Scaling EPA-RIMM with Multicore System Management Interrupt Handlers

Alexander K. Freed
Portland State University

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

Part of the Computer Sciences Commons

Let us know how access to this document benefits you.

Recommended Citation
https://doi.org/10.15760/etd.8115

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.
Scaling EPA-RIMM with Multicore System Management Interrupt Handlers

by

Alexander K. Freed

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

Master of Science
in
Computer Science

Thesis Committee:
Karen L. Karavanic, Chair
Mark P. Jones
Wu-chang Feng

Portland State University
2022
Abstract

Continuous runtime integrity measurement mechanisms (RIMMs) can be used for timely detection of kernel and hypervisor rootkits. Researchers have proposed running RIMMs in privileged execution environments, such as the x86 architecture’s System Management Mode (SMM), to detect interference from rootkits that have gained control of the host operating system. However, the extended amount of time in SMM required to perform inspections can cause severe disruption to the host. A previously proposed RIMM design called EPA-RIMM addresses this by decomposing long inspections across multiple System Management Interrupts (SMI), the interrupt used to invoke SMM.

EPA-RIMM is intended for deployment on server-class computers. There are typically more cores available on server platforms than client platforms. In existing firmware implementations, all but one core are kept idle in SMM, so utilizing additional cores requires changes to the thread-unaware firmware runtime services that execute when SMM is entered. These idle cores could be utilized to improve detection of RIMM-aware scrubbing rootkits by allowing for more security inspections to be done in the same amount of time.

This thesis presents a new multicore version of EPA-RIMM that is capable of functioning on the Linux operating system. It is written in UEFI firmware—the most commonly used firmware specification. Adjustments are proposed to existing EPA-RIMM inspection and check scheduling design to facilitate multicore execution. Enhancements are also proposed to a UEFI implementation, EDK2, to add support
for multicore execution in SMI handlers. Performance results are presented from a modified EPA-RIMM prototype utilizing all four cores of the Intel Atom-class MinnowBoard platform.

We found that the communication-related cryptographic operations should be parallelized as well as the inspection itself in order to achieve performance improvement. Although we were not able to fully parallelize HMAC, performance improvement was achieved within a realistic time-bound of less than 1.5 ms. Single-core inspection performed best with small Task sizes. Two-core inspection outperforms single-core when Task sizes are 2 KiB or greater. Four-core inspection outperforms when Task sizes are 4 KiB or greater.
Acknowledgments

There are many people I owe this success to—Dr. Karen Karavanic for encouraging me and pushing me, Dr. Brian Delgado for the project and all the help along the way, the rest of the EPA-RIMM team past and present, Tejaswini Vibhute, John Fastabend, Cody Shepherd, Payal Joshi, and Dylan Abraham. Thank you to the other committee members, Dr. Mark Jones and Dr. Wu-chang Feng for reviewing the work and also being flexible with the timing. Thank you to the people who helped me decide to go back to grad school—Jim Gower, Dr. Bryant York, and Sandy and Steve Marsden. Thank you to Ted Cooper for reviewing my code. Thank you to my editors and friends—Victoria Smith, Erik Kaneda, Leila Kaneda, and Brian Delgado. Thank you to my friends who encouraged me. Thank you as well to the friends who told me to quit. I really mean that—I know that you just want what is best for me. Friends of both types (you know who you are) include Robbie Penziol, Summer Young, Alyssa Tamayo, Zachary Torres, Kenny Katz, and thank you as well to your better halves for supporting me in friendship. Thank you to all my other friends and anyone else I forgot to mention due to sleep deprivation. Thanks to my coworkers and also my bosses, who were supportive of me beyond what I could have expected. Thank you to the incredible nurses at St. Vincent hospital in the labor and delivery department as well as the mother and baby department.

Extra special thanks to my wife, Sarah. Thank you for being understanding and supportive. It has been quite an adventure so far and I’m so excited to have our first little baby, Felix!
# Table of Contents

Abstract .......................................................................................................................... i
Acknowledgments .......................................................................................................... iii
List of Tables .................................................................................................................... vii
List of Figures ................................................................................................................... viii
Glossary ........................................................................................................................... ix

1 Introduction ..................................................................................................................... 1
  1.1 Motivation .................................................................................................................. 1
  1.2 Problem statement ..................................................................................................... 5
  1.3 Objective and approach ............................................................................................ 6
  1.4 Thesis statement ........................................................................................................ 6
  1.5 Contributions ............................................................................................................ 7

2 Background ..................................................................................................................... 8
  2.1 UEFI ........................................................................................................................... 8
  2.2 EDK2 ........................................................................................................................ 8
    2.2.1 EDK2 on open firmware platforms ...................................................................... 9
    2.2.2 EDK2 boot process ........................................................................................... 10
    2.2.3 SMM .................................................................................................................. 10
    2.2.4 SMRAM ............................................................................................................ 11
    2.2.5 SMI .................................................................................................................... 12
    2.2.6 Issuing SMIs – synchronous vs. asynchronous ................................................. 12
    2.2.7 SMM privileges ................................................................................................. 13
    2.2.8 System perturbation ......................................................................................... 14
    2.2.9 Caching ............................................................................................................. 15
  2.3 Semantic gap and context reconstruction ................................................................... 15
  2.4 STM .......................................................................................................................... 16
  2.5 Concurrency in SMI handlers ................................................................................... 18
  2.6 Concurrency in RIMMs ............................................................................................ 20
  2.7 DMA and IOMMUs ................................................................................................. 22

3 Existing EPA-RIMM Architecture ............................................................................. 23

4 Related Work ............................................................................................................... 29
  4.1 Copilot ....................................................................................................................... 29
  4.2 HyperSentry ............................................................................................................. 30
  4.3 HyperCheck ............................................................................................................. 31
  4.4 SPECTRE ................................................................................................................. 33
  4.5 SATIN ....................................................................................................................... 34
  4.6 STM/PE ..................................................................................................................... 36

5 A Study of SMI Handler Implementation .................................................................. 37
  5.1 Details of SMM enter and exit .................................................................................. 37
  5.2 EDK2 SMM execution flow (no STM) .................................................................... 38
  5.3 EDK2 SMM execution flow (with STM) ................................................................. 42
# Table of Contents

6  Design.......................................................................................................................... 44
   6.1  Firmware modifications to provide multicore SMI handlers................................. 45
      6.1.1  Possibility 1: Modify EDK2 dispatch code ...................................................... 47
      6.1.2  Possibility 2: Reroute SMM flow during SMI Rendezvous ......................... 49
      6.1.3  Possibility 3: Add a concurrent driver dispatch module ............................... 51
      6.1.4  Possibility 4: Put the Inspector inside STM .................................................. 52
      6.1.5  Possibility 5: Put SMI handler in additional SMM Guest .............................. 53
      6.1.6  Selected approach .......................................................................................... 54
      6.1.7  EDK2 modification specifics  ......................................................................... 54
   6.2  Scheduling modifications ....................................................................................... 55
   6.3  SMM Inspector modifications .............................................................................. 57
   6.4  Bin protocol modifications ................................................................................... 57
      6.4.1  Existing protocol ............................................................................................ 58
      6.4.2  Possibility 1: Add a Binset Header ................................................................ 59
      6.4.3  Possibility 2: Add Bin Headers ..................................................................... 62
      6.4.4  Possibility 3: Add a Binset Header and Bin Headers .................................... 65
      6.4.5  Selected approach ......................................................................................... 67

7  Evaluation....................................................................................................................... 68
   7.1  Security analysis .................................................................................................... 68
      7.1.1  Assumptions .................................................................................................. 68
      7.1.2  Threat model and attack surface .................................................................. 68
   7.2  Effectiveness .......................................................................................................... 70
   7.3  Performance model ............................................................................................... 71
   7.4  Predicted performance improvement .................................................................... 73
      7.4.1  Strong scaling ............................................................................................... 73
      7.4.2  Weak scaling ............................................................................................... 75
   7.5  Miscellaneous measurement guidelines .............................................................. 76
   7.6  Performance scaling initial failure ...................................................................... 77
   7.7  Issues with multicore HMAC in OpenSSL ............................................................ 79
   7.8  Performance scaling success ................................................................................ 80
   7.9  Performance impacts of HMAC .......................................................................... 86
   7.10 Discussion .............................................................................................................. 92

8  Conclusions and Future Work .................................................................................... 93
   8.1  Conclusions .......................................................................................................... 93
   8.2  Future work .......................................................................................................... 93
      8.2.1  Per-function inspection scheduling .............................................................. 93
      8.2.2  Binset protocol ............................................................................................. 94
      8.2.3  Caching effects ............................................................................................. 94
      8.2.4  Granular measurements of SMI entry and exit ........................................... 95
References .................................................................................................................................................. 96
Appendix – Details of SMM Execution Flow ............................................................................................ 101
  EDK2 SMM execution flow (no STM) ..................................................................................................... 101
  Steps for identifying PchSmmCoreDispatcher as closed-source ...................................................... 101
  EDK2 SMM execution flow (with STM) ................................................................................................. 103
List of Tables

Table 7-1: Measurements with parallelized inspection only ........................................ 78
Table 7-2: Average inspection times ................................................................. 81
Table 7-3: Standard deviation of Table 7-2 .......................................................... 81
Table 7-4: Additional average inspection times ..................................................... 82
Table 7-5: Standard deviation of Table 7-4 .......................................................... 82
Table 7-6: Average inspection times (no HMAC) .................................................. 86
Table 7-7: Standard deviation of Table 7-6 .......................................................... 87
Table 7-8: Additional average inspection times (no HMAC) ................................... 87
Table 7-9: Standard deviation of Table 7-8 .......................................................... 87
### List of Figures

<table>
<thead>
<tr>
<th>Figure 3-1: EPA-RIMM architecture</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 5-1: EDK2 SMI Execution Flow (Blackbox)</td>
<td>39</td>
</tr>
<tr>
<td>Figure 5-2: EDK2 SMI Execution Flow (Clearbox)</td>
<td>39</td>
</tr>
<tr>
<td>Figure 5-3: EDK2 SMI Execution Flow with STM (Blackbox)</td>
<td>41</td>
</tr>
<tr>
<td>Figure 5-4: EDK2 SMI Execution Flow with STM (Clearbox)</td>
<td>41</td>
</tr>
<tr>
<td>Figure 6-1: Single-core UEFI EPA-RIMM runs as a driver</td>
<td>46</td>
</tr>
<tr>
<td>Figure 6-2: Possibility 1 – Modify Rendezvous and Dispatcher</td>
<td>46</td>
</tr>
<tr>
<td>Figure 6-3: Possibility 2 – Reroute the execution flow</td>
<td>49</td>
</tr>
<tr>
<td>Figure 6-4: Possibility 3 – Add multicore dispatcher</td>
<td>50</td>
</tr>
<tr>
<td>Figure 6-5: Possibility 4 – Put Inspector inside STM</td>
<td>52</td>
</tr>
<tr>
<td>Figure 6-6: Possibility 5 – Put the Inspector in a dedicated SMM Guest</td>
<td>53</td>
</tr>
<tr>
<td>Figure 6-7: Existing Bin memory layout</td>
<td>59</td>
</tr>
<tr>
<td>Figure 6-8: Existing execution sequence mock-up</td>
<td>59</td>
</tr>
<tr>
<td>Figure 6-9: Possibility 1 – Binset memory layout</td>
<td>61</td>
</tr>
<tr>
<td>Figure 6-10: Possibility 1 – Execution sequence mock-up</td>
<td>61</td>
</tr>
<tr>
<td>Figure 6-11: Possibility 2 – Binset memory layout</td>
<td>64</td>
</tr>
<tr>
<td>Figure 6-12: Possibility 2 – Execution sequence mock-up</td>
<td>64</td>
</tr>
<tr>
<td>Figure 6-13: Possibility 3 – Binset memory layout</td>
<td>66</td>
</tr>
<tr>
<td>Figure 6-14: Possibility 3 – Execution sequence mock-up</td>
<td>67</td>
</tr>
<tr>
<td>Figure 7-1: Average $T_{ovd}$ as number of cores increases</td>
<td>83</td>
</tr>
<tr>
<td>Figure 7-2: Performance scaling as total inspected bytes increases</td>
<td>84</td>
</tr>
<tr>
<td>Figure 7-3: Zoomed-in version of Figure 7-2</td>
<td>85</td>
</tr>
<tr>
<td>Figure 7-4: Performance scaling as total inspected bytes increases (no HMAC)</td>
<td>88</td>
</tr>
<tr>
<td>Figure 7-5: Effect on $T_{work}$ as number of cores increases when (no HMAC)</td>
<td>90</td>
</tr>
<tr>
<td>Figure 7-6: Effect on $T_{work}$ as number of cores increases when (with HMAC)</td>
<td>91</td>
</tr>
</tbody>
</table>
Glossary

**AP** – Application Processor. Cores 1 through N. See *BSP*.

**Asynchronous SMI** – An *SMI* triggered by hardware.

**BSP** – Boot-Strap Processor. This is the core (core 0 by default) that performs the work in firmware while the others cores, or Application Processors (*APs*), wait. It is called the Boot-Strap Processor because it is the one performing the primary function of firmware—to bootstrap the system after power-on.

**Coreboot** – A lightweight open-source firmware. Formerly known as LinuxBIOS.

**DMA** – Direct Memory Access

**DXE** – Driver Execution Environment

**EDK2, EDK II** – EFI Developer Kit II. Intel’s reference implementation for *UEFI*.

**EFI** – Extensible Firmware Interface. The proprietary BIOS replacement superseded by non-proprietary *UEFI*.

**FV** – Firmware Volume. A logical firmware device defined in the *UEFI* specification.

**Guest** – A Virtual Machine (*VM*)

**Host** – See *Hypervisor*

**Hypervisor** – A kernel that manages Virtual Machines (*VMs*).

**IOMMU** – Input / Output Memory Management Unit

**Kernel** – The core of the Operating System (*OS*) that performs its essential functions.

**MM** – Management Mode. A generic term for the facility provided by CPU instruction architectures.
**Glossary**

**MMI** – Management Mode Invocation. A generic term for the interrupt that causes a CPU to enter Management Mode.

**OEM** – Original Equipment Manufacturer. This refers to platform manufacturers in the context of this thesis.

**OS** – Operating System. In this thesis, the term OS is generally used to refer to the kernel.

**PCH** – Platform Controller Hub

**PEI** – Pre-EFI Initialization

**PEIM** – *PEI* Module

**PI** – Platform Initialization

**Rich OS** – The feature-rich OS on a system that executes in the non-secured Normal World and runs and manages typical applications. The term is commonly used in relation to ARM TrustZone.

**Ring 3** ("Ring Three") – The level of privilege given to user-mode applications (fewer privileges than the Operating System).

**Ring 0** – ("Ring Zero") The level of privilege given to the Operating System.

**Ring -1** ("Ring Negative One") – The level of privilege given to the hypervisor on systems that have a hypervisor (more privileges than the Operating System).

**Ring -2** ("Ring Negative Two") – Any level of privilege that is more privileged than the hypervisor, such as another hypervisor or SMM.

**Ring -3** ("Ring Negative Three") – Any level of privilege that is more privileged than Ring -2, such as STM.
SMI – System Management Interrupt

SMM – System Management Mode

SMM IPL – SMM Initial Program Loader. Code that loads SMM services from flash to SMRAM and calls the initialization function.

SMRAM – System Management RAM. Main memory used for executing firmware.

SoC – System on Chip

Synchronous SMI – An SMI triggered by software.

System – Common shorthand for Operating System. Depending on context, however, it could also refer to firmware, the hardware platform, or a software architecture.

TEE – Trusted Execution Environment

TCB – Trusted Computing Base

TianoCore – A name for the collection of open source EDK2 projects and the community that contributes to them. Originally the codename for the EDK project at Intel before it became open-source.

TXT – Intel Trusted Execution Technology

UDK – UEFI Developer Kit

UEFI – Unified Extensible Firmware Interface

VM – Virtual Machine

VMCS – Virtual Machine Control Structure. Stores the execution context for an interrupted VM.

VMX — Intel Virtual Machine Extensions. The set of ten instructions: VMPtrLD,
VMPtrST, Vmclear, VMRead, VMWrite, VMCALL,VMLaunch, VMResume,
VMXoff, and VMXon that represent Virtual Technology (VT)

VT — Virtualization Technology

VT-i — VT for Itanium (IA-64). Hardware virtualization of Virtual Technology (VT)

VT-x — VT for x86. Hardware virtualization of Virtual Technology (VT)
1 Introduction

This research demonstrates performance improvement of System Management Mode (SMM)-based Runtime Integrity Measurement Mechanisms (RIMMs) by increasing concurrency. We consider the scalability of the measurements themselves as well as the cryptographic operations used for secure communication. We also discuss design changes needed to enable concurrency in firmware. In this section, we will discuss the motivation, problem statement, objective, thesis statement, and contributions of this thesis.

