minimize the impact on a latency-sensitive system, system impacts can be greatly reduced by adjusting these knobs. However, if the infrastructure is under attack, these knobs present the means to perform deeper inspections to identify issues sooner.

9
Task Scheduling in EPA-RIMM

Each EPA-RIMM measurement incurs an SMI entry and exit cost. This motivates a need to make effective use of the time spent in SMM as these fixed costs are incurred for any measurement. Additionally, with a large measurement queue, maintaining a high measurement throughput is important to ensure that the Checks that the Diagnosis Manager added are processed in a timely manner.

EPA-RIMM's Backend Manager plays a key role in effectively allocating sets of work to be performed in each SMM session. As new Checks can arrive at any point in time, the precise set of Tasks to form Bins over is not known at the outset of the simulator operation. Thus, the Backend Manager must perform an online Bin formation without knowledge of what Tasks may may arrive in the future. If available Bin capacity is not effectively used, the overheads of entering and exiting SMM will be incurred but fewer measurements will be performed, hurting efficiency. This results in increased system performance degradation and slower progress through the measurement queue.

Bin size and Bin frequency also play a key role in the effectiveness of processing a large set of measurements. Increasing both of these results in faster processing of the measurement queue, however, this results in additional system overhead. The question of how to prevent starvation of older Tasks on the queue when new Tasks arrive also merits investigation. To facilitate an understanding of these questions, we developed a simulator, RIMM-SIM. This simulator allows varying key adjustable parameters to explore scheduling efficiency, speed of measurement processing, and prevention of Task starvation.

As different EPA-RIMM deployments could measure different resources, it is possible that the incoming Checks arriving at the Backend Manager may fall into various sizes. For example, memory hashes of small data structures might take minimal time whereas measurements over the entire Linux kernel could be decomposed into larger Task sizes where one Task consumes all of the available Bin capacity. For this reason, we evaluate two scenarios for Checks arriving at the Backend Manager: Uniform distribution and a normal distribution. The uniform distribution option allows examining the impacts of a wide variety of incoming Tasks where their costs do not cluster around a median value. We envision that there may also be scenarios where measurement sizes may cluster around a mean value and thus we also evaluate a normal distribution.

In Section 9.1, we provide background information for the Knapsack Problem, First Come First Serve, and a Priority queue with optional backfilling and aging features. In Section 9.2, we provide the results of our experiments that investigate the performance of the First Come First Serve and Priority Queue with backfilling and aging options. We discuss our results in Section 9.3.

9.1 Scheduling Approaches

We begin with the the Knapsack Problem, the cover the First Come First Serve algorithm, the Priority queue, and priority queues with backfilling and aging.

9.1.1 Knapsack Problem

The classic "0/1" Knapsack Problem [92] consists of a knapsack that can hold a capacity of W (weight), a set of items from 1 to N with weights w_i to w_N and values v_i to v_N. The goal is to choose an optimal subset of items with a weight no greater than W and the highest-possible value. The classic 0/1 Knapsack Problem is NP-Complete [17]. Applying this to EPA-RIMM Bin

formation, the maximum weight, W, can be set to maximum Bin size. The weights are the measurement costs and the values are the Task priorities.

In contrast, the fractional Knapsack Problem allows dividing items into fractions which could allow better utilization of the available capacity. However, this variant is not applicable to EPA-RIMM as a finite set of hash values is gathered during EPA-RIMM's provisioning phase and measuring reduced sizes in an attempt to better fill a Bin would result in different hash values that cannot be evaluated.

9.1.2 First Come First Serve

The First Come First Serve ("FCFS") algorithm is a basic algorithm that selects processes based on their order of arrival. Applied to to EPA-RIMM's Backend Manager, the Backend Manager would add Tasks from newly arrived Checks to the end of the queue and fill the Bins with Tasks from the front of the queue. This approach does not select tasks deeper in the queue to fill a Bin, even if there would be space for them.

For an example, Figure 9.1 begins in Step 1 with five Tasks on the queue, each with a cost of 33μs. In Step 2, Bin 0 is formed with four 33μs Tasks that together consist of 132μs. As the maximum Bin cost is 150μs, this does not allow the remaining 33μs Task to be included in this Bin. In Step 3, a new set of Tasks (cost is 91μs for each Task) arrives on the queue and is placed at the end. In Step 4, Bin 1 is formed with a 33μs and a 91μs Task. Step 5 shows the resulting priority queue after this Bin formation.

The benefits of this approach are that the method is simple to implement and is not prone to starvation due to processing tasks in order of arrival. A key drawback of the approach is that it can result in less optimally-filled Bins due to an inability to consider Tasks after the head of the queue for inclusion in the Bin.

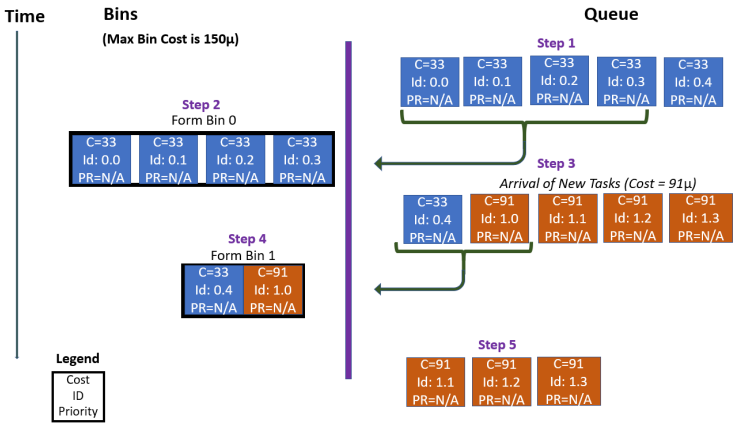

FIGURE 9.1: Bin formation with First Come First Serve

Figure 9.2 shows an example of a problematic scenario for FCFS. In Step 1, there are four Tasks on the queue with costs of 130, 50, 20, and 20μs, respectively. In Step 2, Bin 0 is formed with one Task of cost 130. We observe that the third or fourth Task in the queue with a cost of 20 would completely consume the remaining Bin capacity, however, the algorithm is not able to select either of these Tasks, thus leaving 20μs of Bin capacity unused. In Step 3, new Tasks arrive on the queue with costs of 100 and 80, respectively. In Step 4, the first two Tasks are selected for the Bin, collectively consuming 70μs of Bin capacity. The fourth Task of cost 80 in the queue would completely consume the remaining Bin capacity. However, it is not able to be selected. Thus Bin 1 is unable to use 80μs of the Bin capacity. These limitations in the First Come First Serve algorithm present clear limitations for EPA-RIMM as it can result in significant amounts of Bin capacity being unused.

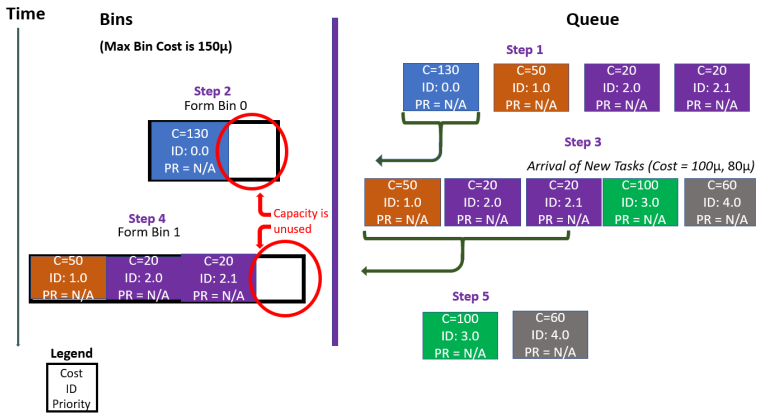

FIGURE 9.2: Problematic Case - First Come First Serve

9.1.3 Priority Queue

In contrast to the FCFS, the Priority Queue adds a priority mechanism to order tasks in the queue. The Priority Queue maintains a sorted list of Tasks in priority order and chooses the first N highest priority Tasks that will fit in the Bin. However, just as with FCFS, this method does not allow looking deeper into the queue to identify lower-priority Tasks that would help fill the Bin.

Figure 9.3 shows a priority queue. In this example, in Step 1, five Tasks of cost 33μs are added to the queue with a priority of 10. In Step 1, the first four Tasks are selected to form a Bin as they are the highest-priority Tasks that fit within the Bin. In Step 3, a new Check arrives with priority 11 and its Tasks are added to the head of the queue as they are now the highest priority Tasks. In Step 4, a new Bin is formed with a single 91μs Task at priority 11. While the 33μs Task at priority 10 would fit in the Bin, it is not considered. Step 5 shows the resulting queue.

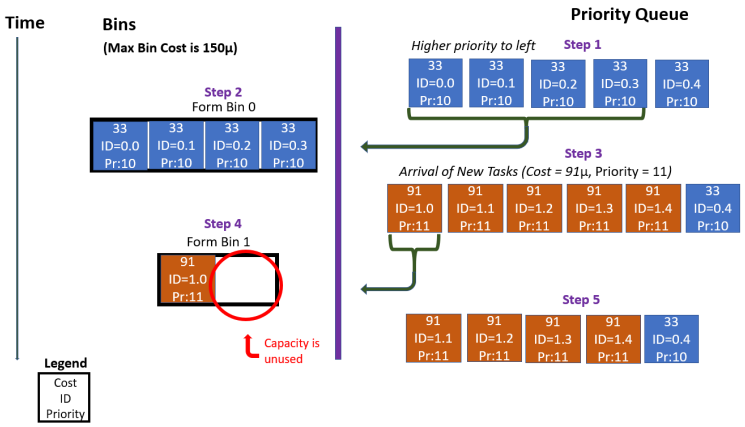

FIGURE 9.3: Priority Queue

9.1.4 Priority Queue with Backfilling

Backfilling is a method to allow looking deeper into the queue to select lower-priority Tasks to make better use of available scheduler capacity [61]. Applied to a priority queue, backfilling would provide the ability to consider a lower priority Task to better fill the Bin. This has the potential to resolve a key limitation in a Priority Queue without backfilling.

In Figure 9.4, we revisit the example previously shown in Figure 9.2, however, this time with backfilling enabled. This time, in Step 2, the Backend Manager is able to select the third task in the queue with a cost of 20μs to fill the Bin. We observe in Step 3 that the length of the queue is smaller due to processing the 20μs Task in Step 2. In Step 4, the Backend Manager is able to fill Bin 1 with an 80μs Task. Step 5 also has a smaller queue size due to making fuller use of the available Bin capacity.

While backfilling provides an additional opportunity to fill Bins to a higher capacity, it requires more Task cost comparisons to be made in the Backend Manager than if backfilling were disabled. Additionally, it does not address starvation of Tasks on the queue.

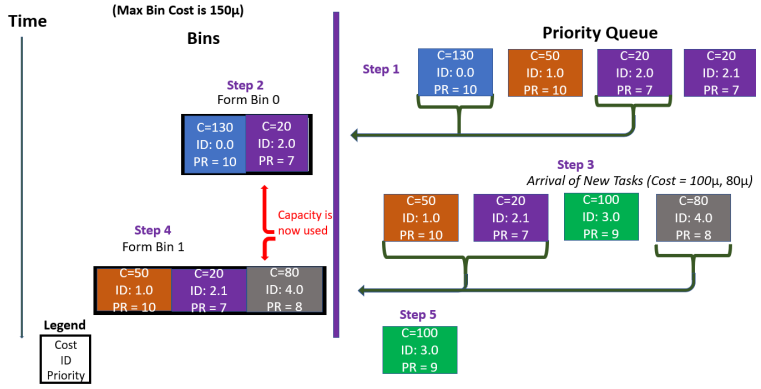

FIGURE 9.4: Applying Backfilling for fuller Bin Capacities

9.1.5 Priority Queue with Aging

While the backfilling helps produce fuller Bins, it can be experience Task starvation. Starvation can occur when newly arrived higher priority Tasks take precedence over older Tasks. To address this issue, aging provides a method of increasing the priority of older Tasks on the queue so that their priority eventually rises to a level where they will be prioritized [103].

Figure 9.5 shows an example of how aging helps avoid Task starvation. In Step 1, the priority queue consists of three Tasks with a 150μs cost and a priority of 10. In Step 2, the first Task is selected which completely fills the Bin. In Step 3, aging is first applied to existing Tasks on the queue. Then two new Tasks arrive with a cost of 100 and 50μs and a priority of 10. In Step 4, a new Bin is formed with the highest priority Task from the first set of Tasks added to the queue. In Step 5, aging is again applied to existing Tasks on the queue and the last remaining Task from the initial set of Tasks has the highest priority (12) of any Task on the queue.

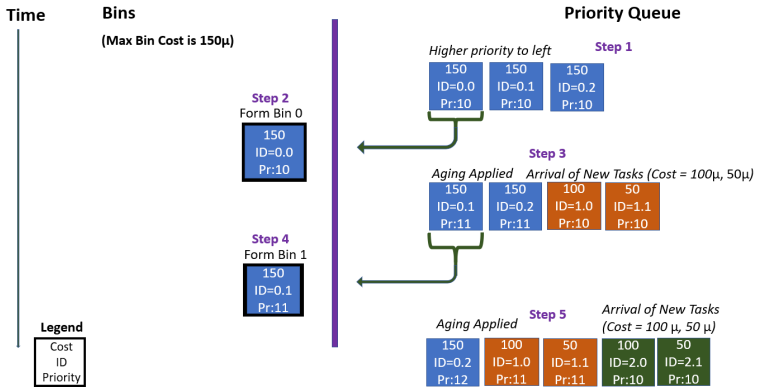

FIGURE 9.5: Priority Queue with Aging

9.2 Experiments

9.2.1 Simulation Parameters

9.2.1.1 Check Arrival Rates, Sizes, and Priorities

The arrival rate of Checks on the Backend Manager depends on the Diagnosis Manager's decisions to enqueue Checks based on a schedule or trigger mechanism. Additionally, the sizes and priorities will vary based on the resources being measured and the Diagnosis Manager's priority to accomplish the Checks. While the Backend Manager can support arbitrary Task durations, for the purpose of RIMM-SIM, we analyze two key Task distributions:

1. Uniform distribution of random-sized Tasks: This represents a scenario with a wide variety of Tasks in which any Task size is as likely to occur as any other Task size. Practical scenarios in which this could result could include monitoring a variety of read-only data structures that could each be of arbitrary sizes.

