On Using the System Management Mode for Security Purposes

William Augusto Rodrigues de Souza

Thesis submitted to the University of London
for the degree of Doctor of Philosophy

2016

On Using the System Management Mode for Security Purposes

Department of Mathematics
Royal Holloway, University of London

Seeing much, suffering much and studying much are the three pillars of learning.

(Benjamin Disraeli)

Declaration of Authorship

I, William Augusto Rodrigues de Souza, hereby declare that this thesis and the work presented in it is entirely my own. Where I have consulted the work of others, this is always clearly stated.

Signed:

(William Augusto Rodrigues de Souza)

Date:

Summary

Computer systems are by design insecure and therefore are many security issues around them. So, security practitioners are always trying to enhance security and performing verification tasks to minimise the risk of potential threats become successful attacks. These tasks are usually performed by security tools.

Thus concepts as: isolation, privilege and view are important in the context of computer systems. Security tools must have good isolation, privilege and view of the system. Then, security tools must operate isolated, have high privilege and must have a global view of the system, but also good ability to view and act timely in its own environment to enhance the chances of success when performing their tasks and for not being hit by the problems they are trying to solve.

In this context, this research investigates the System Management Mode (SMM) in the context of Intel processors, current security tools capitalising on SMM and attacks and misuses of SMM to establish a set of requirements and then design a generic architecture for SMM-based security tools. That generic architecture is tested by building a proof of concept to measure the integrity of a file of the Xen hypervisor. This measurement is limited to the minimum necessary to prove the concept of the architecture.

The problem context addressed is a cloud computing environment, comprising of one or more machines (chipsets). Each chipset hosts in its main memory (DRAM) a virtualised environment comprising of one manager virtual machine, one or more guest virtual machines and a hypervisor. We address our research investigation in two levels: the vertical and the horizontal security level. The vertical security level puts the problem in context, relating it to security issues on: cloud, chipset, memory, virtualisation layer and cache memory. The horizontal security level considers the research problem in its environment, relating it to security issues on components of the bootup process and the processor, such as: Intel VMX, TXT and SGX, BIOS and so on.

First, we investigate the SMM, its resources and components. Then, we analyse SMM-based security tools and the opportunities to improve them. We also analyse SMM attacks and how to thwart them. From the acquired knowledge, we establish a set of requirements to use SMM for security purposes. Having the requirements, we design a generic architecture for SMM-based security tools. To test the architecture, we build a proof of concept comprising of a module to probe chipsets and a SMM-based hypervisor integrity measurement tool.

The implementation of that architecture was done in a proof of concept designed to have two modules: a manager and an agent. The manager module is used for learning about and researching on the target machine, as for probing, setting and clearing registers related to SMM. The manager can be used in the target

machine or in a machine with the same chipset of the target machine. So, it can be deployed in main memory. The agent basically comprises of two parts: a basic code embodying management functions and a payload, where the security functions are implemented. So the use of a payload is what makes the architecture generic since any security task might be implemented and added in the agent by changing the payload.

We conclude that any security tool can capitalising on SMM resources provided that it meets the set of requirements established in this research: small, fast, persistent, cooperative, isolated, resistant, complete and SMI-independent (meaning that it can be started by any System management interruption, which occur in the chipset); and stick to the proposed generic architecture.

Acknowledgments

My heartfelt thanks to my supervisor Dr. Allan Tomlinson for his guidance, encouragement, patience and dedication.
I would like to also thank my examiners Professor William Buchanan and Professor Lorenzo Caballaro for taking the time to examine my thesis.
I gratefully acknowledge all professors, specially Professors Chris Mitchell and Carlos Cid, colleagues and staff at ISG and at the Department of Mathematics for making my life easier and for all support, guidance and fruitful discussions.
I would like to also thank all support and encouragement I received from The Centre for Naval System Analysis of Brazilian Navy.
I am also so grateful for all support and understanding received from my family.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Limits and Scope 3
1.3 Significance 8
1.4 Research Questions 9
1.5 Contribution 11
1.6 List of Publications 12
1.7 Overview of the Research 13

2 Background **15**
2.1 Introduction 15
2.2 Definitions 17
2.3 Context and Technologies 19
2.4 Environment and Technologies 35
2.5 Data Integrity with Hash Functions 39
2.6 Related work: System Executive Software Integrity Issues 40
2.7 Discussion 43
2.8 Summary 44

3 The System Management Mode (SMM) **45**
3.1 Introduction 45
3.2 Components 46
3.3 SMM operation and relations 54
3.4 Security implementations using SMM 58
3.5 Launching attacks using SMM resources 62
3.6 Discussion 66
3.7 Summary 66