1.1 Motivation

A rootkit is a type of malware that can attain privileged access (“root access”) to the operating system or hypervisor. This allows an entity to monitor or modify anything the operating system has access to. It also allows the rootkit to hide by, for example, tampering with logs or malware-detection programs. Persistent rootkits are rootkits that are automatically launched when the computer starts. Persistent rootkits can go undetected for months, sometimes only discovered once the hackers take action, like attempting to sell stolen information. Rootkits have been used in high profile attacks, such as stealing customer records from large corporations and committing acts of government-on-government espionage.

One example of such an attack is the 2014 Sony Hack [29] attributed to North Korean state-level entity the Lazarus Group. The attack was in retribution for a comedy film called The Interview, in which North Korea’s leader is assassinated. It is
believed that the hackers were collecting data for months before they revealed themselves.

One way to detect kernel rootkits is with a Runtime Integrity Measurement Mechanism (RIMM). Kernel rootkits modify the normally protected kernel code. A RIMM detects changes in resources that are presumed to be static. A RIMM can, for example, inspect the kernel code by hashing it and comparing with a known hash to detect if the code has been modified. RIMMs can also check for signatures of known attacks by, for example, examining CPU control registers and model-specific registers (MSRs).

If a RIMM were to run at the privilege level of the OS, it would be vulnerable to tampering from a potential rootkit that has gained root access. To address this, researchers have proposed running the mechanism at a higher privilege level than the OS. System Management Mode (SMM) is a privileged operating mode on x86. A System Management Interrupt (SMI) will cause an x86 CPU to switch into SMM and jump to a firmware handler. The SMI is the highest priority interrupt, it is non-maskable, and unlike typical interrupts, it can interrupt all cores on the platform.

While in SMM, the code running has full privileges and access to everything on the platform—memory, CPU register state, Model-Specific Registers (MSRs), I/O devices, etc. SMM has more privileges than the OS, as the OS is not allowed access to some protected firmware resources (such as memory dedicated to firmware). An SMI immediately preempts all cores, even if they are executing OS code. All interrupts are masked in SMM, so the CPU cannot be preempted while in SMM, and
the cores remain in SMM until the SMI handler runs to completion. The OS is not directly notified of an SMI’s occurrence. Thus, SMM is not directly visible to the OS. This makes it more challenging for a malicious rootkit with kernel OS-level privileges to block, tamper with, or hide from a RIMM that runs in SMM.

There are a number of challenges facing SMM RIMMs. The handling of all interrupts is deferred while in SMM. While this is a useful property for preventing malicious preemption, it also prevents the OS from servicing desired interrupts, which may be time-sensitive. Remaining in SMM too long can cause severe disruption of services and can even crash the system. Delgado et al. empirically determined a maximum SMI runtime of 1.5 milliseconds [8]. SMM RIMMs are not viable if they exceed this guideline. Unfortunately, inspecting the entire kernel takes a more significant amount of time, and existing SMM RIMMs overstep the “empirical bound” by orders of magnitude.

To address this, Delgado et al. developed EPA-RIMM [9][10][11][7], an architecture for an SMM RIMM that limits the amount of time it spends handling each SMI. This is accomplished by decomposing inspections of large memory regions into many partial inspections called Tasks, and scheduling a set of Tasks per SMI that stays below the previously mentioned, empirically determined time bound. This approach has an additional benefit that the inspection is less perceptible to the user, since unrelated Tasks may be scheduled together.

EPA-RIMM solves the issue of system perturbation, but with a trade-off: a single inspection of the entire kernel takes more total time than other SMM RIMMs.
in terms of both wall time and work time. The wall time (total time from the beginning of the first measurement to the end of the last) is longer because decomposed partial inspections run at a specified frequency, meaning that the amount of work that can be done per second is bounded. The work time (amount of time spent running EPA-RIMM, not including other time spent on other tasks between inspector invocations) is also affected because each SMI has entry and exit overhead. This overhead counts toward the overall SMI time allotted by EPA-RIMM. Because large inspections are spread over many partial inspections, the additional entry and exit calls increase overall work time.

Delgado et al reported their EPA-RIMM prototype running on the dual core MinnowBoard Turbot took 70 seconds to inspect the 12 MB Linux kernel [11]. The longer it takes to inspect the kernel, the less confident one can be in the result. The decomposed inspection gives rootkits that are aware of the RIMM a window in which to operate undetected. “Scrubbing rootkits” could theoretically hide by temporarily undoing their malicious kernel code changes before inspection of the affected region and reinstating the changes afterwards. Since it is the OS text section that is inspected, the rootkit could regain control by compromising the stack of a running process [45]. Alternatively, they could temporarily relocate to an already-inspected region of the kernel. TZ-Evader is an academic example of such an evasion attack [44]. Zhang et al [57] describes three types of evasion attacks on SMIs.

This window of vulnerability is an unsolved drawback of viable SMM RIMMs. Steps should be taken to make this window as small as possible. One way to increase
the rate of inspecting kernel code is to use multiple cores to perform a concurrent inspection. Fastabend [11] explored creating a parallel version of the EPA-RIMM inspector on a platform with Coreboot firmware and showed linear scaling on 4 cores. However, Coreboot is currently less common than UEFI firmware.

In order to gain production use in data centers, a RIMM should run on platforms with UEFI. There is an opportunity to reduce the window of vulnerability and make better use of available resources by using multicore inspection.

### 1.2 Problem statement

RIMMs can inspect the kernel code for changes made by malicious kernel rootkits. To shield themselves from rootkits, RIMMs can run in System Management Mode (SMM). However, running in System Management Mode too long can disrupt or crash the operating system. EPA-RIMM already solves this by performing many partial kernel code inspections over many interrupts instead of inspecting the entire kernel all at once. This increases the total amount of time needed for inspection due to the overhead of entering and exiting System Management Mode. A shorter runtime would give rootkits less time to react. The UEFI implementations of EPA-RIMM could make use of multiple cores to do more work per partial inspection, lowering the total runtime for inspecting the kernel. Additionally, all cores already spend the time to enter SMM, yet only one core is used while the rest spin idle.

The existing multicore implementation of EPA-RIMM, referred to as EPA-RIMM-M, is written for Coreboot, but UEFI is a more ubiquitous platform. Performance-wise, EPA-RIMM-M was only evaluated for task time, since the
performance model at the time did not yet consider the run-time of other components, such as cryptographic operations. Lastly, it did not explore the design changes needed for the upper layers of EPA-RIMM to communicate to the lower layers the task assignments for specific cores without compromising the existing level of security.

1.3 Objective and approach

The objective of this thesis is to demonstrate a multicore SMM RIMM for UEFI-based systems. The approach is to leverage prior work and extend the EPA-RIMM prototype to perform multicore inspections, and then conduct a performance analysis. The extensions include modifications to EDK2 to allow multiple cores to be passed to an SMI handler, modifications to EPA-RIMM task scheduling to handle assigning tasks to multiple cores, and modifications to the task descriptions used to communicate task assignments. The performance analysis includes runtime data for various operations as the number of cores is scaled up.

1.4 Thesis statement

RIMMs running in SMM can improve performance and resilience by adding concurrency to both the integrity measurements and communication-related cryptographic operations on the monitored node.
1.5 Contributions

1. I have demonstrated how to improve the ability of existing RIMMs to detect rootkits by increasing the amount of measurement that can be accomplished in each SMM time window. Leveraging the existing EPA-RIMM prototype has the benefit of demonstrating that the approach works on virtualized systems and runs on ubiquitous UEFI firmware.

2. I have investigated and mapped the details of the SMM-related EDK2 reference UEFI implementation.

3. I have conducted experiments that detail the performance and effectiveness improvements over existing SMM RIMMs.

4. I have developed an approach for setting up multithreaded SMI handlers in UEFI
2 Background

2.1 UEFI

Unified Extensible Firmware Interface (UEFI) is a specification describing an interface between the OS\(^1\) and firmware [39]. UEFI was derived from (and replaced) Intel’s formerly proprietary Extensible Firmware Interface (EFI), which itself was a replacement for the BIOS interface. Today many companies are stakeholders in UEFI [41].

UEFI is now ubiquitous. According to OSDev.org, “All modern PCs ship with UEFI firmware and UEFI is widely supported by both commercial and open source operating systems” [60].\(^2\)

2.2 EDK2

EDK2 is an implementation of UEFI. Intel had already implemented EFI, UEFI’s foundation, in a project with the code-name TianoCore [37]. TianoCore was (for the most part) open-sourced under the BSD 2-clause license and has evolved into EFI Developer Kit II (“EDK II” or “EDK2”). The repository is available online [62]. The UEFI Forum does not endorse any particular implementation.

“EDK II” is also referred to as “EDK2,” especially in the source code and the project’s directory structure. Likewise, the name “TianoCore” is used occasionally in

---

\(^1\) In this thesis, the term Operating System (OS) is generally used to refer to the kernel.
\(^2\) UEFI is a specification, not an implementation. “Having” UEFI means a platform’s firmware has implemented some or all of the UEFI spec. Das U-Boot firmware has partially implemented some UEFI interfaces by writing translation layers around its own interfaces [15]. Thus, a system that uses U-Boot can be considered to have UEFI. Second-generation Chromebooks use Coreboot for hardware initialization with Das U-Boot as a payload. ARM-based Chromebooks use U-Boot alone. Newer Chromebooks use Coreboot with Depthcharge as a payload [61]. Depthcharge was specifically created to perform only the tasks required to boot Chrome OS [3].
place of “EDK II,” particularly in online documentation. “TianoCore” also refers to the community that contributes to the project. “EDK2” will be the name used hereafter.

Although its name (“EFI Developer Kit II”) references EFI instead of UEFI, EDK2 is an implementation of UEFI. Due to their origin, the terms “EFI” and “UEFI” are sometimes used interchangeably. This is especially true within the EDK2 project. It can be slightly confusing, but understandable in sources that predate UEFI.

2.2.1 EDK2 on open firmware platforms

Platforms are available today that allow the general public to develop and use custom firmware. Two such platforms are the UP2 (“UP Squared”) and the older MinnowBoard Turbot. In order to build EDK2 firmware for an open platform, one needs the core EDK2 code and platform-specific EDK2 code (which are both open-source) as well as a collection of platform-specific precompiled binaries. The precompiled binaries (known as the Binary Object Modules) are released by Intel and are closed-source. The officially stated reason is to protect the intellectual property of Intel and third-parties [21][33]. The license for these binaries explicitly forbids reverse-engineering, decompiling, or disassembling them. The functionality provided by some of these binary object modules is not limited to platform initialization, but also includes runtime services (including some SMI handlers), as will be explained later. Note that this binary package is therefore not perfectly analogous to the Intel Firmware Support Package (FSP) (required by other firmware systems, notably Coreboot), which is not used outside of initialization.
2.2.2 **EDK2 boot process**

The EDK2 boot process follows these phases in order: SEC, PEI, DXE, BDS, TSL, RT, and AF.

1. The Security (SEC) phase initializes temporary memory (such as Cache-as-RAM) and establishes root of trust.
2. The Pre-EFI Initialization (PEI) phase initializes all or some of main memory and performs chip and chipset initialization, bootstrapping enough of the system to be able to run the next phase.
3. The Driver Execution Environment (DXE) phase performs most of the initialization of hardware on the platform. Most drivers and SMI handlers are established here.
4. The Boot Device Selection (BDS) phase executes the platform boot policy.
5. The Transient System Load (TSL) phase hands off boot to the OS boot loader.
6. During the Run Time (RT) phase, the OS is in control, however, some UEFI runtime services are available for the OS to call. Runtime services include SMI handlers (though these are not usually intentionally called by the OS).
7. The After Life (AF) phase can (but is not required to) provide services such as allowing firmware to attempt recovery in case of an OS crash.

2.2.3 **SMM**

System Management Mode (SMM) is an operating mode on the x86 CPU. It is the most privileged mode. SMM was introduced on the i386SL (80386) in 1985. SMM execution is intended only for firmware code [20]. During the boot process, firmware code is loaded from the flash chip to main memory to be executed. After the firmware has initialized the hardware and launched the OS, most of this memory is released to the OS' control. Some firmware code persists past boot and remains available while the operating system is running. This includes code that provides runtime services. This code allows platform manufacturers (AKA Original Equipment Manufacturers, OEMs) to provide hardware handling that is OS-agnostic.
and that distinguishes their brand. An example use of SMM is for firmware to control power states and fan speeds in response to CPU temperature.

While in SMM, the running code has full privileges and access to everything on the platform. Many other chip architectures provide some kind of functionality similar to SMM. For example, ARM has TrustZone [10]. This type of mechanism is referred to in the UEFI Platform Initialization specification as Management Mode (MM) [40].

2.2.4 SMRAM

System Management RAM (SMRAM) is the name for the region of RAM assigned to hold the firmware code (text) as well as the runtime state (data). The CPU must be in SMM in order to access SMRAM—even kernel mode is not privileged enough. The System Management Range Registers (SMRR) prevent accesses to SMRAM when not in SMM and can also be used to prevent accesses outside of SMRAM while in SMM [20][5]. SMRAM can be relocated from its default location, although the SMRR necessitate it reside in the lower 4 GB of physical memory.

Each core has its own stack in SMRAM, which is allocated during initialization. (For CPUs that have Intel Hyper-Threaded Technology, one “logical core” is considered one core.) This stack is used when executing in SMM. There is also a heap in SMRAM that is shared by all cores.

On a system with EDK2 firmware, the code is copied from the flash chip to SMRAM by the SMM Initial Program Loader (SMM IPL), during (by default) the DXE phase, although this can be done in the SEC or PEI phases [49].
2.2.5 SMI

An x86 CPU enters SMM only after receiving a System Management Interrupt (SMI). The analogous interrupt on non-x86 CPUs may go by a different name depending on the chip instruction architecture and is referred to generally as a Management Mode Invocation (MMI) by the UEFI PI specification [40].

The System Management Interrupt (SMI) is the highest priority interrupt [20]. An SMI will cause a CPU to switch into System Management Mode (SMM) and jump to the top handler in firmware. The firmware handler code is located in SMRAM. The SMI is non-maskable (except that while in SMM, SMIs are masked, and a maximum of one SMI can be deferred until the running handler completes).

2.2.6 Issuing SMIs – synchronous vs. asynchronous

An SMI can be triggered by both hardware and software [19]. If an SMI is triggered by hardware, it is said to be asynchronous. If an SMI is triggered by software, it is said to be synchronous. Synchronous SMIs are also called software SMIs or SW SMIs. An asynchronous SMI can come from a variety of hardware sources. A CPU can even issue an asynchronous SMI to itself. This might be done, for example, if the CPU reaches a critical temperature. A repeating timer can be configured to generate asynchronous SMIs.

An interesting use of the synchronous SMI is for instruction patching and emulation. Instructions can be configured to trigger an SMI. The SMI handler can
then perform alternate instructions. This can be used to provide alternate execution to work around bugs in the chip design and microcode, or to facilitate emulation.

Another way to trigger a synchronous SMI is by writing to I/O port 0xB2, the Advanced Power Management Control Register (APM_CNT) [19]. This is the triggering mechanism the EPA-RIMM prototype uses because an out-of-band communication channel, such as the Baseboard Management Controller (BMC) is not available on the MinnowBoard platform. Upon a write to port 0xB2, all cores receive an SMI. The value written is used to specify which SMI sub-handler should run. A CPU is capable of self-issuing an SMI in a way such that other cores are not interrupted [26][40]. However, writing to port 0xB2 will issue an SMI to all cores. Some hardware interrupts will also issue an SMI to all cores [2].

2.2.7 SMM privileges

An SMI preempts the CPU, even if it is executing kernel code, after the currently-executing CPU instruction finishes (see Section 5.1 Details of SMM enter and exit for more info). While in SMM, the running code has full privileges and access to everything on the platform—memory, CPU register state, Model-Specific Registers (MSRs), I/O devices, etc. It is more privileged than kernel mode. During SMM, all interrupts are masked, so the CPUs cannot be preempted. The CPU remains in SMM until the SMI handler runs to completion. The OS is not directly notified of an SMI's occurrence. Thus, SMM is, in general, transparent to the OS. For these reasons, SMM operates as if it is in ring-2 [31].
Although SMM operates behind the scenes as far as the OS is concerned, it does not perfectly conceal itself. The OS, or another application with OS privileges, can detect the occurrence of an SMI by monitoring the `MSR_SMI_COUNT` register, which is incremented upon entry to SMM (except if there is an SMI Transfer Monitor—STM). A timing analysis can also be conducted to discover time windows in which the CPUs would appear to spend time unaccounted for by the system [45][44].