2. Normal distribution of random-sized Tasks: This represents a scenario where the majority of Task sizes will cluster around an average value with a selected standard deviation. Practical scenarios in which this

could occur would include scenarios where the size of monitored resources tends to fall within a particular range although deviations from this range are possible. For example, the Diagnosis Manager has selected a set of large fixed sized resources to monitor (e.g. fixed-sized kernel code sections along with a smaller assortment of resources that are smaller and larger in size.)

For each of these scenarios, we bound the Tasks sizes to meet practical EPA-RIMM constraints, for example, the Tasks must be greater than zero as there is no scenario in which a Task would have a cost of zero. We also constrain the Task sizes to be no greater than the Bin size, otherwise, they would require decomposition which does not add a meaningful benefit to this simulation. Thus, we examine truncated distributions that meet these practical constraints.

9.2.1.2 Inputs

RIMM-SIM supports a variety of simulation parameters to allow precise control over the desired scenario. Table 9.1 provides the parameters along with a description.

TABLE 9.1: Simulator Inputs

Input	**Description**
Simulation Duration	How many seconds to simulate
BEM CPU Frequency	CPU cycles per second
Number of Tasks in Check	How many Tasks in a newly arrived Check
Check Arrivals per second	Number of new Checks, per second
Bins per second	Bins formed/processed per second
Set of Tasks with total cost of Check	Specific checks each with a total cost in μs
Maximum Bin size	Maximum number of μs worth of work in Bin
Task decomposition target ("TDT")	Default Task cost (in μs)
Max Tasks per Bin	Limit for how many Tasks can be in a Bin
Default Task priority	The default priority of a new Task
Random Task Priority enable	If enabled, assign a random priority to new Tasks
Random Task Priority max	With random Task priority, the max Task priority
Aging enable	A knob to enable aging
Random simulation seed enabled	A knob to randomize the simulation
Seed number	If simulation seed is not random, the random seed
Backfill enable	A knob to enable backfill
New Tasks lowest priority enable	Set new Tasks at lowest priority

9.2.1.3 Outputs

RIMM-SIM outputs a variety of useful statistics at the conclusion of the simulation. These metrics are shown in Table 9.2.

TABLE 9.2: Simulator Outputs

Output	**Description**
Total number of Tasks processed	The amount of processed Tasks
Total number of Bins processed	The amount of processed Bins
Number of Check arrivals	Number of Checks added to queue
Check completion duration	Simulation cycles to complete all Tasks in a Check
Avg. queue time per Task	Avg. cycles in the queue for all processed Tasks
Max Task queue time	The max time a processed Task was on the queue
Avg., Max, Min Bin Size	The amount of μs of work packed into the Bin
Avg. Tasks in Bin	The Avg. number of Tasks across all Bins
Cumulative Task Age	Total age of all Tasks on the queue
Oldest Task Age	The age of the oldest Task on the queue
Number of waiting Tasks	The total number of Tasks waiting on queue
Per-Bin history report	List of all Bins and their Tasks
Average % of Bin capacity filled	Percentage of Bin capacity used across all Bins

9.2.1.4 Simulator Internal Details

Before beginning RIMM-SIM, the user specifies the appropriate input settings as described in Section 9.2.1.2. The main simulation loop calculates two key events: Bin Formation and Task Arrivals. The timing of these depends on the number of Check Arrivals per second and Bins per second.

A new set of Tasks is added to the queue when the simulation cycles reaches the calculated time for Task Arrival. Depending on the simulation scenario, this is accomplished in the following ways:

1. First Come First Serve: This scenario places new Tasks at the end of the queue, all Tasks have identical priorities, and there is no backfill or aging capability enabled.

2. Priority Queue: This scenario places new Tasks in a priority order, Tasks can have different priorities. Backfill and aging can be independently enabled.

When the simulation cycles reaches the time scheduled for new Bin formation, a new Bin is created. Depending on the Bin formation algorithm:

1. First Come First Serve: The Backend Manager selects the first N Tasks that will fit within the Bin, disregarding later Tasks in the queue that might be used to more fully fill the Bin. The method will consider the Bin filled when the next Task under consideration does not fit in the Bin.

2. Priority Queue without backfilling: This method will proceed through the queue in priority order and select Tasks that fit within the Bin. Once a Task is found that does not fit within the Bin, the Bin is considered closed.

3. Priority Queue with backfilling: This method will scan through the ordered list of waiting Tasks, choosing the highest priority Tasks that fit in the Bin. The algorithm has the ability to select a lower priority Task if it helps fill the remaining Bin capacity.

Once the Bin is formed, the Bin is sent and marked as completed. Then the next simulation event is calculated, e.g. Task Arrival or Bin Formation. If aging is enabled, the priority of each Task on the queue is incremented by one. The simulator then jumps to the next event and begins back in the main simulation loop. Statistics are gathered during execution to record the necessary outputs.

9.2.1.5 Evaluation of EPA-RIMM Scenarios

A variety of EPA-RIMM measurement scenarios are possible to occur in a real deployment. These could include:

1. Light-weight Checking: This scenario performs a comparatively small number of Checks on a less-frequent basis. Examples of this consist of: A portion of the Linux kernel code section measurements, once every 24

hours. In this scenario, no Task has a higher priority than another and the Check arrival rate is roughly one Check arriving per day.

2. Heavy Checking: This scenario performs rigorous checking including DM-selected subsets of Linux kernel code sections and read-only data sections, the IDT/IDTR, the GDT/GDTR, Control Registers, 30 MSRs, and reschedules these Checks as they are completed.

9.2.1.6 Bin Processing Rate vs Task Arrival Rate

RIMM-SIM allows a detailed analysis of the Bin processing rate compared to the Task arrival rate. It provides the ability to control the Bin size, Bin formation frequency, and the Task arrival rate to evaluate this important factor that controls the length of the measurement queue. The length of the EPA-RIMM measurement queue will vary based on three factors:

1. The rate that new Checks arrive at the Backend Manager. This is controlled by the Diagnosis Manager as it selects which Checks to send to the Backend Manager. In the light measurement scenario, the queue length could drain to zero as the Bin processing outpaces the Check arrival rate. However, in the heavy measurement scenario, the queue will receive an infusion of new Checks regularly which introduces the need to ensure that certain resource measurements are not starved for processing as new Checks arrive.

2. The rate that the Backend Manager sends Bins for processing: The Backend Manager will form Bins at the chosen rate. More Bins formed and transmitted per second results in additional measurements being performed, however, each Bin also incurs a cost in SMM and too many Bins per second hampers application performance so the Bin formation frequency cannot grow beyond a specified limit. The light measurement

scenario does not require frequent Bins to accomplish its measurements whereas a heavy scenario may need to schedule Bins more frequently to effectively handle the incoming rate of Checks.

3. The amount of work that the Backend Manager packs into the Bins: Adding additional Tasks to a Bin allows more work to be accomplished, however, this prolongs the amount of time in SMM, thus the size of the Bins cannot grow beyond a specified threshold. A light scenario does not need to increase the Bin size to accommodate the measurement load while the heavy scenario may need to increase the Bin size to make adequate progress through the measurement load.

9.2.1.7 Task Size Distributions

For the truncated uniform distributions in this section, we take random numbers in the range of 1 to 100 for 100μs Bin Size scenarios, 1 to 700 for 700μs Bin Size scenarios, and 1 to 1400 for 1400μs scenarios.

For the truncated normal distributions in this section, we take random numbers from a normal distribution with a statistical mean of 50μs and a standard deviation of 28.856 for the 100 Bin Size scenario. As we do not have empirical data over Task sizes from actual deployments, we approximate by calculating the mean and standard deviation from a set of random numbers from 1 to 100 (Bin Size). For the 700μs Bin Size scenario, we utilize a mean of 350 with a standard deviation of 201.67. For the 1400μs Bin Size scenario, we use a mean of 701 and standard deviation of 405 using the same method.

9.2.2 First Come First Serve

9.2.2.1 Measurement Design

In this simulation, we set the parameters according as shown in Table 9.3 to evaluate the FCFS approach in terms of scheduling, Task ages, and the

number of waiting Tasks.

TABLE 9.3: First Come First Serve Config

Parameter	**Setting**
Simulation Seconds	60
BEM CPU Frequency	3,000,000,000
Bins/sec	12
Check Arrivals/sec	10
Number of Tasks in Check	5
Max Task Size	100μs
Bin Size	100μs
Backfill Enabled	**No**
Aging Enable	No
Random Priority New Checks	No
Random Size New Checks	Yes
New Tasks Lowest Priority	**Yes**
Random Priority for New Tasks	**No**

9.2.2.2 FCFS Results

To introduce this simulation, we begin by examining the Bin Size results of one random simulation, before examining larger sets of simulation runs. We first gathered the results of the Bin sizes with the FCFS approach with truncated uniform and normal distributions in the range of 1 to 100μs.

1. Truncated Uniform Task Distribution: Figure 9.6 shows the Bin sizes. We observe that the amount of work in the Bins averages 76.7μs with a minimum of 22μs and a standard deviation of 15.4μs.

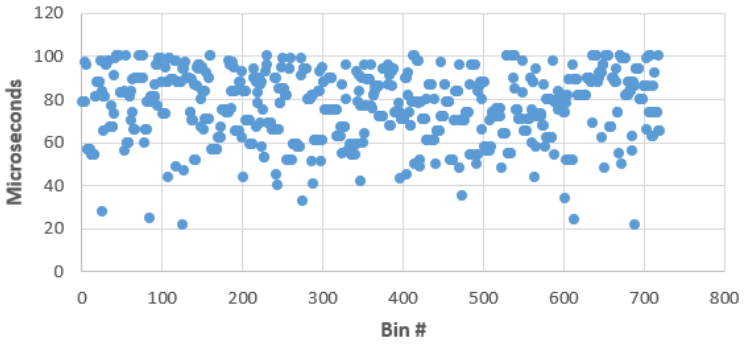

FIGURE 9.6: Bin Size Detail - Truncated Uniform Distribution, FCFS

TABLE 9.4: FCFS Results - Truncated Uniform Distribution

Result	**Value**
Avg. Tasks/Bin	1.51 Tasks
Oldest Task Age	114,660,000,000 cycles
Cumulative Task Age	109,724,400,000,000 cycles
Num. Waiting Tasks	1909.8 Tasks
Avg. Work / Bin	77.8μs

2. Truncated Normal distribution: In this simulation, we observe that the amount of work in the Bins averages 73.6μs with a minimum of 35μs.

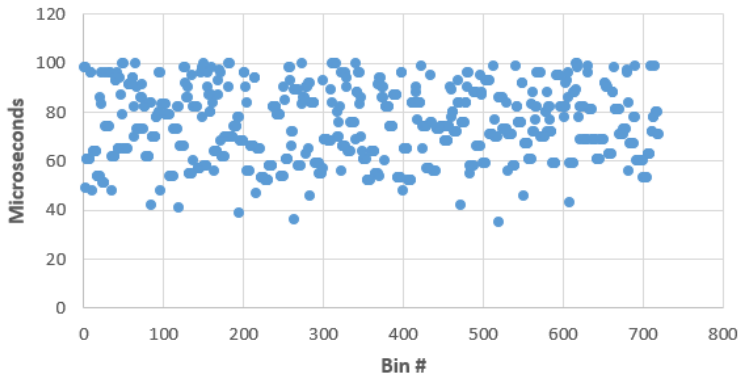

FIGURE 9.7: Bin Size Detail - Truncated Normal Distribution, FCFS

TABLE 9.5: FCFS Results - Truncated Normal Distribution

Result	**Value**
Avg. Tasks/Bin	1.51 Tasks
Oldest Task Age	114,660,000,000 cycles
Cumulative Task Age	109,653,360,000,000 cycles
Num. Waiting Tasks	1,909 Tasks
Avg. Work / Bin	73.4μs

For the light scenario, maximizing the usage of available Bin capacity is less critical and a simpler Backend Manager algorithm such as FCFS would be adequate. However, in the heavy scenario, the achieved Bin usage around 77μs out of 100μs is not ideal, which motives the need to investigate alternate approaches.

9.2.3 Priority Queue

In this set of simulation measurements, we analyze a priority queue with several options: Priority Queue ("PQ", with no backfilling or aging), Priority Queue with backfilling ("PQB"), Priority Queue with aging ("PQA"), and a Priority Queue with backfilling and aging ("PQBA"). We compare our results in two key areas to FCFS: Bin capacity and Task aging.

9.2.3.1 Measurement Design

In this simulation, we set the parameters as shown in Table 9.6.

TABLE 9.6: Priority Queue Configs

Parameter	**PQ**	**PQB**	**PQA**	**PQBA**
Simulation Seconds	60			
BEM CPU Frequency	3,000,000,000			
Bins/sec	12			
Check Arrivals/sec	10			
Number of Tasks in Check	5			
Max Task Size	100μs			
Bin Size	100μs			
Backfill Enabled	No	**Yes**	No	**Yes**
Aging Enable	No	No	**Yes**	**Yes**
Random Priority New Checks	Yes			
Random Size New Checks	Yes			
New Tasks Lowest Priority	No			
Random Priority for New Tasks	Yes			

9.2.3.2 Results for PQ, PQB, PQA, PQBA configurations

Figure 9.8 provides the uniform distribution results and Figure 9.9 provides the normal distribution results. These figures are normalized to the Backfill off, Aging off scenario. Table 9.7 and Table 9.8 provide the raw values.