4 Requirements **69**
4.1 Introduction 69
4.2 Threat model 69
4.3 Assumptions 72
4.4 Requirements for using SMM for security purposes 72
4.5 Discussion 74
4.6 Summary 76

5 A Generic Architecture for SMM-Based Security Tools **79**
5.1 Introduction 79

5.2 Requirements Specification 79
5.3 General Architecture 82
5.4 Architecture Design 82
5.5 Discussion 97
5.6 Summary 99

6 Implementation and Evaluation - Manager Module and SBST 105
6.1 Introduction 105
6.2 Functions in the Manager Module 108
6.3 Manager Module Computational Experiments 117
6.4 SBST Implementation and Evaluation 130
6.5 SBST Limits and Constraints 132
6.6 Manager Limits and Constraints 133
6.7 Discussion 133
6.8 Summary 133

7 Conclusion 135
7.1 Directions for Future work 137
7.2 Investigate the interaction of an $SBST$ with technologies in the chipset 137
7.3 Investigate the Impact of SMI Latency 137
7.4 Optimize the Proof of Concept Execution Time 138
7.5 Embed the Tool in a BIOS to Test It in a More Realistic Scenario . . . 138

A Specific SMM Registers 139
A.1 Chipset 1 Specific Registers 139
A.2 Chipset 2 System Management RAM Control register 148

Bibliography 155

List of Figures

1.1 Machine and Chipset 1 4
1.2 Machine and Chipset 2 5
1.3 First set of machines used in the experiments 5
1.4 Second set of Machines used in the experiments 6
1.5 General architecture 8

2.1 Security Context 16
2.2 The Cloud Reference Model (figure from [131]) 21
2.3 Virtualisation Layer 24
2.4 Consolidation 24
2.5 Containment 25
2.6 Full virtualisation (figure from[140]) 28
2.7 Paravirtualisation (figure from [140]) 29
2.8 Hardware-assisted virtualisation (figure from [140]) 30
2.9 Xen Architecture 31
2.10 Intel Hub Architecture, based on [57] 32
2.11 The Intel Platform Controller Hub 33
2.12 Ring security scheme, based on [57] 35
2.13 The PI and UEFI layers, based on [153] 37
2.14 TPM Overview (figure from [54]) [57] 39

3.1 SMM components 47
3.2 SMRAM space for 32-bit machines 48
3.3 SMRAM space for 64-bit machines 49
3.4 SMRAM control register 50
3.5 SMRAMC state before SMRAM initialisation 51
3.6 SMRAMC state after SMRAM initialisation 51
3.7 SMI Frequency Graph 54
3.8 SMI Frequency Graph 55
3.9 Entering and exiting from SMM 56
3.10 Current SMM-based security tools architecture 59

4.1 Threat model 71
4.2 Mitigating threats 75

5.1 General architecture 80
5.2 Execution flow of the algorithms 85
5.3 Algorithm Measure Integrity 86
5.4 Algorithm Manage Load Data 90

5.5 Algorithm Load Data . . . 92
5.6 Algorithm Manage Compute Hash . . . 94
5.7 Algorithm Compute Hash . . . 96
5.8 Algorithm Manage Verify Hash . . . 98
5.9 Algorithm Verify Hash . . . 101

6.1 Manager program files . . . 107
6.2 Manager main menu . . . 109
6.3 SMRAMC status reported by the manager module for machine 12. Offset 88H. . . . 118
6.4 SMRAMC status reported by the manager module for machines 1 to 5 and 13. Offset 90H for machines 1 to 5 and 9dH for machine 13. . . . 119
6.5 All PMBASE SMM Related Registers Status. . . . 120
6.6 SMI_EN Register Status. . . . 120
6.7 SMI_STS Register Status. . . . 121
6.8 GPE0_EN Register Status. . . . 123
6.9 GPE0_STS Registers Status. . . . 124
6.10 ALT_GPI_SMI_EN Register Status. . . . 125
6.11 ALT_GPI_SMI_STS Register Status. . . . 126
6.12 ALT_GPI_SMI_EN2 Register Status. . . . 126
6.13 ALT_GPI_SMI_STS2 Register Status. . . . 127
6.14 GEN_PMCON_1 Register Status. . . . 128