### 2.2.8 System perturbation

All interrupt handling is deferred while in SMM. Some interrupts may be time-sensitive, so remaining in SMM too long can cause notable perturbation of the system. Time-sensitive data, such as audio, may be lost. The system may eventually crash [8].

Therefore, the system designers recommend spending as little time in SMM as possible. The Intel BIOS Implementation Test Suite (BITS) defines 150 microseconds as the acceptable limit that an SMI should not exceed [38][8]. Delgado et al. (2013) analyzed SMM latencies and empirically found an upper limit of about 1.5 milliseconds to be acceptable [8] before user experience is degraded. Thus, in our work we keep SMM time under 1.5ms with the goal of keeping it under 150us (though the latter is difficult to achieve on the slower platforms our prototypes use). Note that no guideline is given recommending how frequently SMIs should be triggered.
2.2.9 Caching

Some prior SMM RIMM research papers mention that caching is disabled in SMM, resulting in slower execution [2][57]. This is no longer the case since the introduction of the SMRR. The SMRR feature, in addition to what is mentioned in Section 2.2.4, allows SMRAM to be cacheable [20]. Any cached values from SMRAM are rendered inaccessible after the RSM instruction completes. See Section 5.1 Details of SMM enter and exit for information on the RSM instruction.

The impact of cache misses on performance when resuming normal workload after using additional cores in SMM should be studied in future work.

2.3 Semantic gap and context reconstruction

In order for a RIMM to inspect certain system information, it must be able to locate the system’s saved context state. The location of this information depends on what was running on the CPU at the time it was interrupted. For example, consider a situation where the RIMM needs to inspect a hypervisor. If an SMI occurs while the hypervisor is running, the hypervisor context can be found in the SMRAM Saved State Map. If an SMI occurs while a VM is running, the SMRAM Saved State Map instead holds context information for the VM.

Researchers have referred to the need to reconstruct OS data structures from outside the OS as a semantic gap [6]. In the context of introspection, the semantic gap problem may involve translating virtual addresses, determining what the pre-empted execution context belongs to, locating process memory, etc.
2.4 STM

A given SMM task may not strictly require full access to the entire system. For example, adjusting the fan speed based on CPU temperature does not require access to OS memory. SMM has full privileges, making it an attractive target. Malware that exploits a vulnerability in SMM will gain full control over the system.

SMI Transfer Monitor (STM) was developed to provide a way to constrain an SMI handler's access to resources [19].

STM controls the privileges given to common SMI handling code by encapsulating it as a virtual machine (VM). This VM is known as the SMM Guest. The STM acts as a kind of hypervisor, or Virtual Machine Manager (VMM), for the SMM Guest. STM itself necessarily retains full privileges. Because of this, and the fact that it provides virtualization of SMM functionality, STM operates as if it were in ring -3.

STM is a critical pillar for Intel's Trusted Execution Technology (TXT), and TXT provides root of trust in the system. During boot, SMI handlers are established before TXT launch [19]. Intel's solution is to de-privilege the root SMI handler via STM. For example, memory access can be constrained so the SMI handler cannot read memory outside of SMRAM. Guests can make requests to STM via VMCall.

On systems that have a VMM host operating system, the STM is a “co-hypervisor” with the host VMM. In order for an STM to be initialized, both the OS and firmware must opt in during boot. This is called “Dual-Monitor Treatment.” The host VMM manages its guest VMs as usual. STM manages the SMM Guest. However, STM has full privileges, so it is not prevented from accessing the host VMM or its
guest VMs, though it is not the envisioned use of the design. It is intended that the SMM Guest (with its privileges constrained by the STM) run the handlers that must interact with the host VMM and its guest VMs.

Five VMX instructions are explained here: VMXON, VMXOFF, VMCALL, VMLAUNCH, and VMRESUME.

- **VMXON** activates the other instructions. It must be initially called with root privileges. The caller becomes VMX root.
- **VMXOFF** deactivates the other instructions.
- **VMCALL** is analogous to a system call; a guest VM can use a VMCALL to request services from the VMM.
- **VMLAUNCH** is used by the VMM to start a guest VM for the first time.
- **VMRESUME** If execution of a particular VM has been suspended somehow, the VMM can recommence executing the VM with VMRESUME.

There are two events, “VM Entry” and “VM Exit,” that are used when discussing VMX instructions. VM Entry and VM Exit are not instructions themselves. Any event that causes a VMM to switch to executing a VM is a VM Entry. For example, VMLAUNCH and VMRESUME both cause VM Entries. Any event that causes the VM to switch control over to the VMM is known as a VM Exit. For example, VMSCALL causes a VM Exit.

Intel has released a specification for STM [19]. Intel has also released an STM reference implementation that integrates with EDK2 [48]. This reference implementation includes a simple example VMM to demonstrate its capabilities. As mentioned before, the OS and firmware must both opt in during boot in order to enable STM. A member of our team has modified Xen to opt-in to STM [43]. This is publicly available [42].
2.5 Concurrency in SMI handlers

Multithreaded code is nothing new, so developers outside the realm of firmware may be surprised to learn that adding concurrency in firmware drivers is not a straightforward task. Threads are an operating system construct. The libraries for creating, scheduling, and synchronizing threads use OS calls. Firmware does not use the OS scheduler.

Firmware implementations do not provide robust library support for concurrency. However, they are capable of handling the basic multicore synchronization. EDK2, for example, provides basic spinlocks and spinning semaphore-style barriers. Synchronization is needed to prevent data races when entering and exiting SMM. Both EDK2 and Coreboot allocate, in protected SMRAM, a stack per core and a shared heap for running firmware. After entry, one core executes, while the others spin-wait. See Chapter 5 for more details on enter and exit.

Prior to EPA-RIMM, SMI handlers have not been multicore. Typical SMI handlers are not compute-intensive. They are usually designed to adjust a system critical function (e.g. CPU fan speed) and leave SMM as quickly as possible. These simple services do not need concurrent execution and can avoid the associated complexity. As mentioned previously, the guideline published by Intel in BITS [38] recommends that an SMI should take no longer than 150 microseconds. To address the system perturbation problem, recent Management Mode (e.g. SMM or
TrustZone) RIMM research efforts use an “asynchronous” design, in which a single core performs introspection while the others are allowed to execute as usual.

There is risk associated with providing multicore support in firmware and implementing multicore SMI handlers. SMM is the most privileged mode and the OS cannot prevent preemption. As SMM lacks advanced multithreading library support, there is consequently no protection from bugs such as deadlocks and race conditions. Such bugs can have serious impact, possibly leaving the system in a state that is unrecoverable without a power cycle. At the same time, it is notoriously easy to miss bugs in multithreaded code bugs during product development. If a design were to provide the general ability to write concurrent drivers, it should also provide ways to mitigate any issues introduced by concurrency. (Note that STM has a VM preemption timer.)

Examining firmware can reveal information on proprietary chip architecture implementation. Also, the same reasons that make firmware a desirable location for RIMMs similarly make it a high-value target for malicious actors. Thus, firmware on commodity systems is typically closed-source and firmware changes are restricted to updates from the manufacturer. This poses a research challenge. Fortunately, open-source firmware implementations are available as well as a variety of open-firmware platforms. The MinnowBoard is an inexpensive, open-hardware platform. It comes in several models. The MinnowBoard Turbot Quad-Core has an Intel Atom E3845 processor. It supports both EDK2 and Coreboot.
EDK2 is open-source and is Intel’s preferred implementation of the UEFI specification [37]. Coreboot (previously LinuxBIOS) is an open-source firmware implementation that is not attached to a particular specification. EDK2 is designed such that all handlers will be executed sequentially on one core.

### 2.6 Concurrency in RIMMs

Server-class machines, for which EPA-RIMM is intended, have many cores. It is not uncommon to have around 64 cores. There are currently no multithreaded RIMMs that run in SMM, with the exception of EPA-RIMM-M [11], which is written for Coreboot.

Besides being an obvious optimization, there are many ways an SMM RIMM can benefit from using all available cores. As mentioned before, when an SMI occurs, every core switches context to enter SMM. One core then executes the appropriate SMI handler while the others spin-wait. For security purposes, EPA-RIMM intentionally preserves this property of keeping the non-worker cores waiting in SMM until the SMI handler completes [10]. As described previously, EPA-RIMM may spread the inspection of a code region over many partial inspections, with each SMI triggering one partial inspection. The amount of memory examined in a partial inspection is chosen in order to constrain the runtime to a given limit. Since an SMI causes all cores to enter SMM (with only one core doing work) and SMM is prescribed to run for a specific amount of time, the unused cores may as well be put to work.
The overhead involved in entering and exiting SMM is not insignificant. Therefore it makes sense to try to get as much done as possible while in SMM to amortize the overhead over more work. Also, when the AP cores are spin-waiting, they are wasting time that could be spent on useful work.

Some Management Mode RIMMs (XHIM and SATIN) run asynchronously, returning all but one core from the privileged management mode instead of busy-waiting [28][44]. In this design, inspection runs concurrently with the rest of the system. This is an alternative method of alleviating system perturbation. The scrubbing rootkit named TZ-Evader can detect when one core is commandeered for the purposes of running a RIMM [44].

Using more cores should reduce the length of time required to inspect the kernel. This in turn can mean shorter detection times, making it harder for scrubbing rootkits to evade detection. The current time to inspect all 12MiB with 22 decomposed segments of size 8KiB per second is 70 seconds [11]. Given 4 cores, this could be reduced to about 17.5 seconds if perfect scaling were achievable. Perfect scaling in multithreaded code is generally difficult to accomplish. Typically a proportion of the algorithm cannot be parallelized. There is also increased communication overhead. However, in this example, no communication is required between cores except at the beginning and end of inspection and the inspection itself is read-only. Fastabend was able to achieve near-perfect scaling with EPA-RIMM-M [43].
2.7 DMA and IOMMUs

Direct Memory Access (DMA) allows reads and writes to main memory to be performed by a device, offloading the CPU. Devices generally use real physical (host physical) addresses in DMA requests. The desired physical address must be communicated to the device, commonly by a driver.

Many modern CPUs have an Input / Output Memory Management Unit (IOMMU) that allows the host to setup virtualization for DMA. A PCI device that makes a DMA request to (what it thinks is) a physical address can be redirected to a different memory address by the IOMMU. This process is transparent to the device.

One benefit of an IOMMU is that it helps VMs use PCI devices. Since VMs use guest physical addresses and do not know the host physical addresses, a virtualization-unaware device driver running on the VM would specify a guest physical address when configuring the device. The device would then attempt to access this address. Without a guest-physical-to-host-physical translation layer, the device is likely to access the wrong memory. Adding an IOMMU would help in this example because the hypervisor could configure it to apply address translation to memory accesses from that device. The device could continue to use guest-physical addresses and the IOMMU will automatically translate those accesses to host-physical addresses.
3 Existing EPA-RIMM Architecture

This chapter describes the pre-existing EPA-RIMM [9][10][43][11][7] design. This thesis extends EPA-RIMM research, so many design specifications remain unchanged. Our modifications to the original EPA-RIMM design and prototype are discussed in detail in Chapter 6. In this chapter, the architecture is described in its state prior to any changes proposed by this thesis.

EPA-RIMM stands for Efficient, Performance-Aware Runtime Integrity Measurement Mechanism. EPA-RIMM is a framework for kernel and hypervisor rootkit detection. An inspection module in SMM on the monitored node is directed by encrypted messages from a remote node. EPA-RIMM decomposes long-running checks into multiple smaller tasks to prevent fatal perturbation to system functions.

As shown in Figure 3-1, EPA-RIMM specifies an Inspector, a Host Communications Manager (HCM), a Backend Manager (BEM), and a Diagnosis Manager (DM). These components communicate Checks, Check Results, Tasks, Task Results, and Bins.

The Diagnosis Manager runs on a remote node. It sends Check instructions to the Backend Manager. When Checks have been completed, it stores the results.

EPA-RIMM has implemented memory checks and register checks. Memory checks can read a specified range of memory, hash it with a cryptographically secure hashing algorithm, and compare against an expected result. Register checks will read a register from the SMRAM saved state map, which is saved to SMRAM by each core by microcode upon receiving an SMI. This can be compared with an expected
value. Memory checks need a secure initial provisioning step to determine the expected hash for a given region. Currently, provisioning is done on the first measurement taken for a region.

The Backend Manager is the component that decomposes Checks into one or more Tasks and sends them to the Host Communications Manager. When the Tasks are completed, it sends a Check Result back to the Diagnosis Manager.

Tasks are packed into Bins. One Bin contains the Tasks that are to be completed in a single SMI. The number of Tasks in a Bin can be changed to affect the amount of time spent in SMM. There is overhead involved in invoking an SMI, so it is desirable to get as much work done as possible. However, too much time spent in
SMM will perturb the system by causing the OS to delay handling or miss interrupts, in turn causing a number of possible effects, such as noticeable jitter, inaccurate process accounting, and remaining in higher power states [8]. As there is no pre-emption in SMM, function runtimes are much more consistent in SMM than user space. Note that STM provides preemption capabilities.

The Host Communications Manager runs on the monitored node. It passes Bins from the Backend Manager to the Inspector and triggers the inspector with SMIs. When the inspection is done, it returns the Bins to the Backend Manager. A communication side-channel, such as the Baseboard Management Controller (BMC) used by HyperSentry [2], is needed to prevent the potentially-compromised host from tampering with the Bin. (Note that the EPA-RIMM prototype platform is the MinnowBoard, which does not have a BMC, so it does not demonstrate the use of a side-channel.)

Communication between the Backend Manager and the Inspector are HMAC-ed then encrypted. HMAC provides message integrity and authentication via shared secret, and encryption provides privacy via shared secret. A particular algorithm is not specified, but the prototypes SHA-256 for HMAC and AES-256-CBC for encryption. Since these cryptographic algorithms use secret keys, a provisioning step is required to setup the keys. Delgado says, “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]. Because
key exchange would occur in a provisioning step, it would have negligible impact on runtime performance for both the pre-existing EPA-RIMM research as well as the extensions suggested in this thesis.

The Inspector executes in SMM and is triggered by SMI. This is intended to provide stealthy [2][43] and unpreventable preemption of the host. The Inspector can inspect host memory as well as CPU registers, including Model-Specific Registers (MSRs). When the Inspector is done performing all the Tasks in the Bin, it returns Task Results to the Host Communications Manager.

The first EPA-RIMM research demonstrated inspection of the Linux kernel [10]. The inspector is written as an EDK2 module. EDK2 is Intel’s (mostly) open-source reference UEFI implementation. This prototype was later extended to include inspection of the Xen hypervisor [43]. Another prototype was written for Coreboot [11] and was the first RIMM to execute a concurrent SMI handler. It inspected the Linux kernel.

EDK2 is mostly open-source. The code responsible for handing off execution to the appropriate SMI handler is closed-source, released as a precompiled binary—one of the Binary Object Modules required to build the firmware image. The license agreement of this module specifically prohibits reverse engineering this binary. In contrast, SMI handlers in the Coreboot implementation are open-source.

On systems with hypervisors, SMM RIMMs have an additional challenge to get an in-context measurement. After entering SMM, the inspector does not know if it is currently operating in the context of the hypervisor or a virtual machine [2].
HyperSentry solves the semantic gap by usurping the CPU’s performance counters. EPA-RIMM solves this by using an STM and reading the memory region that saves the VMCS during a VM context switch into the STM.

EPA-RIMM relies on preemptioning all cores. This gives EPA-RIMM a window to operate wherein any rootkits that have gained OS or hypervisor privileges cannot run [11]. The prototype triggers an SMI on all cores by writing to I/O port 0xB2.