Examining the average Tasks per Bin and average Work per Bin metrics, we observe that the backfill-enabled configurations have significantly higher average Tasks per Bin and Work per Bin. Aging does not help in this regard.

Examining the Oldest Task Age metric, we see that aging is the mechanism that is responsible for improving this metric and the combined backfill and aging scenario leverages the aging priority mechanism and the fuller Bins to work through the measurement queue more rapidly. This allows the oldest Task's age to drop. Similarly, the Cumulative Task Age metric shows the best performance in the combined backfilling and aging scenario as the

combinations of aging and backfill allow prioritizing older Tasks and working through these Tasks more rapidly.

One metric that at first appears to be an outlier is that the combined backfill and aging-enabled scenario does not outperform the "Backfill Enabled, Aging Disabled" scenario in the Number of Waiting Tasks and Average Tasks per Bin metrics. However, as the Average Work per Bin metric is higher for the combined backfill and aging enabled scenario, this signifies that larger Tasks have been selected in the combined scenario.

For the heavy scenario, the results of backfilling and aging are encouraging. The backfilling capability helps utilize between 93μs and 96.7μs of the 100μs Bin capacity compared to between 73.1μs and 77.5μs when backfilling was disabled. It also reduces the amount of waiting Tasks. Adding an aging capability reduces the age of the oldest Task. For the light measurement scenario, these improvements may not be required, however, for the heavy scenario, the improvements are significant.

TABLE 9.7: Backfill and Aging Results, Truncated Uniform Distributions

Backfill	**Aging**	Avg. Tasks /Bin	Oldest Task Age	Cumul. Task Age	Num. Waiting Tasks	Avg. μs Work/Bin
Off	Off	1.52	1.792E+11	1.655E+14	1909	75.8
Off	On	1.55	1.135E+11	1.067E+14	1883	77.5
On	Off	2.38	1.789E+11	1.078E+14	1287	94.3
On	On	2.20	1.013E+11	6.703E+13	1416	96.7

TABLE 9.8: Backfill and Aging Results, Truncated Normal Distributions

Backfill	**Aging**	Avg. Tasks /Bin	Oldest Task Age	Cumul. Task Age	Num. Waiting Tasks	Avg. μs Work/Bin
Off	Off	1.50	1.798E+11	1.673E+14	1919	73.1
Off	On	1.49	1.163E+11	1.118E+14	1928	73.7
On	Off	2.30	1.795E+11	1.140E+14	1343	93.5
On	On	2.15	1.002E+11	6.797E+13	1449	96.0

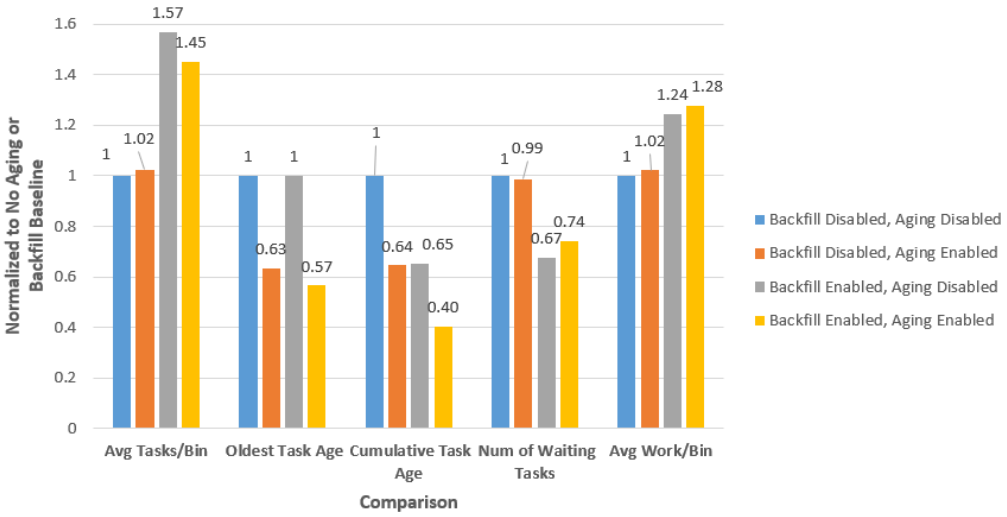

FIGURE 9.8: Comparison normalized to "Backfill Disabled, Aging Disabled" configuration - Truncated Uniform Distribution

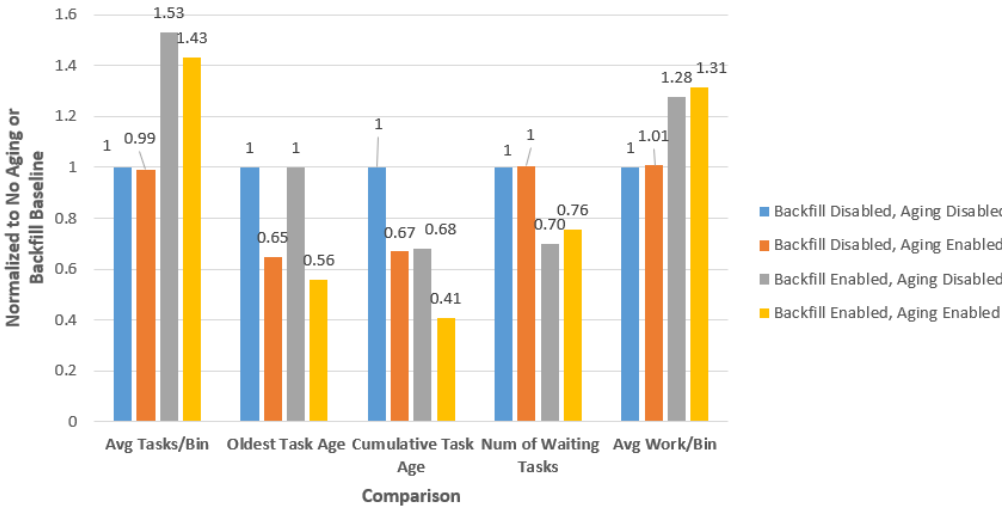

FIGURE 9.9: Comparison normalized to "Backfill Disabled, Aging Disabled" configuration - Truncated Normal Distribution

To examine Bin size in more detail for the PQB configuration, Figure 9.10 and Figure 9.11 show the truncated uniform and normal distributions, respectively. We observe that in contrast to the FCFS results, the average Bin sizes have increased (94.6μs (truncated uniform) and 94.9μs (truncated normal) for these two priority queue-based simulations) in contrast to the 77.8μs and 73.4μs results from the FCFS approach.

Figure 9.12 and Figure 9.13 compare PQB to FCFS. We observe that the average Tasks per Bin and average work per Bin have increased significantly with the PQB. This increased measurement throughput, however, does not address the age of the oldest Task.

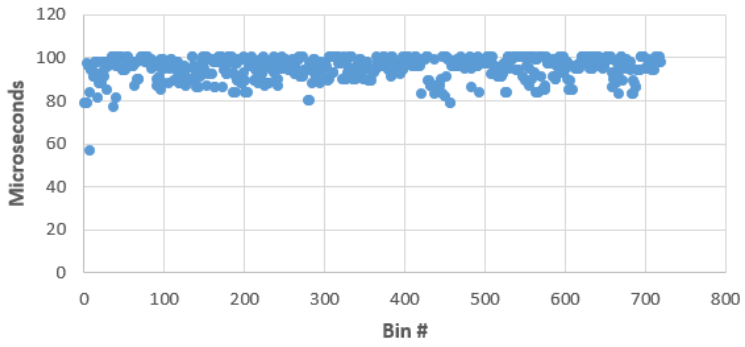

FIGURE 9.10: Bin Size Detail - Priority Queue with Backfilling - Truncated Uniform Distribution

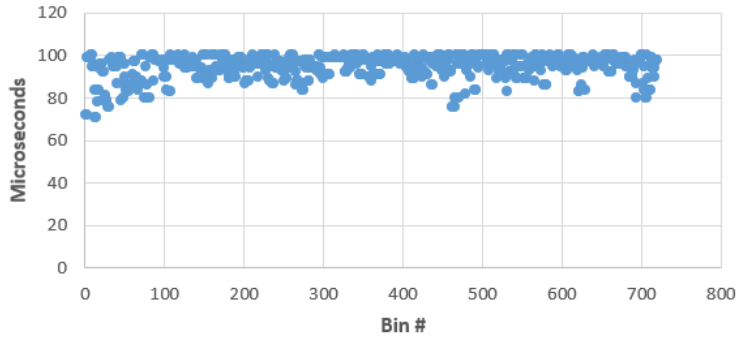

FIGURE 9.11: Bin Size Detail - Priority Queue with Backfilling - Normal Distribution

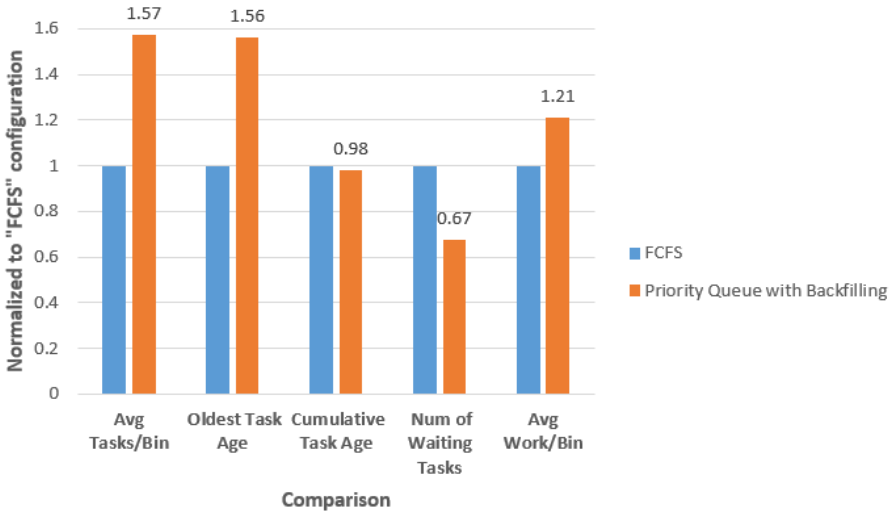

FIGURE 9.12: Priority Queue with Backfilling vs FCFS - Truncated Uniform Distribution

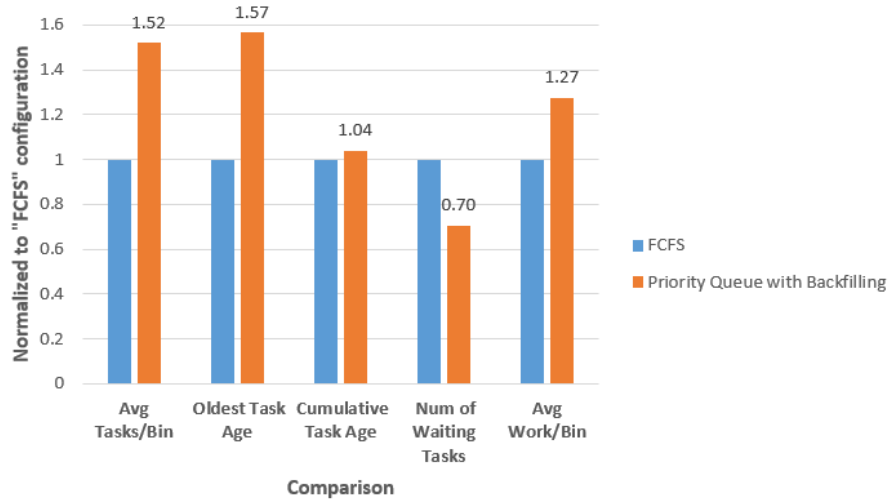

FIGURE 9.13: Priority Queue with Backfilling vs FCFS - Truncated Normal Distribution

9.2.3.3 Discussion

Comparing the results of FCFS and the Priority Queue with backfilling, we conclude that the latter fills the Bins closer to the target Bin size and provides faster processing of the measurement queue by reducing the number of waiting Tasks. The Priority Queue effectiveness was close to the maximum (e.g. filling 95μs out of 100μs). This indicates that moving to a Knapsack solving approach may not provide meaningful results that merit the additional complexity.

The Priority Queue with aging showed the ability to reduce the age of Tasks on the queue which helps avoid starvation. It is important to note that aging does not significantly impact the throughput-related metrics such as average work per Bin, number of waiting Tasks, or average number of Tasks/Bin as it is designed to facilitate the choice of older Tasks as opposed to filling Bins more fully. In the light measurement scenario, aging is not necessarily required as the lower Check arrival rate is not likely to result in Task starvation. In the heavy measurement scenario, the aging capability would help reduce the risk of older Tasks experiencing starvation as new Checks arrive.

The simulator shows that combining aging and backfilling improves both the Bin size as well as reduces the age of Tasks on the queue. This method resulted in fuller Bin sizes and avoiding starvation on the queue. While backfilling and aging can incur additional overheads on the BEM, excessive performance impacts can be mitigated by enforcing a limit on the maximum queue length or increasing the measurement throughput.

9.2.4 Bin Size Scaling

The Bin size (amount of μs of work in the Bin) is a key performance factor for EPA-RIMM. In this RIMM-SIM scenario, we analyze the impact of scaling the size of the Bin using 100, 700, and 1400μs Bin sizes with:

1. The maximum Task size equal to the Bin size which represents a scenario in which the measurements are allowed to scale upwards with the Bin size, allowing more checking in a single Task.

2. The maximum Task size set to 100μs which represents the scenario when smaller measurements were provisioned and in a time of threat, more of these smaller measurements can be readily run.

For each configuration, we examine the resulting impact on the number of Tasks/Bin, the oldest Task's age, the cumulative age of all Tasks on the queue, the number of waiting Tasks, and the average work per Bin.

9.2.4.1 Measurement Design

We list the simulator parameters used in Table 9.9 and the Bin Size distributions described in Section 9.2.1.7.