List of Tables

1.1 Main SMM-Based security tools ($SBST$) 7

2.1 Intel Chipsets 34

3.1 SMI latency 53
3.2 SMI average latency 53
3.3 SMI frequency 54
3.4 Reported execution time 62

5.1 Persistant variables 88
5.2 Non-persistant variables 88

6.1 Execution time 132

List of Theorems

2.1 Virtual Machine Monitor 26

List of Definitions

2.1 System Executive Software . . . 17
2.2 Code . . . 17
2.3 Agent . . . 17
2.4 Basic Code . . . 17
2.5 Atomic Function . . . 17
2.6 Task . . . 17
2.7 Set of Tasks . . . 17
2.8 Round . . . 17
2.9 Payload . . . 18
2.10 Set of Payloads . . . 18
2.11 Set of Data . . . 18
2.12 Memory Unit . . . 18
2.13 SMI Handler . . . 18
2.14 Set of Registers . . . 18
2.15 Resume Instruction . . . 18
2.16 Set of SMI . . . 18
2.17 Maximum Latency . . . 18
2.18 Maximum Memory Size . . . 18
2.19 Set of Requirements . . . 18
2.20 Set of Threats . . . 18
2.21 Set of Assumptions . . . 19
2.22 Time Elapsed Measurement Function . . . 19
2.23 SMM-Based Security Tool . . . 19

List of Algorithms

5.1 Measure the integrity of hypervisor dynamic data 87
5.2 Manage to call function LoadData( ) 91
5.3 Load the data to be measured from DRAM to SMRAM 93
5.4 Manage to call function ComputeHash( ) 95
5.5 Compute data hash 97
5.6 Manage to call function VerifyHash( ) 99
5.7 Verify data hash 100
5.8 Verify and set registers to reinforce SMM security 101
5.9 Verify SMMR interface integrity and trigger a late launch instruction . 102

Chapter 1

Introduction

> Only the paranoid survive.
>
> ANDREW S. GROVE

1.1 Motivation

There are many security issues around computer systems. Thus, it is necessary to enhance security and perform verification tasks to minimize the risk of potential threats become successful attacks. These tasks are usually performed by security tools, such as anti-virus.

In this context some concepts, such as isolation, privilege and view, are fundamental. Security tools must operate isolated to perform their tasks with minimal risk of being hit by the problems they are trying to solve. Security tools need to have high privilege to get in action at the right time and must have a global view of the system, but also good ability to view and deal of its own environment to enhance the chances of success when performing their tasks.

A common security task demanding isolation and high privilege is to measure for checking the integrity of system components, such as the System executive software (operating system or hypervisor). Such verification can be static or dynamic (at run-time).

Generally speaking, to measure the integrity of hypervisors in a static manner, a tool performs a cryptographic service to generate a hash code of the hypervisor static code or data laid out in the main memory. To check the integrity it is necessary and sufficient to generate again the hash code of the same hypervisor static code or data in the memory and compare them. If the hash codes are equal, no changes have happened and then the integrity is preserved. The dynamic measurement is much more complex because the hypervisor dynamic code and data will vary along the hypervisor execution. In both cases, the tool must be deployed in an isolated environment, with enough access privilege whilst preserving the view of the hypervisor code and data.

Then, the problem security analysts are facing is to deploy a security tool in an isolated and high privileged environment, with good view of the system and minimal risk to be tampered with by any threat.

When a System Management Interruption (SMI) is generated the processor enters in the System Management Mode (SMM). So a set of powerful resources, as isolated and protected memory, become available to the SMM executive software (called SMI handler). Those resources can be used by a security tool so it becomes

as resourceful as the SMI handler. More details about the SMM resources on Chapter 3.

The System Management Mode (SMM) was introduced in IA-32 architecture into Intel 386SL processor released in 1990, aiming to provide resources to manage the system [87]. Since then, SMM has been a standard architectural feature in IA-64 and IA-32 processors [57]. A key feature of SMM is its high privilege; consequently, the SMM executive software (SMI handler) has higher privileges than the system executive software (operating system or hypervisor), allowing it to manage critical system tasks.

The Intel 386SL processor is mobile version of the Intel 386 processor [66, 71, 87, 97]. Mobile computers have strong requirement for power saving, so they need a robust way to manage power in the system. Thus, SMM was developed to meet such a demand and provide power management and other system management functions, such as hardware control. Prior to SMM, most of these functions were accomplished by In-Circuit Emulation (ICE) tools.

Due to its powerful resources, as: isolated memory, high privilege, high priority and complete view of the system; SMM has been used for purposes other than those described in the Intel manuals [85, 86, 87].