The authors of Copilot [30] have outlined a set of requirements: unrestricted memory access, transparency to the host, independence from the host, sufficient processing power, sufficient memory resources, and out-of-band reporting [30]. EPA-RIMM fulfills these requirements to some degree. It has unrestricted access to memory, although the use of this level of access is atypical for an SMI handler. (In fact, one of the purposes of STM is to restrict access for SMI handlers [49].) It is independent of the host. SMM invocations are somewhat transparent when using STM. (The SMI counter register is incremented when not using STM [43].) However, malware can detect SMIs through timing analysis [44]. EPA-RIMM uses a remote diagnosis manager to store golden hashes. It has sufficient processing power because it uses the system CPU, so performance scales with the system, although cryptographic operations remain a bottleneck and care must be taken to avoid system perturbation. Lastly, EPA-RIMM calls for an out-of-band communication and SMI-triggering mechanism, such as the BMC [10]. This was demonstrated by HyperSentry [2]. EPA-RIMM prototypes have not implemented this, as the
MinnowBoard open firmware platform was used and does not have an out-of-band mechanism [10].
4 Related Work

4.1 Copilot

Copilot is a “coprocessor-based kernel integrity monitor for commodity systems” [30]. It uses an evaluation board plugged into a PCI slot (the “monitor”) to inspect a “host” machine’s OS. The monitor is controlled by a dedicated out-of-band line from an “admin station” machine to the monitor card. The monitor accesses memory with DMA, allowing it to circumvent the OS. It calculates an MD5 hash of sections of kernel memory and compares them against previously provisioned known-good hashes.

Copilot always runs in parallel with the host and can only check memory. Because EPA-RIMM uses SMM, it is capable of examining the state of execution of the host. The majority of Copilot’s perturbation to the host is due to memory and PCI use. Both Copilot and EPA-RIMM inspect regions of kernel memory by hashing and must deal with provisioning and human interpretation of mismatched hashes. EPA-RIMM uses the cryptographically secure SHA-256 algorithm.

Since Copilot [30] was published, IOMMUs have become more common. A RIMM that requests memory for inspection through DMA is vulnerable to a compromised hypervisor on platforms with an IOMMU. The hypervisor could configure the IOMMU to redirect those requests to a different memory region. This redirection would be transparent to the RIMM.

For example, a hypervisor that has had its text region altered by a rootkit could keep a copy of the unmodified kernel text region in a different part of
memory. It could then configure the IOMMU so DMA requests from a RIMM running on a PCI device are silently redirected to the unmodified kernel text region. When the RIMM attempts to inspect kernel memory (which has been modified by the rootkit), it will actually be inspecting the pristine, unmodified copy, and the RIMM would detect no changes.

4.2 HyperSentry

HyperSentry [2][1] provides a framework for triggering a measurement agent to inspect a hypervisor. The authors showed an agent running in SMM verifying a host running Xen. The focus was more on the techniques for running an agent securely than the agent itself; however, performance data for their prototype agent is provided.

The framework was developed with several goals in mind. Stealthy out-of-band invocation via the Intelligent Platform Management Interface (IPMI) on the Baseboard Management Controller (BMC) is intended to allow a measurement to be triggered without alerting malicious code running in the hypervisor. To maintain integrity, the agent is verifiable before invocation. It is deterministic, in that it cannot be changed while running. The hypervisor cannot interrupt the agent, as interrupts are disabled and the Interrupt Descriptor Table is temporarily modified during invocation to block NMIIs. The necessary context, such as hypervisor memory and CPU register state, is provided to the agent. The measurement result is attestable through cryptographic signing with asymmetric keys generated at boot. It is assumed that the platform is equipped with trusted boot hardware.
When the agent is invoked, the system is completely halted until the agent completes. The average total end-to-end runtime was measured to be 35ms. Delgado and Karavanic [8] found that halting even for relatively short periods of time (1.5 milliseconds) could cause the system to crash or miss important interrupts. For this reason, EPA-RIMM decomposes a long check into many SMI check invocations that are short enough so as not to crash the system.

HyperSentry uses an instruction injection scheme with an initial SMI to force the BSP to VM Exit before using a second SMI to initiate the measurement processes. This guarantees that the BSP is executing as VMX Root, giving it hypervisor context. The measurement agent’s code is in the hypervisor, but it is verified with SHA-1 in SMM before execution. The measurement is done in protected mode (with hypervisor context) instead of SMM. A third SMI is then triggered for results storage in SMRAM.

Vibhute [43] extended EPA-RIMM to use an STM with Xen. She added a VMCall that returns a pointer to the hypervisor’s context. If the SMI was triggered from VMX Root, the VMCall finds the information in the SMRAM Saved State Map. If the SMI was triggered from VMX non-root, the VMCall finds the information in the host region of the saved VMCS.

4.3 HyperCheck

HyperCheck [57][14] is a hardware-assisted tampering detection framework. It has three components. The physical memory acquisition module and CPU register checking module are on the monitored target machine. The analysis module is on a
remote monitor machine. On the target machine, a PCI network card uses DMA to read hypervisor, guest, or OS memory. The network card then transfers this snapshot to the monitoring machine where it is compared with a known-good snapshot. Device drivers for the network card were moved into SMM for protection, and the inspection is triggered with an SMI from the network card.

HyperCheck was tested on Xen hypervisor, Xen's Dom0, bare-metal Linux, and bare-metal Windows. There are two implementations: HyperCheck-I and HyperCheck-II. HyperCheck-I injects code into the closed-source UEFI. (Note that EDK is now mostly open-source.) HyperCheck-II uses open-source Coreboot.

The target machine's CPU was a single-core Intel Pentium 4. The time spent in SMM for HyperCheck-II for a single invocation on the target machine was in the millisecond range, about 90ms to 110ms.\(^3\) This amount of time exceeds the limit beyond which serious system perturbation will occur [8]. EPA-RIMM decomposes long inspections into many SMI check invocations small enough to avoid crashing the system.

Since EDK2 is now open source, EPA-RIMM does not need to abnormally inject code into firmware, but can build it into the firmware using the prescribed build process. The HyperCheck authors found it time-consuming to add functionality to HyperCheck-I in firmware, due to needing to reverse-engineer and program in assembly. This was part of their motivation for implementing HyperCheck-II remains in SMM until the NIC completes sending [14]. Based on the breakdown graph, the entire process takes about 90ms. It is also stated that benchmarking measured with HyperCheck-II triggering once per second has an 11% overhead over the benchmarking of the system without triggering HyperCheck-II. 11% of one second is 110ms.

---
\(^3\) HyperCheck-II remains in SMM until the NIC completes sending [14]. Based on the breakdown graph, the entire process takes about 90ms. It is also stated that benchmarking measured with HyperCheck-II triggering once per second has an 11% overhead over the benchmarking of the system without triggering HyperCheck-II. 11% of one second is 110ms.
HyperCheck-II in Coreboot, which allowed them to write it in C. EDK2 is open-source now, which has allowed us to write the inspector into the firmware using C. Like HyperCheck-II, EPA-RIMM-M [11] is implemented in Coreboot. The EPA-RIMM prototype inspects memory by comparing a hash instead of a snapshot.

4.4 SPECTRE

SPECTRE [56][14] is another SMM RIMM by the authors of HyperCheck [57][14]. (SPECTRE is not to be confused with the 2019 speculative execution attack of the same name [24].) The framework calls for a target machine and a monitor machine. The monitor machine receives alerts from the SMM module on the target machine. The monitor machine expects to receive regular “heartbeat” notifications from the SMM module. The timer is setup to continuously trigger hardware SMIs at a regular interval, but the authors suggest using a random scheduler to make it more difficult for malware to predict and evade an inspection.

The prototype ran on a single-core, non-HyperThreaded CPU and was written for Coreboot/SeaBIOS like HyperCheck-II. Although capable of reading hypervisor memory, the authors did not evaluate on a hypervisor for this paper as they did with HyperCheck. It runs on bare-metal Windows and Linux.

SPECTRE offers more types of memory inspections than HyperCheck—having modules that detect heap spray attacks, heap overflow attacks, and rootkits. Rootkits are detected with basic hashing. The SMM module can perform inspections while in SMM or alternatively offload hashing by sending a memory snapshot to the monitor machine.
SPECTRE demonstrates bridging the semantic gap. It can walk the page table to translate virtual addresses to physical addresses. On Windows it finds the Kernel Processor Control Region and uses this to eventually locate the list of running processes and their associated heaps. On Linux, it uses the System.map file to find a specific process in the list of running processes. It can then follow the links to find other processes and eventually their heaps. It can then scan each heap for evidence of attacks, such as the presence of a NOP sled or broken links. SPECTRE takes 5ms-8ms just to reconstruct semantic information and more time when running the heap spray detection and heap overflow detection modules.

4.5 SATIN

SATIN [44], developed by Wan et al. is a RIMM that runs in ARM TrustZone. Wan et al. described a vulnerability that occurs when single-core RIMMs are executed on multicore hardware and are allowed to run concurrently with Normal World processes. This attack is named “TZ-Evader”. By probing with a thread fixed to each core, malware can measure the timing of every core’s scheduling. If a core takes an unexpectedly long time to be rescheduled, it is surmised that the core is being used to perform an inspection. This information can be used by malware to detect when inspection begins and temporarily remove malicious traces from the host before the inspecting core reaches it.

Wan et al. designed SATIN to be unsusceptible to the TZ-Evader attack. SATIN executes on one randomly selected core at a time, and it is scheduled to run at random times. To avoid leaking inspection information to a potential TZ-Evader
attack, SATIN measures the kernel in smaller sections so as to take less time. This keeps inspection time below the required threshold for TZ-Evader to infer the core’s activity. The region of memory to scan is randomized to be less predictable. The introspection core is also chosen at random. TZ-Evader is occasionally able to detect SATIN, but not with enough time to scrub its traces.

The most obvious difference between EPA-RIMM and SATIN is that EPA-RIMM uses x86’s SMM while SATIN uses Arm’s TrustZone. TrustZone is purpose-built with security in mind, while SMM was intended for the firmware vendors to provide custom hardware handling.

SATIN, like EPA-RIMM, also decomposes larger checks into smaller ones to keep the runtime low. For both systems, the purpose is to minimize the time spent in management mode. SATIN’s goal is to make it harder for TZ-Evader to detect that inspection has been invoked. EPA-RIMM’s goal is to prevent system perturbation.

Regarding evasion attacks, the authors state, “As long as the normal world is not frozen on all cores, there exists a race condition between the normal world to remove the attacking traces” [44]. EPA-RIMM indeed invokes all cores to enter SMM. This would apparently make EPA-RIMM invulnerable to TZ-Evader by definition, with the caveat that x86 waits for the current instruction to retire before entering SMM. However, TZ-Evader’s required detection time is on the order of milliseconds. EPA-RIMM itself should not spend more than 1.5ms in SMM to prevent system perturbation [8].
They observed that SATIN could hash about 1.16 MiB of the OS (10%) before the introspection is detected by TZ-Evader’s KProber. Thus, they scanned memory chunks of this size (1,218,351 bytes). The hash function used by SATIN, djb2, is faster than EPA-RIMM’s SHA-256, but is not cryptographically secure. EPA-RIMM generally scans memory in chunks of 512 to 8,196 bytes.

SATIN waits a random amount of time before starting the next inspection in the range \([0, 2 \times t_p]\) where \(t_p\) is some base period, such as 8s or 16s. EPA-RIMM could become less predictable by using such a rescheduling scheme.

### 4.6 STM/PE

STM/PE [28], developed by Myers at the NSA, also uses STM. “PE” stands for protected execution. STM/PE adds additional SMM guest VMs to the single SMM guest STM provides. These are called PE/VMs and can be used to run software protected from interference of a compromised OS. These PE/VMs are allocated one core. Myers released open-source contributions that add an STM to the Coreboot firmware.

STM/PE demonstrates enhancing STM to provide additional guest VMs that have protected execution. The inspection agent runs in a PE/VM. It is an existing RIMM called XHIM, based on Linux Kernel Integrity Monitor (LKIM), designed to inspect Xen.

With STM/PE inspection runs in parallel with execution of the OS. EPA-RIMM, on the other hand, preempts execution of the OS, though long inspections must be broken up into several preemptive SMIs so as not to crash the system.
5 A Study of SMI Handler Implementation

This chapter describes the execution flow that EDK2 follows when an SMI is received. See Appendix for specific function names and file names used by EDK2.

5.1 Details of SMM enter and exit

When an SMI is received by a core, it is handled on an instruction boundary after all instructions retire and stores complete [20]. Some specific tasks are then performed by microcode:

- SMM is enabled.
- The register context is saved to a special table in SMRAM called the “SMRAM Saved State Map”.
- The address mode is switched to 16-bit.\(^4\)
- The instruction pointer is set to an entry instruction in SMRAM.

The entry instruction is the common SMI handler, used for all SMIs, that eventually selects the appropriate sub-handler. It is always placed at the same offset from the SMBASE register. Each core has a different SMBASE and therefore has a different entry point. In EDK2, the code at the initial SMI entry handler is duplicated for all cores, and this initial entry handler code contains a jump to a common label.

---

\(^4\) Although the address mode is switched to 16-bit by the microcode, one of the first things the SMI entry handler does is switch to 32-bit mode.
To leave SMM, the code must use the RSM instruction. The RSM instruction can only be called from SMM. It performs the necessary steps to restore the execution state before the SMI. It is called per-CPU. The microcode reverses the steps taken above:

- The register context from the values saved in the SMRAM saved state map is restored.
- The address mode is returned to what it was in the interrupted state.
- SMM is disabled.
- The CPU jumps back to wherever it left off when the SMI occurred.

5.2 EDK2 SMM execution flow (no STM)

This section is specific to the EDK2 firmware design. There are two execution flows that can occur upon entering SMM. The first is the default flow. The second flow occurs when the firmware has an SMI Transfer Monitor (STM). This section explains the default flow.
Figure 5-1: EDK2 SMI Execution Flow (Blackbox)

Figure 5-2: EDK2 SMI Execution Flow (Clearbox)

Figure 5-1 shows a high-level SMM flow diagram. Figure 5-2 shows the same flow is shown with more detail. The numbers shown in Figure 5-2 indicate the order in which execution occurs. These steps are described here. After jumping (1) to the entry point, the code eventually calls (2) into a CPU rendezvous function.
Up until this point, all processors have followed the same instruction flow. Now, one processor is selected to service the SMI while the others spin-wait. The processor selected to execute the service routines is called the Boot-Strap Processor (BSP). This designation originates from the initialization phase of EDK2, in which a single processor is used to setup the SMRAM execution environment. The other processors are called Application Processors (AP).

The APs spin-wait and the BSP calls into a module named the PI SMM Core. There is a list of SMI handlers. These would have been registered during platform initialization. The SMI handlers may themselves have lists of sub-handlers. Although not completely precise, it might help to imagine the handlers as nodes in a tree-like structure in which each node has 0 or more children. The deepest nodes in the tree (the leaves) are the handlers (or “drivers”), typically provided by platform vendors, that provide the custom handling for a specific SMI. The correct driver(s) are selected based on the SMI source, SMI type, and/or other arguments passed from Non-SMM space (such as the value written to port 0xB2 or information in the comm buffer [52]).

After the BSP runs all the handlers (3), (4), & (5) for the SMI, it returns (6) to the processor rendezvous code and signals the APs to exit their wait. After returning (7) to the entry point, each core then (8) calls the RSM instruction to leave SMM.
Section 5.2: EDK2 SMM execution flow (no STM)
5.3 EDK2 SMM execution flow (with STM)

See Figure 5-3 for a high-level SMM flow diagram when EDK2 has an STM. Figure 5-4 shows the same flow in more detail. During Dual-Monitor Treatment, an SMI (1) is a VM Exit from the VMM or guest VM to STM. The STM then uses VMResume (2) to run the SMM Guest. The entry point is the common SMI handler used in Non-STM operation. (Note the similarity of the box labeled “SMM Guest” to Figure 5-2.) Therefore, the handling of the SMI unfolds as in the Non-STM flow, except that access to protected resources may be deprivileged by the STM (as configured by the programmer). STM-aware handlers can make hypercalls to STM via VMCall if desired. Eventually each core calls the RSM instruction (9). Instead of exiting SMM, as would normally happen in the absence of STM, the RSM instruction is a VM Exit from the SMM Guest to STM. STM then calls VMResume (10) to return execution to the VMM or guest VM, whichever was running when the SMI occurred.

With STM and non-STM flows alike, an SMI uses the SMBASE register plus an offset as the entry point for SMM. The difference when using STM is that is address has been configured to point to STM’s entry point. (This is shown as the box labeled “Special VM Exit Handler” in Figure 5-4.) Note that the microcode handles the SMRAM saved state map differently. In the presence of an STM, the microcode saves the register context to the Virtual Machine Control Structure (VMCS) instead of the SMRAM saved state map. The STM synchronizes the interrupted VM’s VMCS with the SMRAM saved state map before VMResume of the SMM Guest, and again before
VMResume of the interrupted VM. Note also that the MSR_SMI_COUNT is no longer incremented upon SMI.
6 Design

EPA-RIMM’s SMM Inspector was single-core. This thesis extends pre-existing EPA-RIMM research to allow the SMM Inspector to perform inspections with multiple cores in parallel in UEFI. All cores are invoked into SMM for inspection, paying the time price of running the SMI entry and exit handlers, yet all but one do no useful work. At the same time, leveraging more cores for inspection could scale the amount of work done during inspection and lower the overall time it takes to measure the entire kernel.