TABLE 9.9: Bin Size Scaling Config

Parameter	**Setting**
Simulation Seconds	60
BEM CPU Frequency	3,000,000,000
Bins/sec	12
Check Arrivals/sec	70
Number of Tasks in Check	5
Max Task Size	**Config 1: Bin size** **Config 2: 100μs**
Bin Size	**100, 700, 1400μs**
Backfill Enabled	Yes
Aging Enable	Yes
Random Size New Checks	Yes
New Tasks Lowest Priority	No
Random Priority New Checks	Yes

9.2.4.2 Bin Size Scaling Results - Config 1: Max Task Size = Bin Size

Figure 9.14 and Figure 9.15 provide the results, normalized to the 100μs configuration. In these results, we see the average Tasks per Bin, oldest Task age, cumulative Task age, and number of waiting Tasks hold constant as the Bin holds larger Tasks. The average work per Bin increases significantly as larger Tasks are processed.

The results show that Bin size is a useful knob to perform larger Tasks which incorporate more checking. However, it is a knob that needs to be used carefully to not preempt the system for a prolonged amount of time which would result in negative system impacts.

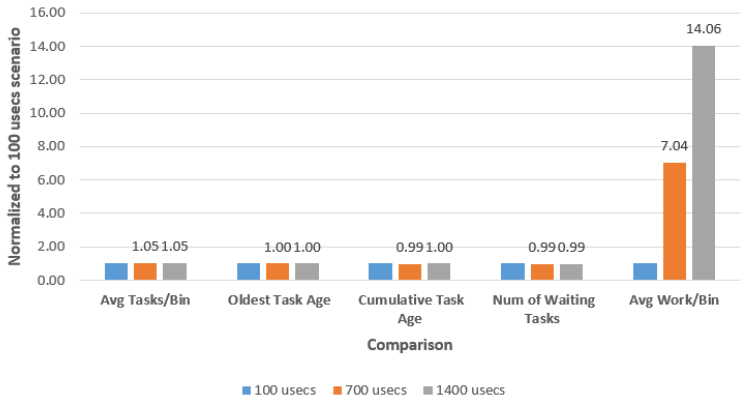

FIGURE 9.14: Impact of Bin size scaling, normalized to 100μs configuration, Uniform Distribution

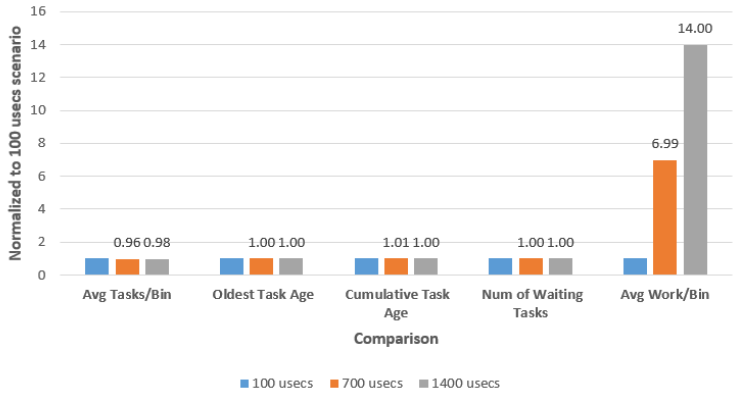

FIGURE 9.15: Impact of Bin size scaling, normalized to 100μs configuration, Normal Distribution

9.2.4.3 Bin Size Scaling Results - Config 2: Max Task Size = 100μs

In this scenario, smaller ($\leq$100μs) hashes are performed and the larger Bin sizes allow incorporating more of them in a Bin. The practical benefit of this scenario is that a single SHA hash value for each Task needs to be gathered as opposed to requiring new SHA hashes to support up to 700 or 1400μs worth of work.

The results, as shown in Figure 9.16 and Figure 9.17 show that the average number of Tasks/Bin and average work/Bin increase with the Bin size as this scenario allows running more Tasks in the Bin. Additionally, the age of the

oldest Task drops as does the cumulative Task age. The number of waiting Tasks also drops as more Bin capacity is available to process them.

This scenario shows its value in the heavy measurement case. To accommodate the high rate of Checks arriving, the ability to increase the Bin size allows faster progress through the measurement queue. The choice of whether to utilize Config 1 (Task max size scales with Bin Size) or Config 2 (Task max sizes set to the lowest Bin size) depends on the tolerance to create and maintain provisioned values over larger resources that can be used when required. If the expectation is that increasing the Bin size is a rare occurrence, Config 2 is adequate and avoids the need to generate and maintain a separate set of provisioned values. Config 1 provides the ability to perform larger measurements as required to get results more readily.

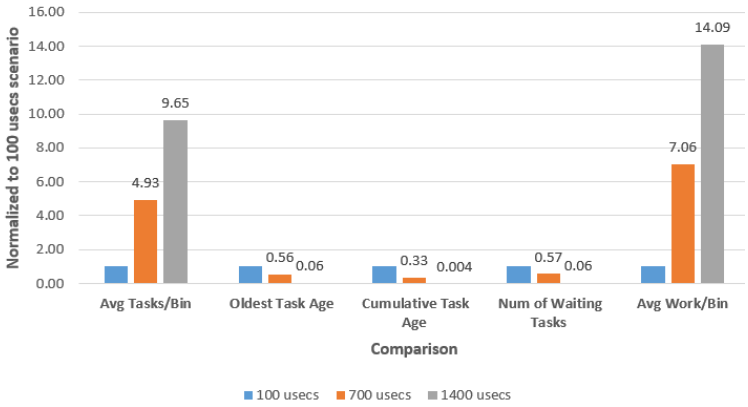

FIGURE 9.16: Impact of Bin size scaling, normalized to 100μs configuration, Uniform Distribution, Max Task Size 100μs

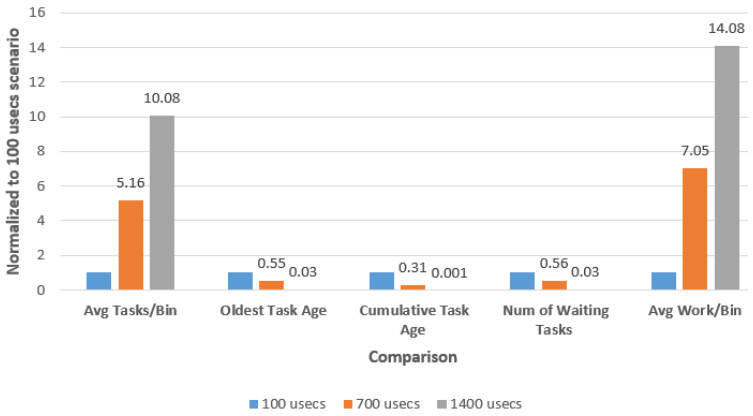

FIGURE 9.17: Impact of Bin size scaling, normalized to 100μs configuration, Normal Distribution,, Max Task Size 100μs

9.2.5 Bin Frequency Scaling

Bin frequency (e.g. Bins per second) represents another important knob for accomplish more measurements in less wall-clock time, besides increasing the Bin size. In this simulation run, we examine the impact of Bin frequencies of 12, 48, 96 a second while holding Bin size and Check arrival rate constant.

9.2.5.1 Measurement Design

Table 9.10 provides the simulation parameters.

TABLE 9.10: Bin Frequency Scaling Config

Parameter	**Setting**
Simulation Seconds	60
BEM CPU Frequency	3,000,000,000
Bins/sec	**12,48,96**
Check Arrivals/sec	70
Number of Tasks in Check	5
Max Task Size	100μs
Bin Size	100μs
Backfill Enabled	Yes
Aging Enable	Yes
Random Size New Checks	Yes
New Tasks Lowest Priority	No
Random Priority New Checks	Yes

9.2.5.2 Bin Frequency Results

The results in Figure 9.18 and Figure 9.19 show that the increased Bin frequencies successfully drop the age of the oldest Task on the queue and the total age of all of the Tasks on the queue at the end of the simulation. The number of waiting Tasks also drops as the increased Bin frequency allows faster progression through Tasks on the queue. The results also show a drop in the number of Tasks per Bin however Work per Bin remains roughly constant. This indicates that the increased Bins are taking larger Tasks in the measurement queue. The Tasks per Bin and Work per Bin results are dependent on the makeup of the items in the measurement queue. Each Bin incurs a set cost as the CPU cores are completely consumed by the SMI processing, thus 12, 48, 96 Bins with a cost of 150μs each (accounting for SMI entry and exit costs) would consume 0.18%, 0.72%, and 1.44% of the available CPU cycles. Workloads that perform high computation would benefit from less frequent Bin/second rates.

The choice of how to set the Bin frequency and size depends on a few considerations:

1. The cost of SMI entry and exit. If these factors are high, then larger Bins would be more efficient than increasing Bin frequency as the SMI entry/exit costs are incurred on a per-Bin basis.

2. Tolerance to provision hashes of different sizes, e.g. if gathering a set of hashes that fit within 100, 700, and 1400μs Bins is acceptable, then these larger sizes can allow more checking to occur in a single measurement. However, if only one set of provisioned values is needed, utilizing the smallest common denominator of Task size (e.g up to 100μs Task size)

would be preferable and performing more of these measurements in a single Bin would allow more checking to occur in the Bin.

3. Sensitivity to SMI latency: If the software on the monitored system is more sensitive to prolonged SMI latencies, more frequent Bins would be preferable to larger Bins as the more frequent (but shorter) disruptions would delay latency-sensitive code for a shorter duration.

The heavy measurement scenario would benefit from the Bin frequency scaling to allow faster progression through the measurement queue. However, this knob needs to be balanced against the above considerations.

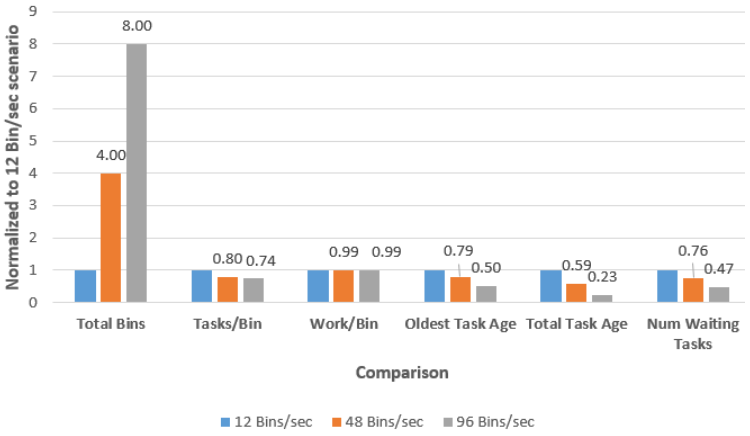

FIGURE 9.18: Bin Frequency Scaling Comparison normalized to "12 Bins/sec" configuration

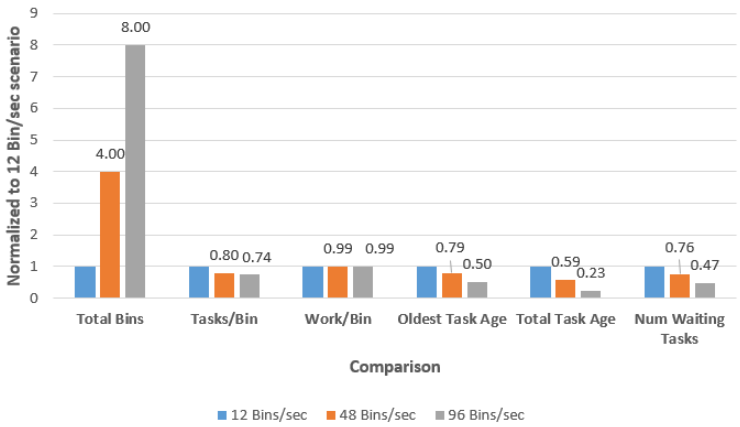

FIGURE 9.19: Bin Frequency Scaling Comparison normalized to "12 Bins/sec" configuration

9.2.6 Check Arrival Rate

The Check Arrival Rate represents how quickly Tasks from newly-arrived Checks are added to the queue. This factor has queue management implications. If the Check Arrival Rate is lower than the Bin processing rate, the measurement queue drains over time and when it is empty, no new Bins would be formed. If the Check Arrival Rate exceeds the Bin processing rate, then the queue length grows. A larger measurement queue requires more work on the Backend Manager to implement backfilling and aging as there are more Tasks to consider for inclusion and also update aging for.

9.2.6.1 Measurement Design

In this simulation run, we vary the arrival rate of new Checks using rates of 200, 400, 800 Checks a second. We fix the Bin size at 100μs and Bin frequency at 96 Bins/second. The number of Tasks in a Check is fixed at five. Table 9.11 provides the configuration settings for this run.

TABLE 9.11: Check Arrival Config

Parameter	**Setting**
Simulation Seconds	60
BEM CPU Frequency	3,000,000,000
Bins/sec	96
Check Arrivals/sec	**200, 400, 800**
Number of Tasks in Check	5
Max Task Size	100μs
Bin Size	100μs
Backfill Enabled	Yes
Aging Enable	Yes
Random Priority New Checks	Yes
Random Size New Checks	Yes

9.2.6.2 Check Arrival Rate Results

In Figure 9.20 and Figure 9.21, we observe that the fixed rate of Bin processing is insufficient to prevent increases in the age of Tasks on the queue as both the age of the oldest Task on the queue and the cumulative age of Tasks on the queue increase significantly. Similarly the number of waiting Tasks also increases as Bins cannot be formed quickly enough or large enough to process them given the configuration constraints.

One mitigation technique for this scenario of frequent Check arrivals that outpace Bin processing rates, is to increase the Bin size and/or frequency. However, due to system impact concerns, there are limits to how much these knobs can be adjusted upwards. The BEM could also be enhanced to monitor the Check Arrival Rate and the Bin processing rate to provide feedback to the DM to reduce the Check Arrival Rate.