In recent years, SMM was exploited successfully in different ways. For example, by circumventing its protection mechanisms using a technique called “cache poisoning” as described in [45, 148] and by exploiting a fail in implementation in the SMM executive software (SMI handler), which lead to breach the Intel TXT security to exploit the Xen hypervisor, as described in [147]. On the other hand, security researchers have been presenting solutions that capitalise on SMM resources to design and implement new security tools, as HyperSentry [14] to measure the integrity of hypervisors, SICE [15] to provide an isolated execution environment and AppCheck [142] to protect applications by inspecting their code in the physical memory. A discussion about those tools in the context of this work can be seen in 3.4.

Because of this, Intel, OEMs, BIOS and other related manufacturers, have been: 1) extending specifications, such as the addition of a new register to deal with ”cache poisoning” attacks [87, 46, 148] (section 3.2.3); 2) changing implementations, such as setting the BIOS to lock the SMM access control register [87, 23, 47] (section 3.2.3 and section 3.5); and 3) introducing new technologies, such as (those technologies, their interrelations and interactions with SMM are discussed at particular sections in chapter 3, as detailed ahead): the Intel Trusted Execution Technology (Intel TXT) [57, 58] (section 2.4.3), the Dual-Monitor Treatment in the Virtual Machine Extensions (VMX) [87] (section 3.3.4) and the Intel Software Guard Extensions (SGX) [111, 60, 10] (section 3.3.5). These changes and their impact on security are discussed later in this thesis, mainly in chapter 3. Observing previous Intel Architecture Software Developers Manual (as[66, 71]) and the chipsets manuals [64, 65, 80, 81], considered in our target chipsets for this research, one can notice that those characteristics and technologies were not or are not present in all machines.

Despite of a considerable number of works capitalising on SMM, there are a lack of research on investigating the SMM itself and its fundamentals. The found works report the use of SMM resources and how they take advantage of some SMM

characteristics, as the use of the SMM isolated memory space (SMRAM), discussed in 3.4. In spite of that, there is a lack of research to understand SMM, establish requirements to use it for security purposes and define an architecture to use SMM as a platform for security tools.

In particular, the SMM-based security tools studied have not been taking full advantage from the SMM resources, as isolation and transparency (the SMI handler executes unnoticed by the system executive software). Also, many of those tools do not abide all the SMM constraints and limits, as latency time. In general, their architectures are modular, being partially deployed in SMM and making connections with their other parts deployed in unprotected areas in the system, which let them subject to general attacks.

Thus, this research investigates the System Management Mode in the context of Intel processors to establish a set of requirements and then design a generic architecture for SMM-based security tools. That generic architecture is tested by building a proof of concept to dynamically measure the integrity of the Xen hypervisor. This measurement is limited to the minimum necessary to prove the concept of the architecture.

1.2 Limits and Scope

This research limits its scope to Intel processors and chipsets in the x86 platform. We chose Intel due to the amount of open publications, research, attacks and security tools on Intel chipsets and processors available and due to Intel has the biggest market share for processors in the world. However, AMD processors based on the x86 architecture are similar to Intel processors, including the SMM.

We will develop a proof of concept and a manager module for being used in two machines with different chipsets. The reason behind that is because SMM is specific for each machine and the set of resources and registers, SMM or non-SMM, vary depending on the chipset and processor in use at the considered machine. So, any SMM-based tool must be developed specifically for a particular machine. We use Linux as the operating system in all machines utilised in our experiments due to easiness to access PCI bus and the components connected to that bus. In our early experiments we noticed that the Linux distribution CentOS 5.11 is the most appropriated operating system for our work, since that operating system installs Xen hypervisor during its own installation and the tests with a version of our proof of concept using the library libpci was successful in accessing the set of registers we targeted in this work.

The proof of concept aims to perform measurement of essential data of a hypervisor. We choose the Xen hypervisor 4.0 because the amount of open publications and researches available for that hypervisor and because it fits well in the CentOS 5.5 operating system. We focus on **xend-config.sxp** file for the measurement (more details can be found in chapter 6). It is noteworthy that the focus of this work is not to provide a research around measuring the integrity of hypervisors, but capitalising on SMM resources to provide an environment isolated, high privileged and with good view for security tools. In general, the term SMM is used to refer to the set of resources available in the system when the processor is operating in the System Management Mode.

Figure 1.1: **Machine and Chipset 1.** The Acer Aspire X1935, endowed with Intel i5-3450 processor, has the SMRAMC register locked and SMRAM not opened after the bootup process.

The main chipsets considered in this research are (For checking all machines used in this thesis, please check table 3.1):

- **Machine and Chipset 1.** ACER Aspire X1935, Intel core i5-3450, 3.1GHz, 8GB RAM, Northbridge Ivy Bridge 3rd generation, Southbridge B75, socket Intel LGA 1155. Year of releasing: 2012 [9].