Fastabend [11] implemented multicore memory measurement Tasks for EPA-RIMM in Coreboot and the results showed good linear scaling. In contrast, this thesis presents a UEFI implementation. Although Coreboot is gaining in popularity, UEFI is still more common. This thesis looks in depth at the changes necessary in EDK2 in order to enable multicore SMI handlers as well as the changes to EPA-RIMM that are needed to communicate Tasks for multiple cores from the BEM to the Inspector.

Four changes are needed to allow EPA-RIMM to leverage concurrency. First, firmware needs to be modified to call the inspector with all cores. Second, task decomposition and scheduling must be adjusted to use the additional cores. Third, the EPA-RIMM inspector needs to be modified to perform inspection with all cores. Fourth, Task communication needs to be modified to allow not only for the BEM to tell the Inspector which core to run a Task on, but also to allow the Inspector to perform Task packing/unpacking and encryption/decryption concurrently.
6.1 Firmware modifications to provide multicore SMI handlers

The inspector is a client of the EDK2 architecture. EDK2 allocates only a single CPU to run handler code (see Figure 5-4). However, all CPUs are switched into SMM upon SMI, and are merely held back in the “CPU Rendezvous” block. In order to enable a concurrent inspector, EDK2 must be modified to call the inspector with all cores.

We have identified three ways to modify EDK2 to allow concurrent execution of an SMI handler for non-STM systems and an additional two for systems with an STM for a total of five possible approaches. This is not necessarily an exhaustive list. The figures in this chapter are modifications of the figures from Chapter 5. Figure 6-1 shows where EPA-RIMM ran prior to this thesis. It is a “driver” SMI module.

Although the prototype for this demonstration does not include STM, the potential presence of STM was considered because EPA-RIMM provides support for STM. Note that the figures in this chapter show STM and an SMM Guest. On systems without STM, the SMI sends execution from Non-SMM to the Entry Handler block. In other words, the flow inside the SMM Guest is the same for systems with and without STM unless mentioned.
Section 6.1: Firmware modifications to provide multicore SMI handlers

Figure 6-1: Single-core UEFI EPA-RIMM runs as a driver

Figure 6-2: Possibility 1 – Modify Rendezvous and Dispatcher to pass multiple cores. This is not possible without access to the close source or somehow using a different dispatcher.
6.1.1 Possibility 1: Modify EDK2 dispatch code

See Figure 6-2. The SMI Rendezvous is responsible for holding back the APs and sending the BSP to call into the PI SMM Core (the firmware component that handles SMI handler dispatch, among other things). This function could be modified to send all CPUs into the PI SMM Core. A system would be designed for SMI handlers to indicate the desired number of cores during handler registration. Since all existing handlers are single core and would not be written to accept additional cores, the default number of desired cores should be one. The PI SMM Core would be modified to be re-entrant, along with any dispatchers and anything else for which concurrent child handlers are desired. A benefit of this approach is that it would work on systems both with and without an STM and does not require any modification to the STM if the system has it.

The EPA-RIMM prototype’s inspector is registered as an SMI handler using the SW Dispatch Protocol. On the MinnowBoard, we observed this assigns it as a sub-handler of a root handler called the PchSmiDispatcher. Unfortunately, this is one of the closed-source binary object modules. It is not available for modification by us or anyone without the source code. It would have to be done by the code owner, Intel, or by a third party with permission.

It could be possible to register the SMI handler using the newer SW Dispatch Protocol 2. This may register the SMI handler under one of the open-source modules. However, we did not explore this.
We did not choose to implement this approach. It would be a fairly involved design change requiring a moderate amount of expertise in EDK2 architecture. The PchSmiDispatcher is closed-source, so it was not possible to review the source code to determine if it is safe for multicore execution. Any code initially designed under the assumption of running single-core might have race conditions that cause undefined behavior when run with multiple cores without adding additional synchronization.
6.1.2 Possibility 2: Reroute SMM flow during SMI Rendezvous

See Figure 6-3. The SMI Rendezvous is responsible for holding back the APs and sending the BSP to call into the PI SMM Core. This function can be modified to check the source of the SMI. If the SMI was triggered by EPA-RIMM, the function can immediately have all CPUs call directly into the SMM Inspector entry point. After inspection is complete, the SMI status flags need to be properly cleared.

This approach diverges from the modular design of EDK2’s SMI handling. It is

Figure 6-3: Possibility 2 – If the SMI was triggered by EPA-RIMM, reroute the execution flow to call the SMM Inspector with all cores. Otherwise, follow the normal single-core execution flow.
not an elegant modification and therefore may not be preferred in production firmware. On the other hand, it is a simple change that can be verified easily. This is suitable for research purposes to measure the performance of EPA-RIMM. This approach would allow the inspector to run concurrently on systems both with and without an STM. It does not require any modification to STM for systems that have an STM.

Figure 6-4: Possibility 3 – Add multicore dispatcher
6.1.3 Possibility 3: Add a concurrent driver dispatch module

See Figure 6-4. SMI drivers are selected from a couple different root handlers. These are installed as modules. The existing modules can be kept for compatibility and an additional module can be added along with a new protocol for registering concurrent drivers. The rendezvous code must be modified to send all cores into the PI SMM Core. At registration, modules would have a way to specify the number of cores desired. The existing root handler would be modified to direct execution of the desired number of cores to the dispatch modules. (See Appendix for more details about execution flow.) This would enable multicore SMI handlers in general (not just EPA-RIMM).

Unlike Possibility 1: Modify EDK2 dispatch code, only the PI SMM Core entry point would be modified to be reentrant. Additionally, it would allow the ability to create other multicore SMI handlers without big changes to the interface. This approach is a compromise between Possibility 1: Modify EDK2 dispatch code and Possibility 2: Reroute SMM flow during SMI Rendezvous. Some functions must be made reentrant, but not so many that it is difficult to verify correctness. It would be a bit of work to implement, but not a massive refactor.
6.1.4 Possibility 4: Put the Inspector inside STM

See Figure 6-5. The STM code can be modified to check the source of the SMI. If the SMI was triggered by EPA-RIMM, STM can avoid resuming the SMM Guest and run the EPA-RIMM inspector instead. After inspection completes, STM calls VMResume to return to the pre-SMM context.

This approach circumvents some of the protections that made us want to use STM in the first place. Specifically, the Inspector is not deprivileged. A vulnerability in the inspector equates to a vulnerability in STM, the most privileged part the system. However, the SMM guest could still be as deprivileged as possible.

This approach would allow the inspector to run concurrently on systems with an STM, but not on systems that do not have STM. It does not require any
modification to the SMM Core code.

6.1.5 Possibility 5: Put SMI handler in additional SMM Guest

See Figure 6-6. The default implementation of STM only has one VM, the SMM Guest. A second VM can be launched to encapsulate the EPA-RIMM inspector. Note that this method is used by Myers to launch single-core Protected Execution (PE) environments [28].

The STM code can be modified to check the source of the SMI. If the SMI was triggered by EPA-RIMM, STM can avoid resuming the SMM Guest and run the second SMM guest instead. This guest will encapsulate the EPA-RIMM inspector code.

The benefit of this approach is that custom deprivileging can be applied to the second SMM guest. It does require some small change to the STM code. This

![Figure 6-6: Possibility 5 – Put the Inspector in a dedicated SMM Guest](image)
approach would allow the inspector to run concurrently on systems with an STM, but not on systems that do not have STM. It does not require any modifications to the SMM Core code.

6.1.6 Selected approach

The approach chosen in this thesis is **Possibility 2: Reroute SMM flow during SMI Rendezvous**. This approach may not be a clean or elegant solution by industry standards, but it is suitable to demonstrate the viability of concurrent SMI handlers and provide realistic performance measurements. It also should work on systems both with and without STM, although this thesis does not evaluate a prototype with STM.

6.1.7 EDK2 modification specifics

Source file *MpService.c* (*edk2/UefiCpuPkg/PiSmmCpuDxeSmm/MpService.c*) contains a function called *SmiRendezvous*. This function selects the BSP and calls function *BSPHandler*. The APs call *APHandler* instead. Inside these functions, the BSP signals back and forth with the APs to coordinate some tasks. Eventually, the BSP calls into the PI SMM Core with a call to *gSmmCpuPrivate->SmmCoreEntry (&gSmmCpuPrivate->SmmCoreEntryContext)*.

Immediately before this call, *BSPHandler* will be modified to read the *APM_CNT* register (0xB2). If the value of this register matches the registered EPA-RIMM SMI handler value, *BSPHandler* will call a function pointer to the EPA-RIMM concurrent inspector entry point. The SMI handler value and the function pointer
will be added as global variables. These variables are to be set by the EPA-RIMM registration code during initialization in the DXE boot phase.

A matching change will be made to the corresponding section of APHandler (immediately prior to the part that spin-waits for the BSP to complete in the normal flow). After the EPA-RIMM inspector completes, execution returns to the calling function (either BSPHandler or APHandler) and then continues as in the normal flow.

See Appendix for more details on SMI enter and exit code execution flow in EDK2. As mentioned in Possibility 2: Reroute SMM flow during SMI Rendezvous, this approach is not an elegant modification. It was introduced as simple change that can be verified easily. More acceptable permanent solutions are proposed in Possibility 1: Modify EDK2 dispatch code, Possibility 3: Add a concurrent driver dispatch module, and Possibility 5: Put SMI handler in additional SMM Guest.

6.2 Scheduling modifications

Some changes must be made to the memory inspection decomposition scheduler as well. This scheduler is located in the EPA-RIMM architecture component called the Back-end Manager (BEM). The BEM is responsible for decomposing inspections of memory regions into partial inspections (such that no partial inspection requires more than the specified maximum time to complete), scheduling the partial inspections, and then triggering partial inspections at the
scheduled time. This is known as the Bin Packing Problem. Calculating the optimal schedule offline is NP-Hard, but can be approximated efficiently.

With the existing version of EPA-RIMM, each bin represents the maximum time that can be spent in one SMI, and each task, representing a partial memory inspection, has a predicted runtime. (Note that the runtimes of SMI handlers are consistent because their execution cannot be interrupted). A bin should be filled with as many tasks as possible without exceeding the maximum time limit.

This must be adapted slightly for the concurrent version of EPA-RIMM. Each SMI now has N bins (the number of available CPUs available on the platform) in which tasks can be scheduled. Like the single-core version, each bin should be filled with as many tasks as possible. No bin can exceed the maximum runtime. Bins are dispatched N-at-a-time. Bins are packed together in a Binset. Ideally, Bins in a Binset are load-balanced. This is variation of the Bin Packing Problem is known as the Number Partition Problem for two bins and the Multi-way Partitioning Problem for N bins. This setup allows the BEM to retain the ability to assign an arbitrary amount of Tasks to a given core.

Besides considering Task size when scheduling, the BEM should also consider the amount of time it takes to perform the cryptographic operations on a Task Description.

Scheduling algorithms should be a subject of future work. Only simple scheduling is considered in this thesis. The size of inspected memory regions will be
even powers of two for the evaluation, so scheduling and load balancing will be trivial.

### 6.3 SMM Inspector modifications

The inspector can get the index of the executing CPU from EDK2. This index will be used to read from the corresponding bin. The bin has all the information required regarding the memory region to inspect. Some performance reporting data structures must be included per-core.

A cyclic barrier function is introduced to provide synchronization between cores. The cores that reach a barrier earlier will spin-wait for the other cores. Once all the cores reach the barrier they are released nearly simultaneously.

### 6.4 Bin protocol modifications

A Task tells the Inspector what to do. A memory measurement Task, for example, specifies a starting address and a length. When the Inspector executes a memory measurement Task, the SMM runtime varies based on the number of bytes inspected.

A Task Description is the data structure that contains all the information necessary to communicate the Task to the Inspector. Task Descriptions are packed into Bins (one Bin per core), which are then packed into a Binset (one Binset per SMI). AES-256 encryption and SHA-256 HMAC are used to ensure privacy, integrity, and authentication of communications. When the Inspector unpacks a Binset, and later packs the results, the SMM runtime varies based on the size (in bytes) of the Task Description data structure and the number of Tasks. The size is fixed at

Section 6.3: SMM Inspector modifications 57
compile time. The number of Tasks sent depends on scheduling, though for this thesis we fix scheduling to one Task per core.

### 6.4.1 Existing protocol

In the existing EPA-RIMM implementation, a Task Description is a struct that contains all the fields necessary to direct the Inspector. Each Task Description is encrypted and HMAC-ed individually. As depicted in Figure 6-7, Task Descriptions for the same SMI are packed into a buffer. The packed Tasks are called a Bin. The simplest Bin has only one Task Description. The BEM is capable of assigning multiple Tasks to a Bin to increase the runtime. See Figure 6-8 for an example of this. Note that in Figure 6-8, the depicted time for a Task includes decryption and HMAC verification, as well as encryption and HMAC creation for the Task Result.

This setup must be modified to allow the BEM to indicate what Bin a Task is assigned to. The difficulty in adding a Bin assignment comes from selecting a method that allows as much work to be done in parallel as possible. Early performance results showed that it was not sufficient for the inspection to be parallelized—the Binset processing, particularly the cryptographic operations, must be parallelized as well.

To enable as much parallel processing in the Inspector as possible, we should reduce inter-core data dependencies. For example, compute time is wasted if one core decrypts Bin assignments while the others wait. It might be better for the BEM to arrange the Binset such that each core already knows where to get its Bin and can
start processing it immediately. Of course, some amount of dependency is usually unavoidable.

![Figure 6-7: Existing Bin memory layout](image)

**Figure 6-7: Existing Bin memory layout**

![Figure 6-8: Existing execution sequence mock-up (not based on real measurements)](image)

**Figure 6-8: Existing execution sequence mock-up (not based on real measurements)**

### 6.4.2 Possibility 1: Add a Binset Header

One solution is to add a Binset Header. Here is an example implementation:
typedef struct {
    UINT8  Ivec[16];
    UINT64 Nonce;
    UINT16 TaskEndIdx[NUM_CORES];
    UINT8  Hmac[32];
} BinsetHeader;

The size of the struct depends on number of cores on the platform. Where \( N \) is the number of cores, the size in bytes is:

\[
56 + 2 \times N
\]

For four cores, the size would be 64 bytes. Note the fields required for security purposes: a random IVec is used to initialize the AES state machine, an HMAC for authenticated integrity, and a nonce to detect replay attacks. The TaskEndIdx field indicates the location of each Bin. This field is shown in this example as a fixed length for simplicity, but a schema can be created to allow the field size to be specified once during runtime initialization.

The memory layout for a Binset, depicted in Figure 6-9, would have the Binset Header first, followed by its Task Descriptions. Figure 6-10 shows an example execution sequence. In this example, Core 0 and Core 2 are assigned one Task, Core 1 is assigned three Tasks, and Core 4 is assigned no Tasks. Core 0 first must copy in the Binset and decrypt the Binset Header before the other cores can know their Task assignments. After the Binset Header is processed, the cores proceed with their Tasks. In this example, a single Task’s execution includes the time to decrypt the Task, perform the inspection, set the results, and encrypt the Task. The depicted variation in Task time is due to the number of the bytes
inspected. The amount of time shown is for visualization purposes only and must not be misconstrued as an empirically measured time ratio.

Figure 6-9: Possibility 1 – Binset memory layout

Figure 6-10: Possibility 1 – Execution sequence mock-up (not based on real measurements)
6.4.3 Possibility 2: Add Bin Headers

One approach is to add Bin Headers. This is the selected approach, so this is the actual prototype implementation:

```c
typedef struct {
    UINT8 Ivec[16];
    UINT64 Nonce;
    UINT32 TaskStartIdx;
    UINT32 TaskEndIdx;
    UINT8 HmacTasks[32];
    UINT8 HmacMe[32];
} BinHeader;
```

The size of the struct is 96 bytes. There is one BinHeader for each core. This struct has IVec, nonce, and HMAC fields like the previous approach. It has an additional HMAC field that allows all the Task Descriptions in the Bin to be HMAC-ed together. The `TaskStartIdx` and `TaskEndIdx` fields give the indexes for the Bin’s Task Descriptions.