The light scenario will not be affected by the rate of Check arrivals as they are relatively infrequent. However, the heavy scenario will result in a high Check arrival rate. To the degree possible, the BEM can increase the Bin size and frequency to handle the incoming rate of Checks, however, a rate

that is too large to be feasibly support will need to be throttled otherwise the monitored system is not able to perform its required work.

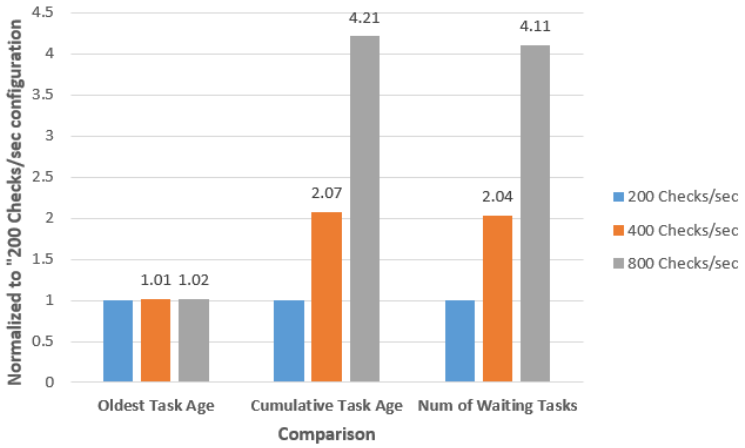

FIGURE 9.20: Check Arrival Rate impact on Task Age and Number of Waiting Tasks, [200,400, and 800 Checks/sec], Truncated Uniform Distribution

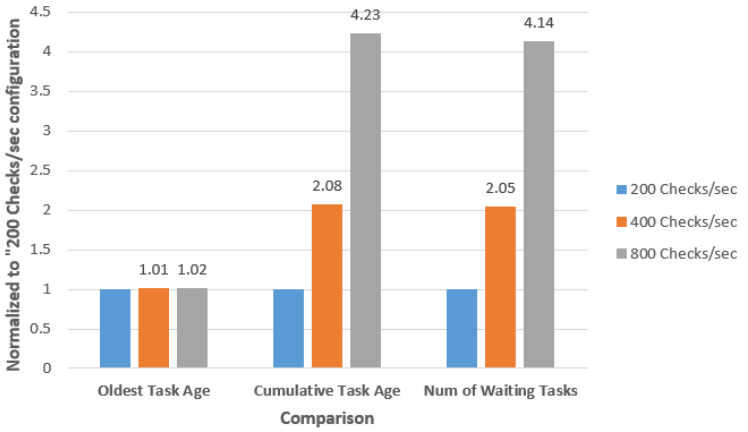

FIGURE 9.21: Check Arrival Rate impact on Task Age and Number of Waiting Tasks, [200,400, and 800 Checks/sec], Truncated Normal Distribution

9.3 Discussion

RIMM-SIM provides a controlled simulation environment to allow examining the impact of different Scheduling mechanisms such as First Come First Serve,

a Priority Queue, a Priority Queue with backfilling, a Priority Queue with aging, and a Priority Queue with aging and backfilling.

For our analysis, we looked at two Task distributions: uniform and normal. Ultimately, the results of the two distributions did not differ significantly. However, the analysis from each distribution provides finer-grained data for scenarios where: 1. There is little ability to predict the size of incoming Tasks, as is the case with the uniform distribution, and 2: The Tasks cluster around a statistical mean.

We observed that the FCFS and Priority Queue without backfilling resulted in smaller Bin sizes than the Priority Queue with backfilling. However, a Priority Queue with backfilling approach alone did not address the Task starvation problem. For that, we enabled aging which provided measurable impacts in reducing the cumulative age of Tasks on the queue and the age of the oldest Task. In both the light and heavy scenarios, the Priority Queue with backfilling and aging achieved high Bin utilization without starvation. This less-complex solution, compared to knapsack, helps reduce the computational requirements on the Backend Manager.

Even with this scheduling approach, varying the knobs of Bin Frequency and Bin size may be required. These parameters when adjusted upwards allow more work to be accomplished, however, at the cost of increased system performance. The choice of larger Bins vs more frequent Bins depends on SMI entry and exit, ability to tolerate prolonged SMIs, and desire to provision and maintain larger provisioned measurements. The goal of making faster progress through the measurement queue must be tempered with the goal to maintain adequate system performance. The rate of Check arrivals also needs to be balanced against the Bin processing rate as the measurement queue can grow if Check arrivals outpace Bin processing and drain completely if Bin

processing outpaces Check arrivals over time.

10
EPA-RIMM Bench

The performance of EPA-RIMM implementations will differ on various systems. Without a tool to measure and compare performance across systems, it is difficult to determine their system impact and potential benefits from improving aspects of their overheads.

This section describes the creation of the first benchmark for an SMM-RIMM, EPA-RIMM Bench. This benchmark provides the ability to compare the performance on different systems for actions that are representative of EPA-RIMM. The performance achieved can very significantly due to important variables including SMI entry and exit costs, CPU hashing performance, CPU support for acceleration of cryptographic operations, and the cryptographic library used. In Section 10.1, we provide an introduction to EPA-RIMM Bench. We provide a description of our performance model that guides EPA-RIMM Bench in Section 10.2. In Section 10.3, we describe the design of the benchmark and we share benchmark results in Section 10.4. We discuss our benchmark results in Section 10.5.

10.1 Introduction

Runtime integrity measurement performance may differ significantly on various systems due to a variety of factors. These factors can include hardware differences, firmware, the measurement agent, and the set of measured resources.

Hardware can play a significant role in EPA-RIMM performance. Server systems with higher CPU thread counts can extract additional measurement performance by leveraging parallelism [24]. Architectural improvements in CPUs, for example, crypto acceleration such as AES-NI [38], can improve

encryption. Similarly, Intel's SHA extensions can also improve hashing performance [47]. Firmware is another important factor that impacts performance. The SMI entry and exit costs can very due to firmware revision or codebase. UEFI's CPU rendezvous for threads entering SMM could incur greater overheads on CPUs with a high number of CPU threads. The size of the hash operation also impacts the time spent in SMM.

The measurement implementation also influences performance. Implementations that reduce measurement agent costs can likewise reduce the amount of time spent in SMM. These reductions could come from streamlining the measurement agent's code itself or by switching to more efficient hashing or encryption algorithms. The measurement scheduling also presents an important performance knob. As each SMI transition incurs a performance cost, this factor can result in additional CPU load.

Ultimately, EPA-RIMM developers benefit from a detailed understanding of the performance overheads of their hardware, firmware, measurement scheduling mechanism, and Inspector. EPA-RIMM Bench quantifies the entire time required for measurements along with detailed breakdowns over the factors that contribute to the performance achieved.

10.2 Performance Modeling

To allow understanding the overheads of the EPA-RIMM measurements, we created a performance model methodology that allows expressing the various SMM-based measurement overheads from the point of SMI interruption to the return from SMM to the operating system. Itemizing the costs of SMI processing by category allows the ability to substitute parameters based on different system particulars or a hypothetical CPU and comprehend the impact on measurements.

10.2.1 EPA-RIMM Performance Model

$\mathbf{T_m}$ is the total measurement time consisting of transition times into and out of SMM as well as the time spent in the Inspector as shown in Equation 10.1. Figure 10.1 provides a graphical representation of the flow.

$$T_m = T_{entry} + T_{work} + T_{exit} \tag{10.1}$$

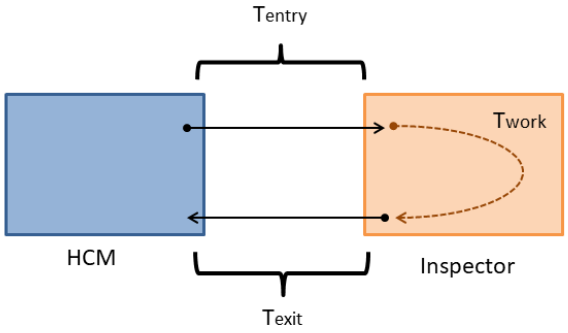

FIGURE 10.1: EPA-RIMM Round Trip Time Components.

$\mathbf{T_{entry}}$ is the context switch time to enter the SMI Inspector. $\mathbf{T_{exit}}$ is the time to transition out of SMM. T_{entry} and T_{exit} are influenced from the platform and firmware.

$\mathbf{T_{work}}$ is the total time to accomplish the measurement which consists of Bin decryption ($T_{decrypt}$), a measurement hash (T_{hash}), Results encryption ($T_{encrypt}$), HMAC comparison ($T_{HMAC\_compare}$), HMAC creation (T_{HMAC_create}), verifying the Bin signature ($T_{signature\_verify}$), creating the Results signature ($T_{signature\_create}$) and other overheads (T_{other}) consisting of all other Inspector overheads such as data copies, memory comparisons, and security checks over Bin placement.) Equation 10.2 shows these components.

$$\begin{aligned} T_{work} = {} & T_{decrypt} + T_{encrypt} + T_{hash} + T_{HMAC\_compare} \\ & + T_{HMAC\_Create} + T_{signature\_verify} + T_{other} \end{aligned} \tag{10.2}$$

Given a threshold $\mathbf{T_{max}}$ for total time in SMM, our upper bound for T_{work} is shown in Equation 10.3:

$$UpperBoundOfT_{work} = T_{max} - T_{entry} - T_{exit} \tag{10.3}$$

10.3 Benchmark Design

This section describes the implementation of EPA-RIMM Bench and how it obtains the necessary metrics to report.

10.3.1 Generating a workload

EPA-RIMM Bench performs memory hashes of varying sizes e.g. 0x100, 0x1000, 0x10000 bytes. The user is able to specify larger hash measurements if desired. The benchmark gathers and averages the costs of the EPA-RIMM performance model and reports their values for the given CPU and hash size.

10.3.2 Measuring Times

EPA-RIMM Bench leverages timestamps before and after SMI generation to calculate T_m which contains the entire cost of the SMI measurement.

We calculate T_{entry} by measuring the time from the SMI generation to the CPU arrival in the Inspector. We calculate T_{exit} as the minimum of the time from the Inspector's exit to returning to the HCM. For additional accuracy over T_{exit}, we expose an optional knob to disable interrupts around the SMI generation to avoid including any interrupt processing in the time measurement. As interrupts can be processed directly after exiting from SMM, their processing can result in the ending timestamp for the measurement being delayed. In some scenarios, this knob can result in some system instabilities, therefore, it may not be usable on all configurations. EPA-RIMM

Bench reports whether interrupts were disabled during the run for proper comparison.

EPA-RIMM Bench leverages time stamp counters in the Inspector placed at the beginning of the encryption, decryption, and hashing operations and at the end of the operations. The time to perform these operations is calculated by subtracting the starting timestamp from the ending timestamp. The timestamps are returned in the Results data structure. As encryption is used, there is one complication that needed to be resolved: two of the timestamps occur after the Results data structure is encrypted by the Inspector and writing the data to the encrypted data structure would corrupt the data. To resolve this complication, we store the cost of the encryption and the ending timestamp of the Inspector in the global memory of the Inspector. We then return these two values in the Result data structure in the subsequent EPA-RIMM measurement session. We take advantage of being able to write into the Results data structure before it is encrypted. EPA-RIMM Bench's post-processing scripts sort all collected timestamp which properly places the timestamps in the proper chronological order.

10.4 Benchmark Results

In this section, we provide the results from two systems on EPA-RIMM Bench. The first system is the dual-core Minnowboard Turbot and the second system is the UP2 board.

For the UP2 board, we examine both one and four core-enabled scenarios. Running one core on the UP2 board allows for a higher frequency core (2.5 GHz) as the available processor power and thermal budget can be entirely used by a single core. It also incurs less cache contention due to a single CPU having access to the cache. Running four cores on the UP2 board limits the

CPU frequency to 1.66 GHz and incurs cache sharing. As the EPA-RIMM Inspector is single-threaded, having additional CPU cores does not benefit the Inspector's measurement times and can incur SMM rendezvous times as all CPUs are collected in SMM before invoking the Inspector.

10.4.1 Hash Input Size Scaling

We begin by analyzing the impact of scaling the hash input size for three sizes: 0x200, 0x1000, and 0x10000 bytes for each configuration. Figure 10.2 shows the results. The UP2 1 core measurement takes the least amount of time. As the UP2 board features a more powerful CPU using Intel's Goldmont CPU architecture compared to an older Intel Silvermont architecture used on the Turbot, the results indicate the potential benefit from higher performance CPUs. None of the configurations achieve the ($\mathbf{LimitSMI_{BITS}}$). All configurations meet the more relaxed ($\mathbf{LimitSMI_{Empirical}}$) bound for the 0x200 and 0x1000 measurements. But, as the hash input size grows to 0x1000 bytes, only the UP2 1 core configuration meets this bound.

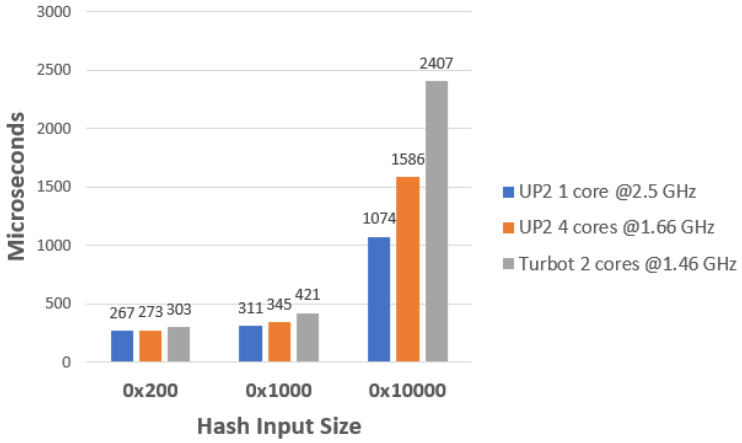

FIGURE 10.2: EPA-RIMM Bench - Bin Costs

10.4.2 Bin Cost Breakdown

From EPA-RIMM Bench's tracing capability, we observe in Figure 10.3 that the 1CPU UP2 config has a reduced HMAC, encrypt and decrypt, and hashing costs which suggests that the additional frequency and lack of cache contention provide measurement improvements for EPA-RIMM. The slower Minnowboard Turbot has higher costs across all trace points except T_{entry}.