- **Machine and Chipset 2.** Compac Evo N410c, Intel Pentium III-M, 1.2 GHz, 8GB RAM, Northbridge 82830M, Southbridge ICH3-M. Year of releasing: 2001 [32].

The criteria for choosing those machines was to have one old computer where potentially the SMRAMC register (System Management RAM Control Register see sections and , for more details on this register) is unlocked and SMRAM is opened after the bootup process and another modern computer where potentially the SMRAMC register is locked and SMRAM is not opened after the bootup process. The SMRAMC register is discussed in details in section 3.2.3.2 and the SMRAM in section 3.2.2.

From the previous paragraph comes our main limitation in this research: to test our proof of concept and met requirements *r3* (persistent) and *r4* (cooperative) (the requirements are discussed in 4.4), we need to insert our tool in the the SMM executive software code (called SMI handler), which is available in coreboot [36] (more details about coreboot in section 2.4.1), compile coreboot, record the compilation into a ROM-BIOS chip, install that chip in the target chipset and them test the whole system. However, we do not have all resources for doing these steps in this research. Thus, we assume that our proof of concept is already compiled and embedded in the BIOS, as described in section 4.3. An implication from this limitation is that we just can test our proof of concept in machines which has the SMRAMC register unlocked and SMRAM opened after the bootup process, by copying it from

Figure 1.2: **Machine and Chipset 2.** The HP Compac Evo N410c, endowed with Intel Pentium III processor, has the SMRAMC register unlocked and SMRAM opened after the bootup process.

Figure 1.3: **First set of machines used in the experiments.** This is the first set of machines used in this research (table 3.1).

DRAM to SMRAM. Although this can have implications to requirements *r5* (isolated) and *r8* (complete) (section 4.4), our assumption also guarantee us to meet requirements *r5* and *r8*, considering the objectives of this thesis.

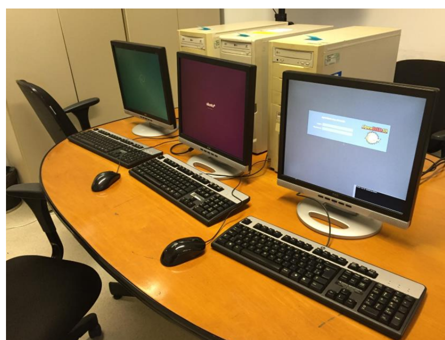

Figure 1.4: **Second set of Machines used in the experiments.** The second set of machines used in our research (table 3.1).

This research is focusing on the issue of using SMM for security purposes. Attacks or exploits related to SMM are just considered when they are useful for improving the issues related to security.

1.2.1 Scope considering other SMM-Based Security tools

In this section we aim to highlight the main differences, meaning the points not covered for other $SBST$ (those one described in table 1.1) and how to make those tools take more advantage from SMM, as by using more SMM resources or by using those resources more intensively.

Section 3.4.1 details those tools and presents a brief analysis of them. First, we identify the opportunities for improvement in the tools. For example: HyperCheck [141], HyperSentry [14], auditing tool [61], AppCheck [142], MUSHI [151], SPECTRE [149] and IOCheck [150] use a similar architecture, taking in account a remote machine to analyse the collected data and for other management functions (figure 1.5). Although that modularised architecture might address and even overcome issues around the limited amount of SMM memory [87] and time limit constraint of 150 μs, which is the maximum amount of time the processor must spend when executing in SMM [90], it opens new ways for attacking the host machine and the tool, as the communication channel (although some tools use attestation mechanisms to prevent that), the drivers and devices used to enable communication and the own remote machine. Moreover, they enlarge the trusted computing base (TCB), since all those items should be in the TCB.

Thus in our proposal all $SBST$ components must be deployed in the SMM memory (the SMRAM) to take advantage from the isolation provided by SMM and to minimise the TCB. The $SBST$ must execute in the available time according to the algorithms described in chapter 5. Note that, according to chapter 6 our proposal has also a manager module, which does not need to be deployed in the SMM memory or even in the target machine. Since the manager module is used for learning about and researching the target machine, the manager might be used in a machine endowed with the same chipset of the target machine.

Another important issue is that since when entering SMM the processor looks for the first instruction to be executed at the address formed by the content of the

SMBASE register + 8000H (by default the SMBASE register content is 30000H, see section 3.2.3.1 for more details) in SMRAM, where the SMM executive software (SMI handler) is located. This implies that any $SBST$ must be a modified version of the SMI handler. So, tools use a modified version of the original SMI handler. For example HypeBIOS, SPECTRE and IOCheck use an SMI handler from Coreboot [36], but it is not clear what happens to the original functions of the SMI handler after the modifications have been made, or the level of cooperation between tools and the original SMI handler. So in our proposal we preserve the code and de functions of the original SMI handler, by working cooperatively with it.