The memory layout for a Binset, depicted in **Figure 6-11**, places all the Bin Headers first. This static position allows each core to locate its Bin Header based on its core index. The Bin Headers are followed by the Task Descriptions. The number of Bin Headers can be configured once during runtime initialization.

**Figure 6-12** shows an example execution sequence. In this example, Core 0 and is assigned one Task, Core 1 is assigned three Tasks, Core 2 is assigned two Tasks, and Core 4 is assigned no Tasks. Core 0 first must copy in the Binset. Once it is in SMRAM, each core uses its index to determine the position of its BinHeader. Each core decrypts and reads its BinHeader in parallel. It uses the indexes provided in the BinHeader to locate its Task Descriptions. The core processes its own Tasks.
In this example, the Task Descriptions in a Bin are decrypted and encrypted together. The depicted variation in Task time is due to the number of bytes inspected. The amount of time shown is for visualization purposes only and must not be misconstrued as an empirically measured time ratio.
Section 6.4: Bin protocol modifications

Figure 6-11: Possibility 2 – Binset memory layout

Figure 6-12: Possibility 2 – Execution sequence mock-up (not based on real measurements)
6.4.4 Possibility 3: Add a Binset Header and Bin Headers

One last solution presented is to add a Binset Header and Bin Headers. This is a combination of Possibility 1 and Possibility 2. Here is an example implementation:

```c
typedef struct {
    UINT8 Ivec[16];
    UINT64 Nonce;
    UINT64 NumBins;
    UINT8 Hmac[32];
} BinsetHeader;
```

```c
typedef struct {
    UINT8 Ivec[16];
    UINT64 Nonce;
    UINT32 TaskStartIdx;
    UINT32 TaskEndIdx;
    UINT8 Hmac[32];
} BinHeader;
```

The size of the BinsetHeader struct is 64 bytes. The size of the BinHeader struct is also 64 bytes. Both structs require IVec, nonce, and HMAC fields because they cannot be batched together; the Binset Header must be processed before any Bin Headers can be processed. The benefit of this design is that it allows the number of cores to be specified on a per-Binset basis.

The memory layout for a Binset, depicted in Figure 6-13, places the Binset Header first, followed by the Bin Headers. The number of Bin Headers that follows is specified in the Binset Header. Following the Bin Headers are the Task Descriptions.

Figure 6-14 shows an example execution sequence. Core 0 copies the Binset into SMRAM and decrypts the Binset Header. Once the Bin Header positions are decrypted, the other cores can begin to work in parallel. They decrypt their Bin
Headers to determine the location of their Task Descriptions. Then they process their Tasks. The depicted variation in Task time is due to the number of bytes inspected. The amount of time shown is for visualization purposes only and must not be misconstrued as an empirically measured time ratio.

![Diagram showing memory layout](image)

Figure 6-13: Possibility 3 – Binset memory layout
6.4.5 Selected approach

The approach chosen in this thesis is **Possibility 2: Add Bin Headers**. From Figure 6-12 it can be seen that there is more core independence and compared to the other approaches.
7 Evaluation

7.1 Security analysis

7.1.1 Assumptions

- Root-of-trust has been established on the monitored node via some kind of secure boot process.
- The flash chip on which firmware is stored is protected from unwanted changes.
- An initial provisioning step is performed to collect the correct measurements.
- There is an out-of-band network interface that provides a communication channel that is protected from interference from the host system.

7.1.2 Threat model and attack surface

EPA-RIMM is a framework designed to detect situations where an attacker has gained control of the host system at runtime. This can include the detection of unauthorized changes to the host system's code. In the remainder of this section we discuss attacks on EPA-RIMM itself.

A core assumption is that firmware is secure. This is not necessarily the case. SMM is a high-value target due to its privilege level. Other attacks might aim to control the flash chip, potentially allowing malware to persist in spite of system reinstallation.

While developing firmware code, best practices must be followed in order to protect SMRAM. When writing to buffers outside of SMRAM, an SMI handler should verify that the destination is indeed outside of SMRAM to prevent the handler from being tricked into overwriting SMRAM. When reading from a buffer outside of SMRAM, the buffer should first be copied into SMRAM before being used. This
avoids a Time Of Check to Time Of Use (TOCTOU) attack where a buffer outside SMRAM is maliciously altered via DMA in parallel with SMI handler operations. Copying into SMRAM before any other operations is the recommended mitigation because SMRAM is protected from DMA.

Because EPA-RIMM detects changes, it cannot detect attacks such as Return Oriented Programming (ROP). With ROP, malware doesn't change code; it achieves arbitrary execution using existing code by manipulating the stack.

If there is a Denial of Service (DoS) attack on EPA-RIMM itself, the lack of response from the Inspector would be considered a detection. DoS attacks directed at the host system may be noticeable by the user without a detection mechanism. However, depending on the attack, EPA-RIMM may catch an initial exploit that primes the system for a future attack.

If malware can predict when the next inspection will run or which memory location will be inspected next, it can hide or move its traces to avoid detection. To make prediction harder, inspection could trigger at a random time within a range, the kernel memory location to be inspected could be selected randomly, and the BEM could vary the workload of each Binset so the length of an inspection is harder to predict.

The changes proposed in this thesis allow EPA-RIMM to inspect the kernel in less time. This gives a malware a smaller window of time in which it can operate before detection. This improves the resilience of the system.
7.2 Effectiveness

Previous research tested EPA-RIMM’s ability to detect simulated attacks on the Linux kernel. In Delgado and Karavanic [10], EPA-RIMM detected changes to the Interrupt Descriptor Table Register (IDTR), the Supervisor Mode Execution Protection (SMEP) bit of Control Register 4 (CR4), to the system call table, and a Linux kernel code section. In subsequent research, EPA-RIMM detected simulated attacks on Xen as well.

For this thesis, we have not repeated these tests because the proposed changes do not affect the inspection results. Both the pre-existing EPA-RIMM-Linux implementation and the implementation of this thesis demonstrate the change detection Task by taking a SHA-256 hash. The implementation for EPA-RIMM-Linux runs one Task at a time. The implementation of this thesis can run multiple Tasks in parallel—the Tasks themselves are not sub-divided.

To verify that the results are indeed the same, we inspected the same region of memory with both implementations. The Task size was fixed to 512 bytes and the memory region was sized such that it was decomposed into eight Tasks. The resulting hash values for all Tasks matched between the pre-existing implementation and the implementation of this thesis. These known-good values were stored to allow comparison with newer code in subsequent integration testing.
7.3 Performance model

The EMA-RIMM model for performance measurement [7] partitions the time spent in SMM by the type of work done in each section. \( T_m \) is the observed time spent in SMM. \( T_m \) is broken down such that:

\[
T_m = T_{entry} + T_{work} + T_{exit}.
\]

- \( T_m \) - total time spent in SMM
- \( T_{entry} \) - time spent entering SMM
- \( T_{work} \) - time spent performing the RIMM inspection
- \( T_{exit} \) - time spent exiting SMM

For the results of this thesis, the measurement points are taken as follows. \( T_{entry} \) starts just before the instruction that triggers the SMI and ends at the start of \( T_{work} \). \( T_{work} \) starts just after the main inspector function is invoked and ends just before the main inspector function returns. \( T_{exit} \) starts at the end of \( T_{work} \) and concludes after the RSM instruction completes on the core that issued the SMI.

Chapter 5 and Appendix explain the operations that incur \( T_{entry} \) and \( T_{exit} \).

\( T_{work} \) can be further partitioned into its constituent sub-tasks.

\[
T_{work} = T_{decrypt} + T_{signature_verify} + T_{HMAC_compare} + T_{hash} + T_{HMAC_create} + T_{encrypt} + T_{other}
\]

- \( T_{decrypt}, T_{encrypt} \) - AES decryption and encryption, respectively.
- \( T_{HMAC_create} \) - calculation of HMAC when preparing the Task Result(s)
- \( T_{signature_verify} \) - calculation of HMAC when receiving the Task(s)
- \( T_{HMAC_compare} \) - comparison of calculated HMAC with claimed HMAC
- \( T_{hash} \) - execution of memory inspection
- \( T_{other} \) - everything else, including
  - Binset copying, both in and out
  - Input sanitization

The bin must be copied into SMRAM to prevent a Time-Of-Check to Time-Of-Use (TOCTOU) attack. After the Inspection is completed, the Binset, which now
contains the Task Results, is copied back. The time taken for copying is included in $T_{\text{other}}$.

Although EPA-RIMM is capable of assigning a different number of Tasks to each core, for this evaluation we only consider trivial scheduling. Each Binset will assign one Task per core, and each Task in a Binset will inspect the same amount of memory.

Since this thesis uses multiple cores, the sub-components of $T_{\text{work}}$ can be measured for each core. ($T_{\text{entry}}$ and $T_{\text{exit}}$ can also be measured per core.) With the exception of $T_{\text{other}}$, the sub-components of $T_{\text{work}}$ can be compared straightforwardly, because they represent the same operations (but on different data).

$T_{\text{other}}$ is a catch-all, so it is less straightforward to compare between the BSP and the APs. Binset copying and some input sanitization operations are only performed by the BSP. Further complicating things, two barriers were added, one after the copy-in and one before the copy-out. The first barrier is necessary to ensure that the BSP has finished copying before the APs begin to process their Bins. The second barrier is necessary to ensure the APs have finished processing their Bins before they are copied out. This means that $T_{\text{other}}$ for one core is elongated if it must wait for another core’s execution. This can happen if one core takes a long time to complete $T_{\text{HMAC,Create}}$, for example.

One can also imagine a situation where one core is waiting at the first barrier while another core for some reason experiences a particularly lengthy $T_{\text{entry}}$. This would have the effect of making $T_{\text{work}}$ appear longer than it actually is. However, this
situation is not possible because the start of $T_{work}$ is nearly synchronized across cores. This is due to pre-existing semaphore signaling in EDK2 prior to the call into the inspector main function. Thus, anomalies in $T_{entry}$ for one core will not affect the $T_{other}$ of other cores.

Since the goal is to measure how utilizing additional cores for inspection affects runtime, all we need to know is the maximum $T_{other}$. The start of $T_{work}$ is approximately synchronized for all cores. The BSP does additional work both before the first barrier and after the second barrier. Thus, the maximum $T_{other}$ is guaranteed to be BSP’s $T_{other}$. It follows that the BSP’s $T_{work}$ is the maximum $T_{work}$.

For convenience, the portion of $T_{work}$ that is not $T_{hash}$ can be referred to as $T_{ovd}$. “OVD” stands for overhead.

\[ T_{ovd} = T_{decrypt} + T_{signature_verify} + T_{HMAC\_compare} + T_{HMAC\_Create} + T_{encrypt} + T_{other} \]

By definition:

\[ T_{work} = T_{task} + T_{ovd} \]

### 7.4 Predicted performance improvement

#### 7.4.1 Strong scaling

In Delgado and Karavanic 2018 [10] it is shown that for a 1024-byte inspection, on average $T_m = 256\text{us}$ with $T_{hash} \approx 36\text{us}$. For a 4096-byte inspection, on average $T_m = 359\text{us}$ with $T_{hash} \approx 139\text{us}$. In both cases, there was one Task per SMI.
Amdahl’s Law is used to predict runtime speedup when parallelizing algorithms. Amdahl’s Law is defined as follows:

\[ S(N) = \frac{1}{(1 - P) + \frac{P}{N}} \]

P is the fraction of the operation that can be parallelized, N is the number of parallel instances, and S(N) is the speedup given N.

We will now use Amdahl’s Law to predict the potential speedup of performing Tasks in parallel during an SMI for 1024-byte and 4096-bytes inspection regions. For the 1024-byte region, the ratio of time spent on inspection to time spent in SMM is

\[ P = \frac{T_{\text{hash}}}{T_{m}} = \frac{36\mu s}{256\mu s} \approx .141 \]

If we were to use 4 cores (N = 4), Amdahl’s Law gives a speedup of

\[ S(4) \approx \frac{1}{(1 - .141) + \frac{.141}{4}} \approx 1.12 \]

With N = 64, Amdahl’s Law gives a speedup of

\[ S(64) \approx 1.16 \]

For the 4096-byte region, the parallel ratio is

\[ P = \frac{T_{\text{hash}}}{T_{m}} = \frac{139\mu s}{359\mu s} \approx .387 \]

Amdahl’s Law predicts a speedup of

\[ S(4) \approx 1.40 \]

\[ S(64) \approx 1.62 \]
Amdahl’s Law shows that parallel inspections reach diminishing returns due to the significant proportion of serialized work.

### 7.4.2 Weak scaling

Instead of trying to reduce SMI time, we can aim to increase the amount of work done per SMI. The potential work increase is theoretically limited to the size of the kernel (at which point the entire kernel would be inspected in one SMI).\(^5\)

Gustafson’s Law, which models weak scaling, is defined as

\[
E(N) = 1 - P + NP
\]

P is the fraction of the operation that can be parallelized (when \(N = 1\)), \(N\) is the number of parallel instances, and \(E(N)\) is the efficiency given \(N\). One can see from the formula that \(E(N)\) has \(O(N)\) time complexity. The result \(E(N)\) represents the amount of time it would take for one core to do the work of \(N\) cores.

From the earlier 1024-byte example:

\[
P = \frac{T_{\text{hash}}}{T_m} = \frac{36\mu s}{256\mu s} \approx .141
\]

With the problem expanded by four times, for a total of 4096 bytes to be inspected by four cores in parallel, Gustafson’s Law predicts

\[
E(4) \approx 1 - .141 + 4(.141) \approx 1.42
\]

This means it would take a single core about 1.42 times the original time, \(T_m\), to do the same amount of work as four cores.

\[
1.42 * T_m = 1.42 * 256\mu s = 364\mu s.
\]

\(^5\) Based on the speed of the MinnowBoard processor, inspecting the entire kernel in one SMI without exceeding SMI latency bounds would require thousands of cores working concurrently, though servers are expected to have faster processors.
This is close to the observed single-core $T_m$ for 4096 bytes, which was measured to be 359us.

Unlike Amdahl’s Law, this efficiency predicted by Gustafson’s Law continues to increase linearly with the number of cores, with 64 cores yielding

$$E(64) \approx 9.88$$

for 9.88 times efficiency. For the 4096-byte example

$$p = \frac{T_{hash}}{T_m} = \frac{139 \mu s}{359 \mu s} \approx .387$$

Gustafson's Law predicts

$$E(4) \approx 2.16$$
$$E(64) \approx 25.4$$

7.5 Miscellaneous measurement guidelines

When measuring $T_{exit}$, interrupts must be disabled prior to SMI. Otherwise, any interrupts received in SMM will be handled immediately after RSM. This will affect the timing measurement, making a $T_{exit}$ inaccurately appear longer [43]. $T_{entry}$, $T_{work}$, and $T_{exit}$ are measured once per SMI.

We pin the CPU's C-state to the readiest level before performing a measurement. Higher C-states generally turn off more parts of the CPU. This is a power-saving mechanism. It takes time to transition between C-states. This can affect measurement consistency.

Timing measurements inside SMM were taken by reading the Time Stamp Counter (TSC) register. This is very accurate. Each core has its own TSC and they are
not necessarily synchronized. This is not a problem for our purposes since there is no task switching in SMM.

### 7.6 Performance scaling initial failure

The first attempt of the multicore inspector only parallelized memory inspection. The rest of the work in the SMM Inspector main function was done by a single core. The work included copying the Binset into SMRAM and per-Task cryptographic operations. It decrypted all the Tasks before allowing the other cores to perform the Task in parallel. After the Tasks were complete, the single core encrypted the Tasks and copied the Binset out.

This exhibited poor scaling behavior and showed the need to further parallelize the Binset-processing operations.

The platform was the MinnowBoard Turbot Quad-Core, with Intel Atom E3845 1.91 GHz processor. The OS was Ubuntu 18. A test run was performed for each combination of inspection size (512, 1024, 2048, and 4096 bytes) and number of concurrent cores (1, 2, 3, 4). For those combinations with two or more cores, each core inspected a different memory region (of the same length of bytes) in parallel. Memory was inspected in sections of 512, 1024, 2048, and 4096 bytes per core per SMI. Timing measurements were recorded for $T_{\text{work}}$ and $T_{\text{hash}}$. $T_{\text{work}}$ was recorded for Core 0. $T_{\text{hash}}$ was recorded for all cores. These results were recorded for 1,000 SMIs.

The results are shown in Table 7-1. The values are the average time in microseconds. As expected, $T_{\text{hash}}$ is consistent no matter the number of cores. As the number of bytes doubles, $T_{\text{hash}}$ approximately doubles.
Table 7-1 suggests that the $T_{ovd}$ is a function of the number of cores, but it is actually a function of the number of Task Descriptions (one per core here).