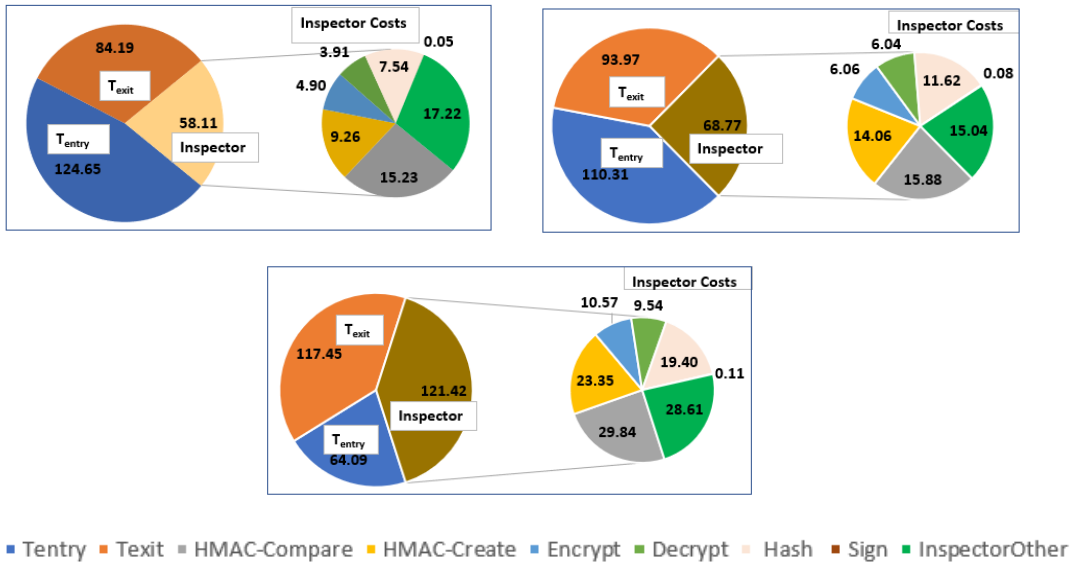

FIGURE 10.3: 0x200 Hash Input Size - Bin Cost Breakdown
(a) (Top left) 1 Core UP2 @2.5GHz
(b) (Top right) 4 Core UP2 @1.66 GHz
(c) (Bottom) 2 Core Turbot @1.46 GHz

For the 0x1000 hash input size as shown in Figure 10.4, while the hash costs grew, the other overheads remain constant compared to the 0x200 byte measurement. As the encrypt, decrypt, and HMAC operations are over the Bin, the sizes do not grow with the size of the measurement. The T_{entry} and T_{exit} costs also do not vary with the size of the measurement.

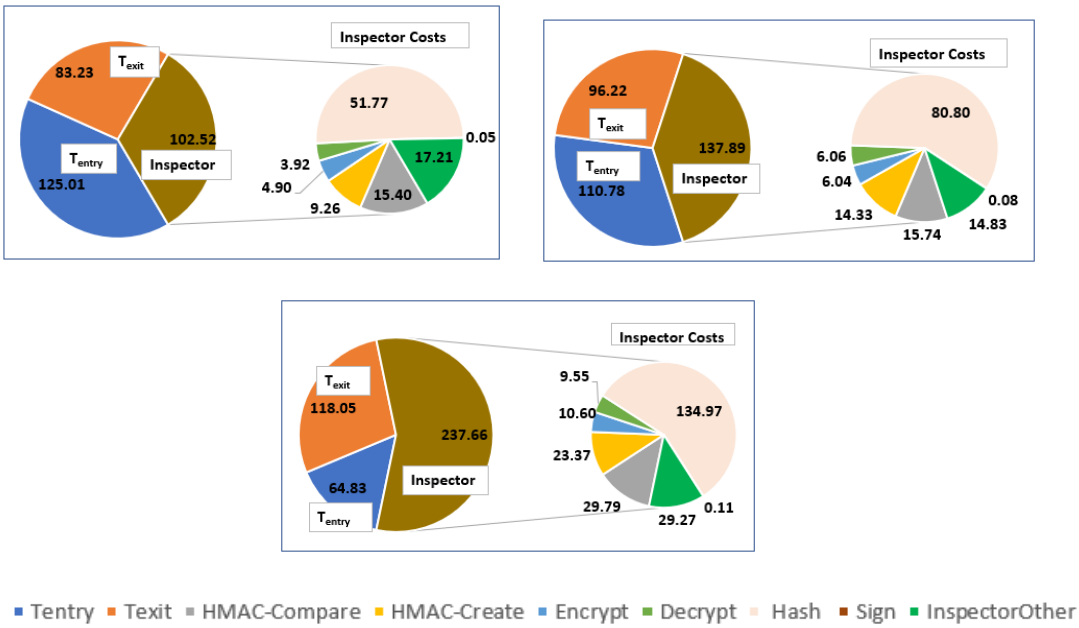

FIGURE 10.4: 0x1000 Hash Input Size - Bin Cost Breakdown
(a) (Top left) 1 Core UP2 @2.5GHz
(b) (Top right) 4 Core UP2 @1.66 GHz
(c) (Bottom) 2 Core Turbot @1.46 GHz

For the 0x10000 hash input size as shown in Figure 10.5, this measurement continues the trend of hash costs overwhelming all other components. As hash costs are the only cost that scales with hash input sizes, the other portions of the Bin cost remain fixed.

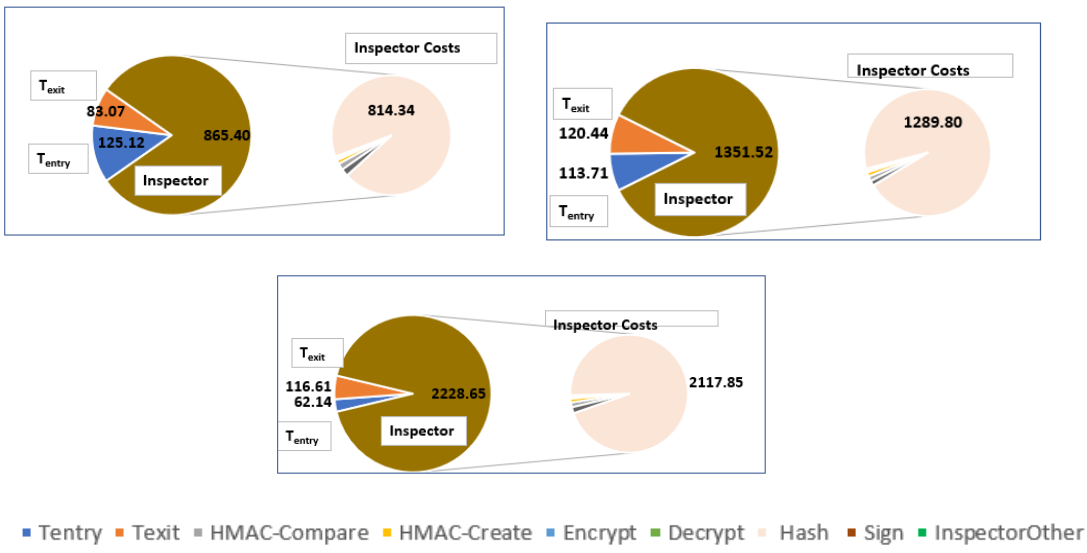

FIGURE 10.5: 0x10000 Hash Input Size - Bin Cost Breakdown
(a) (Top left) 1 Core UP2 @2.5GHz
(b) (Top right) 4 Core UP2 @1.66 GHz
(c) (Bottom) 2 Core Turbot @1.46 GHz

10.5 Discussion

EPA-RIMM Bench provides the ability to compare important performance characteristics of EPA-RIMM operations across different systems. This allows determining if a given configuration can support a hash size within the chosen SMI latency bounds. It also provides a detailed cost breakdown for the time spent in SMM and transitions to and from SMM. We extend the underlying data into a performance model for SMM-RIMMs providing the ability to examine key parameters and results.

The results show that while generational improvements among CPU versions increase the performance, none of the tested EPA-RIMM configurations hit the more stringent $\textbf{LimitSMI}_{\textbf{BITS}}$ limit. All of the tested configurations could hit the less stringent $\textbf{LimitSMI}_{\textbf{Empirical}}$ for the 0x200 and 0x1000 hash input sizes. As hash input sizes grow, they consume the majority of EPA-RIMM's SMM overhead as the other costs are fixed. The performance model

that underpins EPA-RIMM Bench allows quantification of the various overheads and can serve as the basis of estimations for the potential impact in improvements or degradations in any of the listed categories.

11
Conclusions

At the outset of this work, there was no awareness of concerns over SMI latency and SMM-RIMMs or published studies of system impact when SMI latency. To address this limitation, we devised four methods of generating suitable SMI loads and created methods of quantifying the resulting system impacts. This work has resulted in: 1. Recognition that our work was "the first to experimentally expose the performance implications of Intel's System Management Mode (SMM) [35]", leading to a broader awareness of concerns over SMI latency and 2. SMI latency concerns addressed by HP for their SureStart SMM monitoring software, citing our work [18].

We had the early observation that the SMI latency guideline ($\mathbf{LimitSMI_{BITS}}$ did not leave a large amount of time for system inspections. This motivated us to determine where the breaking point was for our test systems. With this study, we found that there was additional headroom ($\mathbf{LimitSmi_{Empirical}}$) above the guideline to perform measurements as long as we did not exceed this upper limit. While raising awareness of SMI latency concerns for SMM-RIMMs was impactful, we were encouraged by reviewer feedback to build upon these insights to actually address the SMI latency concerns for SMM-RIMMs.

This feedback encouraged us to build upon the insights and methodologies we had created to address the performance issue for SMM-RIMMs. Another factor proved motivational for our work: the SMM landscape had changed since the outset of this work. With growing concerns over broad SMM access to the system, previously-developed SMM-RIMMs were not in sync with current trends. This encouraged us to take a fresh look at several other persistent limitations for SMM-RIMMs that worked against our ultimate goal

of demonstrating a usable SMM-RIMM with capabilities to address security, extensibility, and performance concerns.

11.1 Summary

Constructing an effective and performance aware SMM-based RIMM requires careful consideration of many aspects including privilege of the measurement agent, handling a semantic gap, addressing SMI latency, scheduling a potentially large queue of measurements, and enabling extensible measurements. Providing a publicly-available implementation helps researchers build upon this framework.

Addressing the requirements required careful analysis and design to avoid conflicts. For example, the quickest way to proceed through the measurement queue would be to run all measurements immediately as they arrive. However, this would incur unacceptable SMI latencies and system impacts. Enabling extensible measurements must be done without reducing the security of the measurement agent as a new interface is opened.

Leveraging more rigorous cryptography and hash algorithms could result in enhanced resilience to certain attacks, however, this would also increase the time required to operate in SMM. The use of an STM also constrains the SMM measurement agent to prevent it from operating with higher privileges than neccessary. Improving the security design was necessary to satisfy "C1: SMM-RIMM Security".

Proposed SMM-RIMMs did not feature ways to vary the specific measurements to be performed at each inspection. EPA-RIMM's measurement API allows dynamically varying the set of monitored resources to reduce the ability of rootkits to adapt to static measurements. We added support for

fundamental runtime integrity measurement commands that provide building blocks to effectively detect rootkits. By allowing specification of one or more of these measurement commands in a single measurement session and varying the operand, varied sets of measurements can be accomplished. EPA-RIMM's provisioning phase provides guidance to the SMM measurement agent and does not require building details of system state in to SMM which addresses the semantic gap ("C2: SMM and OS Semantic Gap") between SMM and its lack of knowledge about an operating system's internal layout.

With our EPA-RIMM architecture, we were able to demonstrate successful detection of classes of operating system and hypervisor-based rootkits at reasonable performance impacts. Additionally, the mechanism we developed to decompose large measurements into smaller measurements to achieve performance targets also facilitated a useful tuning knob where the amount of measurements could be increased or decreased dynamically based on policy or preference. Check decomposition is key to addressing "C3: SMM-RIMM Performance".

We did not expect that our method of decomposing large measurements would also naturally lead to facilitate extensible measurements. By allowing Tasks to precisely specify the measurement to be performed, their data structure could be readily extended with new commands and operands to perform new measurements with minimal effort across the EPA-RIMM software stack. The Check description API resolves "C4: SMM-RIMM Measurement Variability".

To verify that our design and approach works on actual systems, we constructed the EPA-RIMM prototype on two open hardware platforms where we have the ability to modify the firmware source code. On this prototype, we demonstrated successful detection of kernel and Xen hypervisor code

injection, interrupt descriptor table hooking, supervisor mode execution prevention disabling, and system call hooking. To provide a reference example of a functional SMM-RIMM software stack, we released our prototype as open source source, constituting the first publicly available SMM-RIMM. This resolves "C5: SMM-RIMM Code Availability".

The ability to run actual rootkit detections from our prototype demonstrated the detection effectiveness as well as provided a configurable testbed to evaluate the performance of the approach. This showed us that our modest hardware, we were able to perform these detections within $\mathbf{LimitSmi_{Empirical}}$. Faster processors, optimized firmware implementations, and increased encryption performance may give the ability to read the more stringent limit ($\mathbf{LimitSMI_{BITS}}$).

To provide a mechanism for understanding the impacts of performance-sensitive EPA-RIMM flows, we developed EPA-RIMM Bench which incorporates a performance model and benchmarking capability. With the performance model, we can determine the precise impact of improvements or degradations in portions of these flows. The EPA-RIMM Bench tool provides the ability to quantify these portions of the flow on different hardware to allow empirical measurements over the entire EPA-RIMM flow.