To start the SMI handler and consequently the $SBST$, a System Management Interruption (SMI) must be generated in the chipset. There are common ways to trigger an SMI to start the security tools as writing to the Programmed I/O Port 0xB2H [23, 45]. Because it is a common way to trigger an SMI any attacker might aim to thwart such action by denying the use of that port. Thus, an $SBST$ should take advantage from any SMI generated to start executing its job, which is the simplest way to start and keeping the tool operating. Then our proposal is SMI-independent (requirement *r7*, the requirements are discussed in 4.4), which means it can start at any SMI.

More details on our proposed architecture can be find in chapters 4, 5 and 6.

Table 1.1: **SMM-Based security tools ($SBST$).** This table list the main SMM-Based Security tools considered in this work, providing their objectives and the target platform (as Intel and AMD)

Tool	Objective	CPU
HyperCheck [141]	Check integrity of hypervisors	Intel
HyperSentry [14]	Check integrity of hypervisors	Intel
HyperGuard [146]	Check integrity of hypervisors	*
SICE [15]	Provide an isolated workload	AMD
SPECTRE [149]	Virtual machine introspection	AMD
IOCheck [150]	Check integrity of I/O devices	AMD
Auditing tool [61]	Auditing cloud computing systems	*
AppCheck [142]	Protect integrity of processes	Intel
MUSHI [151]	Provide isolated environment for VM	**
hypeBIOS [152]	Provide isolated environment for VM	AMD
BIOS Chronomancy[25]	Fix fails in a implementation of a CRTM	Intel

(*) Not defined in the paper.
(**) Not Implemented. It was only a proposal of framework for AMD platforms.

1.2.2 Scope considering other technologies present in the chipset

Considering our scope of Intel processors, the main technologies which might be present in the chipset and which might have some interaction with an $SBST$ are Intel TXT, Intel VMX, Intel SGX, TPM, UEFI. Again note that not all chipsets are endowed with VMX support (see section 3.3.4 for more details) and that SGX technology is not yet available commercially (see section 3.3.5 for more details).

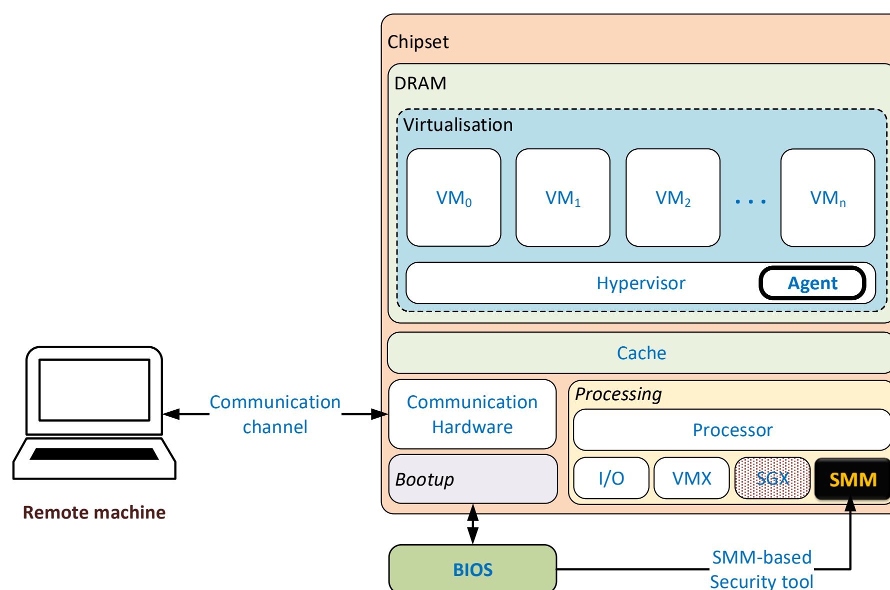

Figure 1.5: **Present *SBST* architecture.** This figure presents the current architecture used for most $SBST$s. The figure reflects a comum chipset, where the host machine is using a virtualization layer (section 2.3.2). It is noteworthy that not all chipsets are endowed with VMX support (see section 3.3.4 for more details) and that SGX technology is not yet available commercially (see section 3.3.5 for more details).