We will isolate some results from the table above for clarity.

<table>
<thead>
<tr>
<th></th>
<th>512 B inspected (per core)</th>
<th>1024 B inspected (per core)</th>
<th>2048 B inspected (per core)</th>
<th>4096 B inspected (per core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-core</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_{hash}$</td>
<td>15.0 us</td>
<td>27.6 us</td>
<td>52.9 us</td>
<td>103.4 us</td>
</tr>
<tr>
<td>$T_{ovd}$</td>
<td>97.1 us</td>
<td>97.2 us</td>
<td>97.4 us</td>
<td>97.2 us</td>
</tr>
<tr>
<td>$T_{work}$</td>
<td><strong>112.1 us</strong></td>
<td><strong>124.8 us</strong></td>
<td><strong>150.3 us</strong></td>
<td><strong>200.6 us</strong></td>
</tr>
<tr>
<td>2-core</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_{hash}$</td>
<td>14.9 us</td>
<td>27.6 us</td>
<td>52.9 us</td>
<td>103.4 us</td>
</tr>
<tr>
<td>$T_{ovd}$</td>
<td>141.3 us</td>
<td>141.2 us</td>
<td>141.1 us</td>
<td>141.4 us</td>
</tr>
<tr>
<td>$T_{work}$</td>
<td><strong>156.2 us</strong></td>
<td><strong>168.8 us</strong></td>
<td><strong>194.0 us</strong></td>
<td><strong>244.8 us</strong></td>
</tr>
<tr>
<td>3-core</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_{hash}$</td>
<td>15.0 us</td>
<td>27.6 us</td>
<td>52.9 us</td>
<td>103.4 us</td>
</tr>
<tr>
<td>$T_{ovd}$</td>
<td>201.7 us</td>
<td>201.9 us</td>
<td>200.6 us</td>
<td>201.3 us</td>
</tr>
<tr>
<td>$T_{work}$</td>
<td><strong>216.7 us</strong></td>
<td><strong>229.5 us</strong></td>
<td><strong>253.5 us</strong></td>
<td><strong>304.7 us</strong></td>
</tr>
<tr>
<td>4-core</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_{hash}$</td>
<td>15.1 us</td>
<td>27.7 us</td>
<td>53.0 us</td>
<td>103.5 us</td>
</tr>
<tr>
<td>$T_{ovd}$</td>
<td>270.5 us</td>
<td>270.5 us</td>
<td>268.8 us</td>
<td>268.5 us</td>
</tr>
<tr>
<td>$T_{work}$</td>
<td><strong>285.6 us</strong></td>
<td><strong>298.2 us</strong></td>
<td><strong>321.8 us</strong></td>
<td><strong>372.0 us</strong></td>
</tr>
</tbody>
</table>

Table 7-1: Measurements with parallelized inspection only

For 1024 bytes, $T_{work}$ for 1-core is 124us, 2-core is 169us, 3-core is 230us, and 4-core is 298us. Although there are very few data points, this appears to be super-linear scaling. For 4096 bytes, 1-core is 200us. Therefore it is faster for 1 core to inspect 4096 bytes than for 4 cores.

$T_{ovd}$ is a significant cost, even exceeding $T_{task}$ when inspecting small memory regions. These results indicate that it is not possible to achieve linear speedup without parallelizing the sub-operations of $T_{ovd}$.
7.7 Issues with multicore HMAC in OpenSSL

The HMAC workflow has three components: init, update, and final. Init is done to reset the object before use. Update does most of the HMAC algorithm, digesting the input bytes. Final finishes the HMAC and makes the result available.

Concurrently running HMAC init or final in EDK2 can cause memory corruption or crash the system. For my implementation I serialized access to these functions with a spinlock. I also collected data with HMAC disabled.

Looking into the OpenSSL code, it appears that HMAC subroutines end up calling malloc and free. However, this cannot be the whole explanation, because SHA-256 also ends up calling malloc and free, yet has no observed corruption or crashing. With multicore code, the absence of adverse effects does not prove that the code is correct. For example, since code in SMM runs at a consistent speed, timing alone could prevent a race condition from causing any problems. However with amount of tests we have run, we would have expected to see some adverse effects from the other cryptographic operations if they had issues.

Regardless of its status as a widely used security library, OpenSSL still does not have great support for multithreading [32]. How EDK2 integrates OpenSSL into an environment where threads do not exist is presently unclear to us. It is known that OpenSSL can be setup to use a custom allocator function other than malloc. Likewise, some of the locking primitives presumably must be implemented without OS calls.
7.8 Performance scaling success

After modifying the inspector to perform the AES encryption/decryption in parallel, along with the other modifications discussed in Sections 6.3 and 6.4, EPA-RIMM was able to improve performance by adding cores. Note that HMAC could not be parallelized. This is discussed in Section 7.7. HMAC init and HMAC final were locked, resulting in some amount of serialization.

The platform was the MinnowBoard Turbot Quad-Core, with Intel Atom E3845 1.91 GHz processor. The OS was Ubuntu 18. A test run was performed for each combination of inspection size (512, 1024, 2048, 4096, 8196) bytes by number of concurrent cores (1, 2, 3, 4). A test run for 16384 bytes by (1, 2) cores was included. Additional test runs for 1 core only with (1536, 3072, 6144, 12288, 24576, 32768) were included. Time measurements were recorded for $T_{work}$ only. Time measurements for $T_{work}$ were recorded for all cores. The results were recorded for 500 SMIs.

The results are shown in Table 7-2, Table 7-3, Table 7-4, and Table 7-5. Note that the standard deviation for $T_{work}$ is significant when HMAC is performed. The unit of time is microseconds. The unit of bytes is KiB. One KiB is 1024 bytes. $T_{ovd}$ was calculated by subtracting $T_{hash}$ from $T_{work}$. 
Section 7.8: Performance scaling success

### Table 7-2: Average inspection times with parallelized inspection and encryption/decryption, and partially parallelized HMAC

<table>
<thead>
<tr>
<th></th>
<th>0.5 KiB inspected (per core)</th>
<th>1 KiB inspected (per core)</th>
<th>2 KiB inspected (per core)</th>
<th>4 KiB inspected (per core)</th>
<th>8 KiB inspected (per core)</th>
<th>16 KiB inspected (per core)</th>
<th>32 KiB inspected (per core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-core</td>
<td>avg. $T_{hash}$ 15.1 us</td>
<td>27.7 us</td>
<td>53.0 us</td>
<td>103.5 us</td>
<td>204.6 us</td>
<td>407.0 us</td>
<td>811.6 us</td>
</tr>
<tr>
<td></td>
<td>$T_{eval}$ 253 us</td>
<td>248 us</td>
<td>242 us</td>
<td>242 us</td>
<td>242 us</td>
<td>240 us</td>
<td>241 us</td>
</tr>
<tr>
<td></td>
<td>= avg. $T_{work}$ 268 us</td>
<td>276 us</td>
<td>295 us</td>
<td>345 us</td>
<td>447 us</td>
<td>647 us</td>
<td>1053 us</td>
</tr>
<tr>
<td>2-core</td>
<td>avg. $T_{hash}$ 15.1 us</td>
<td>27.7 us</td>
<td>52.9 us</td>
<td>103.4 us</td>
<td>204.6 us</td>
<td>406.9 us</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$T_{eval}$ 288 us</td>
<td>281 us</td>
<td>276 us</td>
<td>275 us</td>
<td>275 us</td>
<td>274 us</td>
<td></td>
</tr>
<tr>
<td></td>
<td>= avg. $T_{work}$ 303 us</td>
<td>309 us</td>
<td>329 us</td>
<td>379 us</td>
<td>480 us</td>
<td>681 us</td>
<td></td>
</tr>
<tr>
<td>3-core</td>
<td>avg. $T_{hash}$ 15.1 us</td>
<td>27.8 us</td>
<td>53 us</td>
<td>103.5 us</td>
<td>204.6 us</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$T_{eval}$ 325 us</td>
<td>313 us</td>
<td>300 us</td>
<td>297 us</td>
<td>297 us</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>= avg. $T_{work}$ 340 us</td>
<td>341 us</td>
<td>353 us</td>
<td>400 us</td>
<td>502 us</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-core</td>
<td>avg. $T_{hash}$ 15.3 us</td>
<td>27.8 us</td>
<td>53.0 us</td>
<td>103.5 us</td>
<td>204.7 us</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$T_{eval}$ 373 us</td>
<td>356 us</td>
<td>335 us</td>
<td>333 us</td>
<td>332 us</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>= avg. $T_{work}$ 389 us</td>
<td>384 us</td>
<td>388 us</td>
<td>436 us</td>
<td>537 us</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 7-3: Standard deviation of Table 7-2

<table>
<thead>
<tr>
<th></th>
<th>0.5 KiB inspected (per core)</th>
<th>1 KiB inspected (per core)</th>
<th>2 KiB inspected (per core)</th>
<th>4 KiB inspected (per core)</th>
<th>8 KiB inspected (per core)</th>
<th>16 KiB inspected (per core)</th>
<th>32 KiB inspected (per core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-core</td>
<td>std. dev. $T_{hash}$ 0.51 us</td>
<td>0.45 us</td>
<td>0.51 us</td>
<td>0.35 us</td>
<td>0.35 us</td>
<td>0.40 us</td>
<td>0.34 us</td>
</tr>
<tr>
<td></td>
<td>std. dev. $T_{work}$ 15.47 us</td>
<td>18.16 us</td>
<td>25.91 us</td>
<td>28.81 us</td>
<td>28.80 us</td>
<td>31.22 us</td>
<td>29.94 us</td>
</tr>
<tr>
<td>2-core</td>
<td>std. dev. $T_{hash}$ 0.47 us</td>
<td>0.44 us</td>
<td>0.42 us</td>
<td>0.30 us</td>
<td>0.40 us</td>
<td>0.27 us</td>
<td></td>
</tr>
<tr>
<td></td>
<td>std. dev. $T_{work}$ 14.54 us</td>
<td>14.56 us</td>
<td>15.48 us</td>
<td>14.76 us</td>
<td>15.82 us</td>
<td>15.90 us</td>
<td></td>
</tr>
<tr>
<td>3-core</td>
<td>std. dev. $T_{hash}$ 0.56 us</td>
<td>0.55 us</td>
<td>0.45 us</td>
<td>0.31 us</td>
<td>0.36 us</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>std. dev. $T_{work}$ 15.55 us</td>
<td>15.42 us</td>
<td>19.59 us</td>
<td>21.52 us</td>
<td>21.65 us</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-core</td>
<td>std. dev. $T_{hash}$ 0.74 us</td>
<td>0.61 us</td>
<td>0.54 us</td>
<td>0.40 us</td>
<td>0.45 us</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>std. dev. $T_{work}$ 17.32 us</td>
<td>18.37 us</td>
<td>24.99 us</td>
<td>30.01 us</td>
<td>27.89 us</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 7-4: Additional average inspection times with parallelized inspection and encryption/decryption, and partially parallelized HMAC.

<table>
<thead>
<tr>
<th>1-core</th>
<th>1.5 KiB inspected (per core)</th>
<th>3 KiB inspected (per core)</th>
<th>6 KiB inspected (per core)</th>
<th>12 KiB inspected (per core)</th>
<th>24 KiB inspected (per core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg. $T_{hash}$</td>
<td>40.4 us</td>
<td>78.4 us</td>
<td>154.2 us</td>
<td>305.8 us</td>
<td>609.3 us</td>
</tr>
<tr>
<td>+ $T_{ovd}$</td>
<td>245 us</td>
<td>241 us</td>
<td>243 us</td>
<td>240 us</td>
<td>241 us</td>
</tr>
<tr>
<td>= avg. $T_{work}$</td>
<td>285 us</td>
<td>320 us</td>
<td>397 us</td>
<td>546 us</td>
<td>851 us</td>
</tr>
</tbody>
</table>

### Table 7-5: Standard deviation of Table 7-4

<table>
<thead>
<tr>
<th>1-core</th>
<th>1.5 KiB inspected (per core)</th>
<th>3 KiB inspected (per core)</th>
<th>6 KiB inspected (per core)</th>
<th>12 KiB inspected (per core)</th>
<th>24 KiB inspected (per core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>std. dev. $T_{hash}$</td>
<td>0.61 us</td>
<td>0.54 us</td>
<td>0.54 us</td>
<td>0.45 us</td>
<td>0.45 us</td>
</tr>
<tr>
<td>std. dev. $T_{work}$</td>
<td>22.57 us</td>
<td>27.97 us</td>
<td>28.50 us</td>
<td>28.96 us</td>
<td>29.30 us</td>
</tr>
</tbody>
</table>

$T_{hash}$ is always very consistent. It approximately doubles when the number of bytes to be measured doubles. $T_{ovd}$ increases with the number of CPUs. $T_{ovd}$ is relatively consistent for each amount of CPUs. This is graphed in Figure 7-1. The graph includes error bars with the standard deviation. For 4-cores, 2 KiB and 4 KiB measurements are slightly longer. This only occurred when HMAC was performed; the anomaly was not measured in the run with HMAC disabled.
Performance scaling with added cores is shown in **Figure 7-2. Figure 7-3** shows the same graph with a closer look at the area that has the overlapping lines. This figure shows that multicore inspections are slower than single-core for small inspection sizes. Multicore inspections start to see performance gains over single-core at 2048 inspected bytes per core. That comes out to 4096, 6144, and 8192 total bytes inspected for 2-core, 3-core, and 4-core inspections, respectively. 2-core is faster than 3-core and 4-core at 4096 total bytes. 3-core takes over in the 6 KiB to
12 KiB range. 4-core is the fastest starting at 16 KiB total bytes (4096 per core) or more.

Figure 7-2: Performance scaling as total inspected bytes increases
Figure 7-3: Zoomed-in version of Figure 7-2: Performance scaling as total inspected bytes increases showing the crossover points

Multicore UEFI EPA-RIMM starts to run faster than Delgado’s single-core EPA-RIMM-Linux inspector at about 16 KiB total bytes inspected. EPA-RIMM-Linux appears to have much less HMAC overhead. When run with HMAC disabled, even the 1-core variant of the multicore-capable UEFI inspector outperforms EPA-RIMM-Linux. This is probably due to the differences in Task Description and the Quad-core MinnowBoard having a slightly faster CPU than the Dual-core MinnowBoard.
7.9 Performance impacts of HMAC

Based on the previous performance results, we further explore the impact of HMAC. In this section, we show results with the HMAC operations deactivated. This is not secure according to the EPA-RIMM specification. We measure the parallelized Task and encryption/decryption with HMAC disabled. The results were taken for 1,000 SMIs. Otherwise, the test setup was the same as Section 7.8.