A large set of measurements on the queue requires effective scheduling. Our simulation results show that adding aging to entries on the queue reduces the age of Tasks on the queue, helping prioritize older Tasks before newer Tasks. In circumstances where dependent actions can be identified (e.g. responses to changes in the CR0 register). Triggers can reduce the amount of checking needed by performing operations that could otherwise be avoided.

Making fuller use of available Bin capacity avoids unnecessary SMM transitions. Our simulation results examining the impact of backfilling demonstrate that backfilling significantly increased Bin utilization. We found that the Priority Queue with backfilling and aging results in additive benefits that results in fuller Bins and reduced Task age on the queue. While lighter measurement scenarios may not need these capabilities, they provide major improvements for heavy measurement scenarios.

11.2 Future Work

There are several key areas for future work pertaining to EPA-RIMM. The first of these is exploring methods to parallelize the SMM-based measurements. UEFI SMM code does not yet support multi-threading. This results in measurements only being processed on a single CPU while other CPUs wait for it to complete. This significantly constrains the measurement throughput. While multi-threading does not directly help with SMI latency of a single measurement, it would multiply the performance of the measurement agent's processing of the measurement queue.

A second area for future work is making EPA-RIMM compatible with moving target defenses. The moving target defense approach attempts to complicate the work of an attacker by changing the system configuration at runtime. It can, for example, move kernel code in memory, change IP addresses periodically, in an effort to complicate the work of an attacker. EPA-RIMM currently assumes a static environment in which the system resources are not constantly shifting. Enabling EPA-RIMM for this class of systems could be accomplished by employing behavioral rootkit detections, e.g. leveraging performance counters. These methods that characterize how code operates as opposed to where it operates from would help respond to

this new mechanism.

The third area for future work is fully integrating telemetry into EPA-RIMM. EPA-RIMM's Diagnosis Manager is well situated to direct flows of measurements across an infrastructure. Detections of issues on one or more nodes could help drive detections of these issues on other nodes. This capability would reduce the ability of an attacker to spread an attack. Telemetry could also be used to enable new Checks based on emerging threat indicators. This could result in more effective checking and increase the responsiveness of the framework as successful detections of attacks on some nodes could guide checking on other nodes.

There are also three potential developments that could significantly improve EPA-RIMM's effectiveness. First, one of the largest portions of EPA-RIMM processing are the SMI entry and exit times. CPU optimizations that allow entry and exit into SMM to occur in reduced time would make it more feasible to use SMM as a protected execution environment. This could potentially enable innovative new usages that could readily be enabled via firmware updates.

Second, current chipsets support SMI timers that can trigger SMIs on a regular cadence. Such a timer could trigger measurements, however, there are two key limitations. First, the predictable nature of these triggers would allow rootkits to conduct a transient evasion. Thus, a randomized timer would be necessary to counter-act this attack. Second, current chipset SMI triggers can be disabled by non-SMM Ring 0 code. Improvements to lock this SMI generation source would provide a more resilient measurement trigger.

Third, communication methods that pass through the operating system present challenges for stealthy communication with a measurement agent.

While some SMM-RIMMs have leveraged out-of-band communication mechanisms to retrieve results and trigger measurements, these mechanisms have not been entirely stealthy or required porting network drivers to run in SMM which increases the attack surface. An improved mechanism that natively provides cryptographic support would streamline communications with the Inspector and reduce the overheads for security-related SMM usages.

11.3 Conclusions

From our measurements and analysis, we conclude that constructing a performance-aware, effective, and extensible SMM-RIMM is possible. Our initial SMM performance measurements provided confidence that decomposing large measurements would allow SMM-based integrity measurements to be accomplished over an interval of time within SMI latency bounds. This enables deeper inspections over an interval than could be performed in a single SMI session. The extensible framework also allows new inspections to be created that respond to future rootkits. The framework could incorporate a broad variety of inspections that extend beyond hashes to new detection techniques. Broader deployments of EPA-RIMM would provide an effective new detection mechanism that can detect stealthy rootkits. Enabling this new class of detection mechanisms provides a new capability to defenders to reduce the time to detect malicious code and reduce its ability to operate unnoticed.

Bibliography

[1] P. Alpeyev and G. Huang.
"Sony Hackers Seen Having Snooped for Months, Planted Bomb".
In: *Bloomberg* (2014).

[2] AMD. *AMD64 Architecture Programmer's Manual, Volume 2: System Programming*.

[3] T. Arnold. *Implementing PCI: A Guide for Network Engineers*. URL: http://www.juniper.net/us/en/local/pdf/whitepapers/2000268-en.pdf.

[4] S. Aubert. "Announce: rkscan, a kernel-based rootkit scanner." 2000.
URL: http://seclists.org/incidents/2000/Oct/165.

[5] A. M. Azab et al.
"HIMA: A Hypervisor-Based Integrity Measurement Agent".
In: *2009 Annual Computer Security Applications Conference*. 2009, pp. 461–470. DOI: 10.1109/ACSAC.2009.50.

[6] Ahmed M. Azab, Peng Ning, and Xiaolan Zhang.
"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*. CCS '11. Chicago, Illinois, USA: ACM, 2011, pp. 375–388.
ISBN: 978-1-4503-0948-6. DOI: 10.1145/2046707.2046752.
URL: http://doi.acm.org/10.1145/2046707.2046752.

[7] Ahmed M. Azab et al. "HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS '10. Chicago, Illinois, USA: ACM, 2010, pp. 38–49. ISBN: 978-1-4503-0245-6.
DOI: 10.1145/1866307.1866313.
URL: http://doi.acm.org/10.1145/1866307.1866313.

[8] *Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World*. CCS '14. Scottsdale, Arizona, USA: ACM, 2014, pp. 90–102. ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660350.
URL: http://doi.acm.org/10.1145/2660267.2660350.

[9] F. Bacurio, W. Low, and J. Manuel.
"Evasive Sage 2.2 Ransomware Variant Targets More Countries".
In: (2017).
URL: https://www.fortinet.com/blog/threat-research/evasive-sage-2-2-ransomware-variant-targets-more-countries.html.

[10] Pete Beckman et al. "Benchmarking the Effects of Operating System Interference on Extreme-scale Parallel Machines".
In: *Cluster Computing* 11.1 (Mar. 2008), pp. 3–16. ISSN: 1386-7857.
DOI: 10.1007/s10586-007-0047-2.
URL: http://dx.doi.org/10.1007/s10586-007-0047-2.

[11] Mihir Bellare, Ran Canetti, and Hugo Krawczyk.
"Keying hash functions for message authentication".
In: *Annual International Cryptology Conference*. Springer. 1996, pp. 1–15.

[12] L. Brown. "Linux Idle Power Checkup". LinuxCon. 2010.

[13] Y. Bulygin and D Samyde. "Chipset-based approach to detect virtualization malware a.k.a. DeepWatch". BlackHat 2008. 2008.

[14] CERT. *BIOS implementations permit unsafe SMM function calls to memory locations outside of SMRAM*. 2015.
URL: https://www.kb.cert.org/vuls/id/631788.

[15] CERT. *Dell BIOS in some Latitude laptops and Precision Mobile Workstations vulnerable to buffer overflow*. 2013.
URL: https://www.kb.cert.org/vuls/id/912156.

[16] S. Chaki, A. Vasudevan, and et al. *Design, Development, and Automated Verification of an Integrity-Protected Hypervisor*. Tech. rep. 2012.

[17] Deeparnab Chakrabarty, Yunhong Zhou, and Rajan Lukose.
"Online knapsack problems".
In: *Workshop on internet and network economics (WINE)*. 2008.

[18] Ronny Chevalier et al.
"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*. ACSAC 2017. Orlando, FL, USA: ACM, 2017, pp. 399–411.
ISBN: 978-1-4503-5345-8. DOI: 10.1145/3134600.3134622.
URL: http://doi.acm.org/10.1145/3134600.3134622.

[19] *Common Vulnerabilities and Exposures, https://cve.mitre.org*. 2018.
URL: https://cve.mitre.org/.

[20] Cr4sh. "Thinkpwn". URL: https://github.com/Cr4sh/ThinkPwn.

[21] crowdstrike.com.
"VENOM. Virtualized Environment Neglected Operations Manipulation".
URL: cve.mitre.org.

[22] J. Davis. "Indiana Cancer Agency hacked by TheDarkOverlord".
In: *Healthcare IT News* (2017).
URL: http://www.healthcareitnews.com/news/indiana-cancer-agency-hacked-thedarkoverlord.

[23] B. Delgado and K. L. Karavanic.
"Performance implications of System Management Mode". In: *2013 IEEE International Symposium on Workload Characterization (IISWC)*. 2013, pp. 163–173. DOI: 10.1109/IISWC.2013.6704682.

[24] Brian Delgado et al.
"EPA-RIMM: An Efficient, Performance-Aware Runtime Integrity Measurement Mechanism for Modern Server Platforms".
In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2019, pp. 422–434.

[25] John Demme et al. "On the Feasibility of Online Malware Detection with Performance Counters". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA '13.
Tel-Aviv, Israel: ACM, 2013, pp. 559–570. ISBN: 978-1-4503-2079-5.
DOI: 10.1145/2485922.2485970.
URL: http://doi.acm.org/10.1145/2485922.2485970.

[26] L. Duflot, O. Levillian, and et al. *Getting into SMRAM: SMM reloaded*. CanSecWest. 2009.

[27] J. Edge. "Kernel address space layout randomization".
URL: http://lwn.net/Articles/569635/.

[28] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh.
"Jump over ASLR: Attacking Branch Predictors to Bypass ASLR".
In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-49. Taipei, Taiwan: IEEE Press, 2016, 40:1–40:13.
URL: http://dl.acm.org/citation.cfm?id=3195638.3195686.

[29] f0rb1dd3n. *Linux Rootkit Desmonstration Codes*. URL: https://github.com/f0rb1dd3n/papers/tree/master/rootkit_demonstration.

[30] Kurt B. Ferreira, Patrick Bridges, and Ron Brightwell.
"Characterizing Application Sensitivity to OS Interference Using Kernel-level Noise Injection".
In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. SC '08. Austin, Texas: IEEE Press, 2008, 19:1–19:12.
ISBN: 978-1-4244-2835-9.
URL: http://dl.acm.org/citation.cfm?id=1413370.1413390.

[31] P. Ferrie and N. et al Lawson.
Don't Tell Joanna - The Virtualized Rootkit is Dead. Black Hat. 2007.

[32] Fortinet. "CIO Tips for Beating Security Latency".
URL: http://latonetworks.com/media/3845/cio-tips-for-beating-security-latency.pdf.

[33] Gartner. "Gartner says Detection and Response is Top Security Priority for Organizations in 2017". In: (2017).
URL: https://www.gartner.com/newsroom/id/3638017.

[34] *Getting maximum mileage out of tickless*.
Vol. Proceedings of the Ottawa Linux symposium.
Ottawa, Canada, 2007.

[35] M. Gottscho et al. "Measuring the Impact of Memory Errors on Application Performance".
In: *IEEE Computer Architecture Letters* 16.1 (2017), pp. 51–55.
ISSN: 1556-6056. DOI: 10.1109/LCA.2016.2599513.

[36] Daniel Gruss et al. "KASLR is Dead: Long Live KASLR".
In: *Engineering Secure Software and Systems*.
Ed. by Eric Bodden, Mathias Payer, and Elias Athanasopoulos.
Cham: Springer International Publishing, 2017, pp. 161–176.
ISBN: 978-3-319-62105-0.

[37] Daniel Gruss et al.
"Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR".
In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: ACM, 2016, pp. 368–379. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978356.
URL: http://doi.acm.org/10.1145/2976749.2978356.

[38] Shay Gueron.
"Intel® Advanced Encryption Standard (AES) New Instructions Set".
URL:
https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf.

[39] M. Hoekstra. *Intel SGX for Dummies (Intel SGX Design Objectives)*. 2013.
URL: https://software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx.

[40] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*.
Addison-Wesley Professional, 2005.

[41] Ralf Hund, Carsten Willems, and Thorsten Holz.
"Practical Timing Side Channel Attacks Against Kernel Space ASLR".
In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*.
SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 191–205. ISBN: 978-0-7695-4977-4. DOI: 10.1109/SP.2013.23.
URL: http://dx.doi.org/10.1109/SP.2013.23.

[42] Ponemon Institute. "2014 Cost of Cyber Crime Study: United States".
In: (Oct. 2014). URL:
https://ssl.www8.hp.com/ww/en/secure/pdf/4aa5-5208enw.pdf.

[43] Intel.
"Host Firmware Speculative Execution Side Channel Mitigation".
URL: https://software.intel.com/security-software-guidance/insights/host-firmware-speculative-execution-side-channel-mitigation.

[44] Intel. *Intel 6 Series Chipset and Intel C200 Series Chipset, Data Sheet*. 2011.

[45] Intel.
Intel® 64 and IA-32 Architectures Software Developer's Manual (Vol. 3).

[46] Intel. *Intel Itanium Architecture Software Developer's Manual, Revision 2.3. Volume 2: System Architecture*.

[47] Intel. "New Instructions Supporting the Secure Hash Algorithm on Intel® Architecture Processors". URL: https://software.intel.com/en-us/articles/intel-sha-extensions.

[48] Intel. *SMI Transfer Monitor (STM) User Guide*.

[49] *Intel Atom® Processor E3826, Intel ARK*. 2018.

[50] Daehee Jang et al. "ATRA: Address Translation Redirection Attack Against Hardware-based External Monitors".
In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14.
Scottsdale, Arizona, USA: ACM, 2014, pp. 167–178.
ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660303.
URL: http://doi.acm.org/10.1145/2660267.2660303.