We have not identified any issue related to those technologies, which can have impact in our proposal, as making an $SBST$ stops working or behaving in an unpredictable way. However, it is noteworthy that: Intel VMX and its interactions with SMM are well defined (see section 3.3.4). Our proposal is based on BIOS, so we do not considerer the interaction of SMM with UEFI, however those interactions are specified in [2] and can be subject of a new research topic as also, TPM and SGX. We do not use TPM in this work, but we believe it can be used to improve our $SBST$ as discussed in HyperCheck [141]. SGX is not available yet commercially but there is some documentation available (see section 3.3.5 for details). Intel SGX might be a good answer for low-level security issues since it takes security to the processor level. However, we believe our $SBST$ proposal may work cooperatively with SGX. We do not identify any issue related to Intel TXT (see section 2.4.3, on Intel TXT).

1.3 Significance

An intended outcome of this research is to identify and establish a set of requirements, which should be met when developing a SMM-based security tool. Those requirements arise out of the limits and constraints of the SMM. The requirements contributes to harden the SMM-based security tool and make it more likely to be usable in a real environment, by enabling the tool to utilise more resources of SMM. A second intended outcome is to design a generic architecture for a SMM-based

security tool in compliance with the requirements. Generic architecture in this context means that the architecture holds for any SMM-based security tool, no matter its security task, provided that the requirements are met. The last intended outcome is to built a proof of concept to test and then demonstrating the validity of the architecture. This proof of concept should consist of a agent module, which is deployed in the SMM memory and implements the security task of the tool; and a manager module to help the tool developer to understand the particularities of a target chipset. The manager module does not need to be deployed in the SMM memory, since it is used to probe and research a target machine and then understand the capabilities and limitations of such a machine. The machine probed is not necessarily the machine where the agent will be deployed, but it needs to have the same or similar chipset.

1.4 Research Questions

This section defines a main research question and three specifics questions to guide this work. The subject of this research is how to use the System Management Mode to offer a resourceful environment to a security tool.

1.4.1 Main Research Question

The main research question aims to provide the direction for this work, guiding the research efforts.

Main question. *Can we specify a hardened and isolated security tool capitalising on SMM resources, while considering the limits and constraints of those resources?*

The System Management Mode has a set of powerful resources available to its executive software (SMI handler), which can be used to build a hardened and isolated security tool. However, those resources have firm limits, as amount of memory available, and constraints, as maximum latency time. A SMM-based security tool must abide by such limits and constraints. Some security tools capitalising on SMM resources, as HyperSentry [14], HyperCheck [141], auditing tool [61], AppCheck [142], SPECTRE [149] and IOCheck [150] use a modularised architecture, where a security task is divided in subtasks and each module is responsible to perform one subtask. Then, the modules are deployed in different parts of the system and in another machine. A common configuration in this scenario is to have a module in the SMM memory (SMRAM), another one in the DRAM or in a PCI card and another module in a remote machine. The remote machine is normally responsible for analysing the collected data in the target machine. That architecture is designed to deal mainly with the SMM memory limitation and the maximum latency time. Although it is a good approach, it lost the main reasons to use SMM: the strong isolation. Besides, this enlarge the TCB by adding, for example, a PCI card or another machine. In chapter 3 there are details, explanations and schemes about SMM and those tools

1.4.2 Specific Research Questions

To help answering the main questions, we propose three specific research questions. Each question addresses a specific phase of this research and serves as input to the next one. First one deals with the establishment of requirements to harden, isolate and improve a SMM-base tool, while dealing with the limits and constraints of SMM resources. The second question addresses the issue of design a generic architecture to meet the requirements. The last question addresses the feasibility of such a generic architecture.

Q1.*Can we establish a set of requirements to harden, isolate and improve an SMM-based security tool?*

Security tools can take advantage from SMM resources to harden themselves and enhance the capacity to accomplish their designed task. For example, The SMM memory offers strong isolation to the SMM executive code and data. Also, the SMM executive code has total control and global vision of the system when the processor enters in the SMM. Thus, to understand the limits, constraints, resources, mechanisms and components of SMM, it is required to investigate SMM, its resources and the system components involved in the SMM operation. As this research is considering Intel processors, it is also required to understand the relationship between SMM and other technologies present around Intel processor, such as Intel Trusted Execution Technology (Intel TXT) [57, 58], Intel Virtual Machine Extensions (Intel VMX) [87] and Intel Software Guard Extensions (Intel SGX) [111].