<table>
<thead>
<tr>
<th></th>
<th>0.5 KiB Inspected (per core)</th>
<th>1 KiB Inspected (per core)</th>
<th>2 KiB Inspected (per core)</th>
<th>4 KiB Inspected (per core)</th>
<th>8 KiB Inspected (per core)</th>
<th>16 KiB Inspected (per core)</th>
<th>32 KiB Inspected (per core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-core</td>
<td>avg. $T_{hash}$</td>
<td>16.5</td>
<td>29.1</td>
<td>54.4</td>
<td>105.0</td>
<td>206.1</td>
<td>408.4</td>
</tr>
<tr>
<td></td>
<td>+ $T_{ovld}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>= avg. $T_{work}$</td>
<td>54.6</td>
<td>67.3</td>
<td>92.5</td>
<td>143.0</td>
<td>244.5</td>
<td>446.5</td>
</tr>
<tr>
<td>2-core</td>
<td>avg. $T_{hash}$</td>
<td>16.2</td>
<td>28.9</td>
<td>54.2</td>
<td>104.7</td>
<td>205.8</td>
<td>408.2</td>
</tr>
<tr>
<td></td>
<td>+ $T_{ovld}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>= avg. $T_{work}$</td>
<td>59.6</td>
<td>72.9</td>
<td>98.2</td>
<td>148.8</td>
<td>249.7</td>
<td>452.8</td>
</tr>
<tr>
<td>3-core</td>
<td>avg. $T_{hash}$</td>
<td>16.2</td>
<td>28.9</td>
<td>54.2</td>
<td>104.7</td>
<td>205.9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+ $T_{ovld}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>= avg. $T_{work}$</td>
<td>66.3</td>
<td>79.7</td>
<td>104.8</td>
<td>155.4</td>
<td>256.6</td>
<td></td>
</tr>
<tr>
<td>4-core</td>
<td>avg. $T_{hash}$</td>
<td>16.2</td>
<td>28.9</td>
<td>54.1</td>
<td>104.7</td>
<td>205.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+ $T_{ovld}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>= avg. $T_{work}$</td>
<td>74.6</td>
<td>87.1</td>
<td>112.3</td>
<td>163.1</td>
<td>264.6</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.6: Average inspection times with parallelized inspection and encryption/decryption, and HMAC disabled
### Table 7-7: Standard deviation of Table 7-6

<table>
<thead>
<tr>
<th></th>
<th>0.5 KiB inspected (per core)</th>
<th>1 KiB inspected (per core)</th>
<th>2 KiB inspected (per core)</th>
<th>4 KiB inspected (per core)</th>
<th>8 KiB inspected (per core)</th>
<th>16 KiB inspected (per core)</th>
<th>32 KiB inspected (per core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-core</td>
<td>std. dev. $T_{hash}$</td>
<td>0.27 us</td>
<td>0.24 us</td>
<td>0.30 us</td>
<td>0.28 us</td>
<td>0.30 us</td>
<td>0.33 us</td>
</tr>
<tr>
<td></td>
<td>std. dev. $T_{work}$</td>
<td>1.98 us</td>
<td>2.03 us</td>
<td>2.15 us</td>
<td>2.11 us</td>
<td>2.16 us</td>
<td>2.27 us</td>
</tr>
<tr>
<td>2-core</td>
<td>std. dev. $T_{hash}$</td>
<td>0.94 us</td>
<td>1.06 us</td>
<td>1.02 us</td>
<td>1.03 us</td>
<td>0.95 us</td>
<td>0.94 us</td>
</tr>
<tr>
<td></td>
<td>std. dev. $T_{work}$</td>
<td>2.10 us</td>
<td>2.41 us</td>
<td>2.28 us</td>
<td>2.34 us</td>
<td>2.27 us</td>
<td>2.48 us</td>
</tr>
<tr>
<td>3-core</td>
<td>std. dev. $T_{hash}$</td>
<td>0.63 us</td>
<td>0.61 us</td>
<td>0.67 us</td>
<td>0.67 us</td>
<td>0.67 us</td>
<td>0.68 us</td>
</tr>
<tr>
<td></td>
<td>std. dev. $T_{work}$</td>
<td>2.84 us</td>
<td>2.95 us</td>
<td>3.12 us</td>
<td>2.91 us</td>
<td>2.91 us</td>
<td>2.89 us</td>
</tr>
<tr>
<td>4-core</td>
<td>std. dev. $T_{hash}$</td>
<td>0.99 us</td>
<td>0.98 us</td>
<td>1.01 us</td>
<td>0.98 us</td>
<td>0.94 us</td>
<td>0.94 us</td>
</tr>
<tr>
<td></td>
<td>std. dev. $T_{work}$</td>
<td>3.85 us</td>
<td>3.77 us</td>
<td>3.80 us</td>
<td>3.85 us</td>
<td>3.85 us</td>
<td>3.85 us</td>
</tr>
</tbody>
</table>

### Table 7-8: Additional average inspection times with parallelized inspection and encryption/decryption, and HMAC disabled

<table>
<thead>
<tr>
<th></th>
<th>1.5 KiB inspected (per core)</th>
<th>3 KiB inspected (per core)</th>
<th>6 KiB inspected (per core)</th>
<th>12 KiB inspected (per core)</th>
<th>24 KiB inspected (per core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-core</td>
<td>avg. $T_{hash}$</td>
<td>41.8 us</td>
<td>79.7 us</td>
<td>155.6 us</td>
<td>307.2 us</td>
</tr>
<tr>
<td></td>
<td>$T_{ovd}$</td>
<td>38 us</td>
<td>39 us</td>
<td>38 us</td>
<td>39 us</td>
</tr>
<tr>
<td></td>
<td>$T_{work}$</td>
<td>80 us</td>
<td>118 us</td>
<td>194 us</td>
<td>346 us</td>
</tr>
</tbody>
</table>

### Table 7-9: Standard deviation of Table 7-8

<table>
<thead>
<tr>
<th></th>
<th>1.5 KiB inspected (per core)</th>
<th>3 KiB inspected (per core)</th>
<th>6 KiB inspected (per core)</th>
<th>12 KiB inspected (per core)</th>
<th>24 KiB inspected (per core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-core</td>
<td>std. dev. $T_{hash}$</td>
<td>0.27 us</td>
<td>0.26 us</td>
<td>0.34 us</td>
<td>0.34 us</td>
</tr>
<tr>
<td></td>
<td>std. dev. $T_{work}$</td>
<td>2.15 us</td>
<td>2.16 us</td>
<td>2.11 us</td>
<td>2.10 us</td>
</tr>
</tbody>
</table>

Section 7.9: Performance impacts of HMAC
These results show significant performance improvement and scaling behavior, especially with smaller amounts of total bytes inspected. This indicates that HMAC is a significant contributor to overhead. This is due to both the cost of executing HMAC-SHA-256 (which performs two SHA-256 operations) and the consequence of locking the Init and Final sub-operations of HMAC.

Figure 7-5 and Figure 7-6 compare the scaling with HMAC disabled and HMAC enabled (and partially parallelized), respectively. This shows that 2-core
inspections show improved performance at 1024 total bytes inspected (512 bytes per core) or more when HMAC is disabled and at 4096 total bytes (2048 per core) or more when HMAC is enabled.

4-core inspections show performance improvements at 4096 total bytes inspected (1024 per core) or more when HMAC is disabled and at 16,384 total bytes (4096 per core) or more when HMAC is enabled. 4-core inspections have a negative impact on performance at smaller inspection sizes.
Figure 7-5: Effect on $T_{\text{work}}$ as number of cores increases when HMAC is disabled
Figure 7-6: Effect on $T_{work}$ as number of cores increases when HMAC is enabled (partially parallelized)
7.10 Discussion

Overall, these measurements show that performance can be improved by leveraging multiple cores. It is very likely that even better scaling can be achieved by getting multicore HMAC to work or by switching to an equivalent MAC alternative. Further simplifying the Binset protocol and limiting BEM Task scheduling could reduce the communication overhead required to securely process Task Descriptions in the SMM Inspector. However, this would affect the ability to increase SMM time by assigning additional Tasks to a core, which is fundamental concept for EPA-RIMM.
8 Conclusions and Future Work

8.1 Conclusions

- It is not sufficient to parallelize only the Task portion of the inspection because the increase of overhead dominates.

- Parallelizing the Task, the encryption/decryption, and part of the HMAC shows good scaling above a certain size Task. However, at smaller sizes the overhead dominates.

- HMAC is a factor hindering scaling, especially at smaller Task sizes.

- These results suggest that RIMMs running in SMM can improve performance and resilience by adding concurrency to both the integrity measurements and communication-related cryptographic operations on the monitored node.

8.2 Future work

8.2.1 Per-function inspection scheduling

So far we have only looked at task schedules where each core is scheduled the same amount of work—i.e. each core is assigned one task, and each task has the same number of bytes. This is useful for providing a controlled performance comparison, but tasks might not always be scheduled in this way.

One problem with provisioning the kernel in homogenously sized chunks is that it can be difficult to tell what code has been changed. If malicious code has changed a function, the inspector can only tell that there is a change somewhere in the chunk of memory in which the function resides.
Another option is to provision the kernel per-function. Each function is mapped in the `kallsyms` file and can be provisioned with the expected hash value. One advantage of this is that it gives a more granular result and may indicate more information about the kind of attack. Additionally, some functions may be more attractive targets than others, due to vulnerabilities of the function or sensitive operations the function performs.

### 8.2.2 Binset protocol

A significant portion of time spent in $T_{\text{work}}$ is spent processing Task Descriptions. Each additional Task adds overhead. EPA-RIMM cannot spend too long in SMM. It may be the case that there is no situation wherein assigning a second Task to any core is both a performance improvement and does not exceed the 1.5ms empirical limit on time spent in SMM. This needs to be studied. If it is true, then the Binset protocol can be simplified, as arbitrary Task assignment will no longer be a requirement. Since the cryptographic operations are expensive, a simpler Binset protocol could execute faster if it does not require the overhead of securely packing and unpacking Bin Headers or Binset Headers.

### 8.2.3 Caching effects

We wish to examine how utilizing multiple cores in SMM affects cache misses once they return from SMM. If a core is only spin-waiting, it will not be overwriting its cache. When it is put to work in SMM, the items in the cache from userspace or kernelspace will be replaced with whatever the core is working on in SMM. When it
returns from SMM, it may have more cache misses than it would have if it had not
done any work in SMM.

It would also be interesting to look into effects of cache sharing inside SMM.
Since multiple cores might be utilizing the same cache, this could affect performance
negatively.

8.2.4 Granular measurements of SMI entry and exit

Part of the work done for this thesis included mapping out the EDK2
architecture of SMI enter and exit. Besides the microcode operations of SMI
interrupts and RSM, there is a lot of C code for locating and executing various SMI
handlers. We could further break down $T_{\text{entry}}$ and $T_{\text{exit}}$ as we do for $T_{\text{work}}$. 
References


References


[34] Joanna Rutkowska. 2015. Intel x86 considered harmful.


[44] Shengye Wan, Jianhua Sun, Kun Sun, Ning Zhang, and Qi Li. 2019. SATIN: A Secure and Trustworthy Asynchronous Introspection on Multi-Core ARM Processors. In 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE, Portland, OR, USA, 289–301. DOI:https://doi.org/10.1109/DSN.2019.00040


https://firmware.intel.com/sites/default/files/A_Tour_Beyond_BIOS_Creating_the_Intel_Firmware_Support_Package_with_the_EFI_Developer_Kit_II_%28FSP2.0%29.pdf


Appendix – Details of EDK2 SMM Execution Flow

This appendix details the function call tree and file mapping for SMI handling. This would be useful for anyone who wants to modify the handling.

**EDK2 SMM execution flow (no STM)**

Upon receiving an SMI, the core performs the steps described in 5.1. The entrypoint is function `_SmiHandler`. Filenames shown are from the edk2 repository [62].

```
_SmiHandler (PiSmmCpuDxeSmm\X64\SmiEntry.asm)
    - CPUSmmDebugEntry
    - call SmiRendezvous
    |-> SmiRendezvous (PiSmmCpuDxeSmm\MpService.c)
        - SmmCpuFeaturesRendezvousEntry
        - 1 "Boot Strap Processor" (BSF) does some work, then enters into the SMM Core
        - Other cores "Application Processors" (APs) do some work work, then spin-wait
        - call gSmmCpuPrivate->SmmCoreEntry (which is function pointer)
    |-> SmmEntryPoint (MdeModulePkg\Core\PiSmmCore\PiSmmCore.c)
        - PlatformHookBeforeSmmDispatch
        - SmmManage
        |-> SmiManage (MdeModulePkg\Core\PiSmmCore\Smi.c)
            - searches a linked list for the handler
            - call SmiHandler->Handler (which is function pointer)
            - This is not the actual handler yet
        |-> PchSmmCoreDispatcher (CLOSED SOURCE pchsmiddispatcher\smm\PchSmmCore.c)
            - which calls PchSmiTypeCallbackDispatcher
            - for an example, see edk2-platforms\Silicon\Intel\KabylakeSiliconPkg\Pch\PchSmmDispatcher\Smm\PchSmmCore.c
            - Actual SMM Handler
                |-> SmmCpuFeaturesRendezvousExit
                | - CPUSmmDebugExit
                | - RSM

**Steps for identifying PchSmmCoreDispatcher as closed-source**

1. Function SmiManage (MdeModulePkg\Core\PiSmmCore\Smi.c) calls SmiHandler->Handler
2. SmiHandler is type SMI_HANDLER*
```
3. SMI_HANDLER is defined in MdeModulePkg\Core\PiSmmCore\PiSmmCore.h

```c
typedef struct {
    UINTN Signature;
    LIST_ENTRY Link; // Link on SMI_ENTRY.SmiHandlers
    EFI_SMM_HANDLER_ENTRY_POINT2 Handler; // The smm handler's entry
    UINTN CallerAddr; // The address of caller who
    SMI_ENTRY *SmiEntry;
    VOID *Context; // for profile
    UINTN ContextSize; // for profile
} SMI_HANDLER;
```

4. As you can see, the Handler field is type EFI_SMM_HANDLER_ENTRY_POINT2

5. EFI_SMM_HANDLER_ENTRY_POINT2 is defined in MdeModulePkg\Core\PiSmmCore\Smi.c

```c
/**
 * Main entry point for an SMM handler dispatch or communicate-based callback.
 *
 * @param[in]     DispatchHandle  The unique handle assigned to this handler by
 * @param[in]     Context         Points to an optional handler context which was
 * @param[in,out] CommBuffer      A pointer to a collection of data in memory that will
 * @param[in,out] CommBufferSize  The size of the CommBuffer.
 *
 * @retval EFI_SUCCESS            The interrupt was handled and quiesced. No other
 * @retval EFI_WARN_INTERRUPT_SOURCE_QUIESCED  The interrupt has been quiesced but other
 * @retval EFI_WARN_INTERRUPT_SOURCE_PENDING   The interrupt is still pending and other
 * @retval EFI_INTERRUPT_PENDING               The interrupt could not be quiesced.
 */

typedef EFI_STATUS
(EFIAPI *EFI_SMM_HANDLER_ENTRY_POINT2) (  
    IN EFI_HANDLE  DispatchHandle,  
    IN CONST VOID  *Context OPTIONAL,  
    IN OUT VOID    *CommBuffer OPTIONAL,  
    IN OUT UINTN   *CommBufferSize OPTIONAL 
);  
```

6. So an EFI_SMM_HANDLER_ENTRY_POINT2 is a signature for a function pointer that takes four arguments.

7. Back to SmiManage, the call looks like this:

```c
Status = SmiHandler->Handler (  
    (EFI_HANDLE) SmiHandler,  
    Context,  
    CommBuffer,  
    CommBufferSize 
);  
```

8. However, the registered EPA-RIMM driver has this signature:

```c
VOID  
EFIAPI  
InspectorMain (  
    IN EFI_HANDLE DispatchHandle,  
    IN EFI_SMM_SW_DISPATCH_CONTEXT  *DispatchContext)  
);  
```

9. Therefore there must be another function before the EPA-RIMM handler

10. I could see the address of SmiHandler->Handler via debug print.
    (0x7A6B8FE0 in my build)

Looking through the boot log (access via serial cable), I found the closest SMM driver:

```c
Loading SMM driver at 0x0007A6B7000 EntryPoint=0x0007A6B8240 PchSmiDispatcher.efi  
```

11. Unfortunately, this is one of the closed-source modules
12. The closed source modules do come with some debug .pdb’s. Using DBH, a tool for debugging edk2 binaries, I loaded `silicon\Vlv2BinaryPkg\x64\PchSmiDispatcher.pdb` with base address 0x007A6B7000 and address 0x7A6B8FE0.

```
> base 0x007A6B7000
> laddr 7A6B8FE0
> file : c:\minnowboard2closesource\vlv2devicecodepkg\valleyview2soc\southcluster\pchsmidispenser\asm\pchsmmcore.c
> line : 636
```

13. This file is not included in the public release for minnowboard. However, a different implementation of the module is open released for Kabylake and a similar module for Quark.

**EDK2 SMM execution flow (with STM)**

Upon receiving an SMI, the core jumps to a special VM Exit handler (not the typical VM Exit handler). Filenames shown are from the STM reference implementation [48].

```
AsmHostEntrypointSmi (VMExit) (Core\VmExit.asm)
- performs context switch to STM
- (Hypervisor is a guest in the eyes of STM)
- Then calls StmHandlerSmi

StmHandlerSmi (Core\Runtime\SmiHandler.c)
- Reads VMExit reason
- If reason is SMI, call SmiEventHandler

SmiEventHandler (Core\Runtime\SmiEventHandler.c)
- Swap SMI VMCS for SMM VMCS
- VMResume to SMM Guest
  ---VMResume--- _SmiHandler
  Flow continues as in normal Non-STM handling until RSM.
  ...

<---VMExit--- RSM
AsmHostEntrypointSmm (Core\Runtime\x64\VmExit.asm)
- Note: different from AsmHostEntrypointSmi, but is a counterpart
- performs context switch to STM
- Then calls StmHandlerSmm

StmHandlerSmm (Core\Runtime\SmmHandler.c)
- Reads VMExit reason
- If reason is RSM, call RsmHandler

RsmHandler (Core\Runtime\SmmRsmHandler.c)
- Swap SMM VMCS for SMI VMCS
- optional core rendezvous
<--VMResume-->
```