[51] Yeongjin Jang, Sangho Lee, and Taesoo Kim. "Breaking Kernel Address Space Layout Randomization with Intel TSX".
In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: ACM, 2016, pp. 380–392. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978321.
URL: http://doi.acm.org/10.1145/2976749.2978321.

[52] kad. *Handling Interrupt Descriptor Table for fun and profit*. 2002.
URL: http://phrack.org/issues/59/4.html.

[53] C. Kallenberg and X. Kovah.
"How Many Million BIOSes Would you Like to Infect?" In:
Vancouver, Canada: CanSecWest, 2015.

[54] Erin Kelly. "Officials warn 500 million financial records hacked".
In: *USA Today* (2014). URL: https://www.usatoday.com/story/news/politics/2014/10/20/secret-service-fbi-hack-cybersecuurity/17615029/.

[55] Linux Kernel. "TSC". URL: http://lxr.free-electrons.com/source/arch/x86/kernel/tsc.c#L646.

[56] Gerwin Klein et al. “seL4: Formal Verification of an OS Kernel”. In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP ’09. Big Sky, Montana, USA: ACM, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. URL: http://doi.acm.org/10.1145/1629575.1629596.

[57] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *CoRR* abs/1801.01203 (2018). arXiv: 1801.01203. URL: http://arxiv.org/abs/1801.01203.

[58] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-hashing for message authentication*. Tech. rep. 1997.

[59] Invisible Things Lab. *Invisible Things Lab presents the "Press Cheat Sheet" for the Attacking Intel® Trusted Execution Technology presentation at the Black Hat DC conference*. 2009. URL: https://web.archive.org/web/20170202053751/http://invisiblethingslab.com/press/itl-press-2009-02.pdf.

[60] Michael. Larabel. “System76 Launches Two Intel Laptops With "Open-Source Firmware" Coreboot”. Oct. 2019. URL: https://www.phoronix.com/scan.php?page=news_item&px=System76-Two-Laptops-Coreboot.

[61] Barry G Lawson and Evgenia Smirni. “Multiple-queue backfilling scheduling with priorities and reservations for parallel systems”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2002, pp. 72–87.

[62] M. Lennon. “McAfee "Deep Defender" Endpoint Security Targets Kernel-Mode Malware”. Oct. 2011. URL: https://www.securityweek.com/mcafee-deep-defender-endpoint-security-targets-kernel-mode-malware.

[63] M. Li and H. Cao. “Locky Ransomware Spreads via Flash and Windows Kernel Exploits”. In: (2016). URL: https://web.archive.org/web/20170420170059/https://blog.trendmicro.com/trendlabs-security-intelligence/locky-ransomware-spreads-flash-windows-kernel-exploits/.

[64] Moritz Lipp et al. “Meltdown”. In: *CoRR* abs/1801.01207 (2018). arXiv: 1801.01207. URL: http://arxiv.org/abs/1801.01207.

[65] Ziyi Liu et al. “CPU Transparent Protection of OS Kernel and Hypervisor Integrity with Programmable DRAM”. In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA ’13. Tel-Aviv, Israel: ACM, 2013, pp. 392–403. ISBN: 978-1-4503-2079-5. DOI: 10.1145/2485922.2485956. URL: http://doi.acm.org/10.1145/2485922.2485956.

[66] D. Lo and C. Kozyrakis.
"Dynamic Management of TurboMode in Modern Multi-core Chips".
In: 2013. URL: http://csl.stanford.edu/~christos/publications/2014.autoturbo.hpca.pdf.

[67] J. Loucaides. "BIOS and Secure Boot Attacks Uncovered".
URL: http://www.c7zero.info/stuff/DEFCON22-BIOSAttacks.pdf.

[68] J. Loucaides and Y. Bulygin.
Platform Security Assessment with CHIPSEC.
https://cansecwest.com/slides/2014/Platform. 2014.

[69] Ijlal Loutfi. "SMMDecoy: Detecting GPU Keyloggers using Security by Deception Techniques".
In: *Proceedings of the 5th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP,* INSTICC. SciTePress, 2019, pp. 580–587. ISBN: 978-989-758-359-9.
DOI: 10.5220/0007578505800587.

[70] K Mannthey. "System Management Interrupt Free Hardware". 2009.
URL: http://linuxplumbersconf.org/2009/slides/Keith-Mannthey-SMIplumers-2009.pdf.

[71] J. Masters. *[RFC] simple SMI detector*. Jan. 2009.
URL: https://lwn.net/Articles/316622/.

[72] W Mauerer. *Professional Linux Kernel Architecture*. Wrox, 2008.

[73] Mcafee. *McAfee Deep Defender Data Sheet*.
urlhttp://www.mcafee.com/us/resources/data-sheets/ds-deep-defender.pdf.

[74] Mcafee. *McAfee Deep Defender Technical Evaluation and Best Practices Guide, Version 1.0.* https://kc.mcafee.com/resources/sites/MCAFEE/content/live/PRODUCT_DOCUMENTATION/23000/PD23874/en_US/Deep_Defender_Best_Practices_Guide_Aug_2012.pdf. 2012.

[75] Jonathan M. McCune et al.
"Flicker: An Execution Infrastructure for Tcb Minimization".
In: *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. Eurosys '08.
Glasgow, Scotland UK: ACM, 2008, pp. 315–328.
ISBN: 978-1-60558-013-5. DOI: 10.1145/1352592.1352625.
URL: http://doi.acm.org/10.1145/1352592.1352625.

[76] Frank McKeen et al.
"Innovative Instructions and Software Model for Isolated Execution".
In: *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '13.
Tel-Aviv, Israel: ACM, 2013, 10:1–10:1. ISBN: 978-1-4503-2118-1.
DOI: 10.1145/2487726.2488368.
URL: http://doi.acm.org/10.1145/2487726.2488368.

[77] Microsoft. *Microsoft Launches Windows Vista and Microsoft Office 2007 to Consumers Worldwide*. Jan. 2007. URL: http://news.microsoft.com/2007/01/29/microsoft-launches-windows-vista-and-microsoft-office-2007-to-consumers-worldwide/.

[78] Microsoft. *Understanding the Windows SMM Security Mitigation Table (WSMT)*. Mar. 2018. URL: https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-uefi-wsmt.

[79] Microsoft. *Windows 8.1 security improvements*. URL: https://web.archive.org/web/20160402042352/https://technet.microsoft.com/en-us/windows/jj983723.aspx.

[80] Barton P. Miller et al. "The Paradyn Parallel Performance Measurement Tool". In: *Computer* 28.11 (Nov. 1995), pp. 37–46. ISSN: 0018-9162. DOI: 10.1109/2.471178. URL: http://dx.doi.org/10.1109/2.471178.

[81] Hyungon Moon et al. "Vigilare: Toward Snoop-based Kernel Integrity Monitor". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 28–37. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382202. URL: http://doi.acm.org/10.1145/2382196.2382202.

[82] X. Mupta D. Li. "Defeating patchguard". In: *Whitepaper* (2012).

[83] E.D. Myers. *STM/PE and XHIM. Poster*. USENIX Security Symposium Poster. Aug. 2017.

[84] Eugene D. Myers. *Using the Intel STM for Protected Execution*. 2018. URL: http://www.platformsecuritysummit.com/2018/speaker/myers/STMPE2Intelv84a.pdf.

[85] "Netperf". URL: http://www.netperf.org.

[86] *OpenSSL*. URL: https://www.openssl.org/.

[87] panda. "WannaCry Report". In: (2017). URL: https://www.pandasecurity.com/mediacenter/src/uploads/2017/05/WannaCry_Report-en.pdf.

[88] J. Aaron Pendergrass and Kathleen. McGill. "LKIM: The LInux Kernel Integrity Monitor". 2013. URL: http://www.jhuapl.edu/techdigest/TD/td3202/32_02-Pendergrass-McGill.pdf.

[89] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin.
"The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q".
In: *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. SC '03. Phoenix, AZ, USA: ACM, 2003, pp. 55–. ISBN: 1-58113-695-1. DOI: 10.1145/1048935.1050204.
URL: http://doi.acm.org/10.1145/1048935.1050204.

[90] Nick L. Petroni Jr. et al.
"Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor".
In: *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*. SSYM'04. San Diego, CA: USENIX Association, 2004, pp. 13–13.
URL: http://dl.acm.org/citation.cfm?id=1251375.1251388.

[91] *Phoronix, http://www.phoronix.com*. 2018.
URL: http://www.phoronix.com.

[92] David Pisinger. *Algorithms for Knapsack Problems*. 1995.

[93] V. Prasad, W. Cohen, and et al.
"Locating System Problems Using Dynamic Instrumentation".
URL: https://sourceware.org/systemtap/systemtap-ols.pdf.

[94] Alsa project. "PCM Interface". URL: https://www.alsa-project.org/alsa-doc/alsa-lib/group___p_c_m.html.

[95] *Project RC5*. URL: https://www.distributed.net/RC5.

[96] Purism. "coreboot Firmware on Purism Librem devices".
URL: https://puri.sm/coreboot/.

[97] K. Ram, J. Santos, and et al.
Redesigning Xen Memory Sharing (Grant) Mechanism. Xen Summit. 2011.

[98] J. Roetters. "How Hollywood Got Hacked: Studio at Center of Netflix Leak Breaks Silence". In: *Variety* (2017). URL: http://variety.com/2017/digital/features/netflix-orange-is-the-new-black-leak-dark-overlord-larson-studios-1202471400/.

[99] Joana Rutkowska. "Intel x86 considered harmful". Oct. 2015.
URL: https://blog.invisiblethings.org/papers/2015/x86_harmful.pdf.

[100] Reiner Sailer et al. "Design and Implementation of a TCG-based Integrity Measurement Architecture". In: *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*. SSYM'04. San Diego, CA: USENIX Association, 2004, pp. 16–16.
URL: http://dl.acm.org/citation.cfm?id=1251375.1251391.

[101] *Saint Michael Linux LKM*. 2014.
URL: https://github.com/tomasz-janiczek/stmichael-lkm.

[102] Arvind Seshadri et al. "SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes". In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 335–350. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294294. URL: http://doi.acm.org/10.1145/1294261.1294294.

[103] Galvin Silberchatz. "Gagne, 2003". In: *Operating systems concepts* ().

[104] Push the Stack Consulting. *Security when Nanoseconds Count*. BlackHat Conference. 2011. URL: http://media.blackhat.com/bh-us-11/Arlen/BH_US_11_Arlen-HFT_WP.pdf.

[105] Talos. "Threat Spotlight: Follow the Bad Rabbit". In: (2017). URL: https://blog.talosintelligence.com/2017/10/bad-rabbit.html.

[106] J. Triplett and B. Tripplet. *BITS: BIOS Implementation Test Suite*. http://www.linuxplumbersconf.org/2011/ocw/system/presentations/867/original/bits.pdf.

[107] *Trusted Computing Group's Trusted Network Connect Technology Standards Development for Network Security Interoperability*. URL: https://www.nist.gov/sites/default/files/documents/standardsgov/TCG.pdf.

[108] Dan Tsafrir et al. "System Noise, OS Clock Ticks, and Fine-grained Parallel Applications". In: *Proceedings of the 19th Annual International Conference on Supercomputing*. ICS '05. Cambridge, Massachusetts: ACM, 2005, pp. 303–312. ISBN: 1-59593-167-8. DOI: 10.1145/1088149.1088190. URL: http://doi.acm.org/10.1145/1088149.1088190.

[109] Verizon. "2014 Data Breach Investigations Report". In: (2014). URL: http://www.secretservice.gov/Verizon_Data_Breach_2014.pdf.

[110] T. Vibhute. "EPA-RIMM-V: Efficient Rootkit Detection for Virtualized Environment". MA thesis. Portland State University, 2018.

[111] T. Villa, S. Gitanjali, and et al. *VIS User's Manual*. URL: https://embedded.eecs.berkeley.edu/research/vis/doc/VisUser/vis_user/node4.html.

[112] J. Wang, Sun K., and et al. *An Analysis of System Management Mode (SMM)-based Integrity Checking Systems and Evasion Attacks*. Tech. rep. GMU-CS-TR-2011-8. George Mason University, 2011.

[113] J. Wang, A. Stavrou, and et al. "HyperCheck: A Hardware-assisted Integrity Monitor". In: *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection*. RAID'10. Ottawa, Ontario, Canada: Springer-Verlag, 2010, pp. 158–177.

ISBN: 3-642-15511-1, 978-3-642-15511-6.
URL: http://dl.acm.org/citation.cfm?id=1894166.1894178.

[114] D. Wilkins and B. Richardson. *UEFI SECURE BOOT IN MODERN COMPUTER SECURITY SOLUTIONS*. Sept. 2013. URL: http://www.uefi.org/sites/default/files/resources/UEFI_Secure_Boot_in_Modern_Computer_Security_Solutions_2013.pdf.

[115] "XDD". URL: http://sourceforge.net/projects/xdd.

[116] J. Yao, V. Zimmer, and S. Zeng. *A Tour Beyond BIOS Secure SMM Communication in the EFI Developer Kit II*. Tech. rep. Intel, 2016.

[117] Jiewen Yao. *[edk2] [PATCH V2 0/6] Enable SMM page level protection..* Nov. 2016. URL: https://lists.01.org/pipermail/edk2-devel/2016-November/004185.html.

[118] F. Zhang et al.
"Using Hardware Features for Increased Debugging Transparency".
In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 55–69.
DOI: 10.1109/SP.2015.11.

[119] Fengwei Zhang et al. "SPECTRE: A Dependable Introspection Framework via System Management Mode".
In: *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. DSN '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–12.
ISBN: 978-1-4673-6471-3. DOI: 10.1109/DSN.2013.6575343.
URL: http://dx.doi.org/10.1109/DSN.2013.6575343.

[120] Lei Zhou et al.
"Nighthawk: Transparent System Introspection from Ring-3".
In: *European Symposium on Research in Computer Security*.
Springer. 2019, pp. 217–238.