Q2.*Can we design a generic architecture to fit SMM-based security tools with different tasks, while meeting a set of requirements to harden, isolate and deal with SMM limits and constraints (by generic architecture, we meant one able to fit different security tools with different tasks)?*

Most existent SMM-based security tools, as HyperSentry [14] and HyperCheck [141], employ a modularised architecture comprising of modules deployed in different parts of the system to perform their tasks. Although such an architecture addresses some limitations and constraints of SMM, they leave modules out of the isolation protection of SMM memory. Allegedly, the majority of SMM-based security tools use SMM to achieve strong isolation. So, that modularised architecture does not make much sense. Besides, that architecture enlarges the TCB by adding system components, as PCI cards and remote machines. To take full advantage of SMM resources, a new architecture must be designed. That new designed architecture for SMM-based security tools must deal with all limitations and constraints of SMM and use SMM resources to harden themselves and guarantee the maximum isolation for the whole tool. Also, it need to be generic enough to fit security tools with different tasks.

Q3.*Can we build a proof of concept to demonstrate the feasibility of a generic architecture, which fits SMM-based security tools, addresses SMM limits and constraints, hardens the fit tool and isolate the whole code and data of the tool?*

Build a SMM-based tool is a tricky job, since SMM resources has firm limits and

constraints. For example, the maximum latency time recommend by Intel is 150 μs. Above that time there is the risk of system executive software time-outs [90]. Performing any security task in such a time limit is hard. Observe, for instance, table 3.4 with the execution time reported for some SMM-based security tools. There are other challenges, as how to deploy the tool into SMM memory to guarantee isolation, since the SMM memory access is locked when the processor is not in SMM. The access to SMM memory is allowed when the processor is in SMM, but the full control of the system is passed to the SMM executive software. Another challenge is how to start the tool once it is deployed in SMM, since de processor always starts the SMM executive software when entering in SMM. From this, arise out another challenge: the security tool needs to cooperate with the SMM executive software, since the SMM executive software is exclusive user of SMM resources and after the deployment of the security tool both will be in the same memory space and competing for the processor time. Another important issue is the chipset model of the target machine. Each chipset model and sometimes each chipset family (a set of chipset models) may specify SMM in a different ways. Also, any OEM manufacturing chipsets can implement SMM components in different ways too. Thus, a prove of concept should consist of two part: a security agent, to perform the designed security task; and a probe module to investigate the target chipset. Note that the probe module does not need to be deployed into SMM and it can investigate the chipset in any machine, provided that the chipset model is the same in the probed machine and in the target machine.

1.5 Contribution

1. **A detailed review and description of the SMM resources and components.** SMM resources and components have been changing over the years. Some changes were motivated by attacks, as locking the SMRAMC register (System Management RAM Control Register see sections 3.2.3.2 and A.2 after the bootup process. Other changes were motivated by the introduction of different technologies in the processor, as the dual-monitor treatment for SMM in the Intel Virtual Machine Extensions (Intel VMX) [87]. A deep understanding of SMM is essential to use it. So, this research investigate SMM and describe its resources according to Intel definitions. From a practical context, this research investigate SMM in two chipsets, identifying and describing its components. More details can be found in chapter 3.

2. **Analysis of SMM-based security tools and the opportunities to improve them.** This research analyses the implementation and architecture of security tools capitalising on SMM resources and their security tasks, as: HyperCheck [141], to check the integrity of hypervisors; AppCheck [142], to check the integrity of applications; SPECTRE [149], to perform virtual machine introspection; and IOCheck [150], to check integrity of I/O devices. After analysing the tools, this research describes opportunities to improve the security tools. More details in section 3.4.

3. **Analysis of SMM attacks and how to thwart them.** Attackers also have been capitilising on SMM resources by exploiting SMM to attack specific targets,

as the Xen hypervisor in [145], or to have a isolated and resourceful platform to deploy malwares [23, 47]. Thus, this research analyses the main published attacks against SMM and indicates how they can be thwarted. More details in section 3.5.

4. **Establishment of a set of requirements to use SMM for security purposes.** The SMM was created to manage high priority system management tasks. Moreover, Intel recommends in its Software Developer's Manual [87] that SMM should not be used for **"General-purposes"**. Such a recommendation aims to preserve the important SMM functions, since SMM resources have strict limits and constraints and is too risky let a "General-purpose" software violates those limits and constraints and disturb the correct function of SMM components, as by overwriting the SMM executive software SMI Handler (more details in section 3.2.4 and 3.4). However, the fact that the system offers timely and powerful resources when the processor is in SMM is motivating enough to investigate a way to use SMM in the more transparently way possible. So, this research establishes a set of requirements, which a security tool must met to overcome those limits and constraints, allowing the use of SMM for "Security-purpose". More details about the requirements in chapter 4.

5. **Design of a generic architecture for SMM-based security tools.** This research designs an architecture generic enough to fit SMM-based security tools with different functions. While there are many works designing tools for specific security purposes, this research focus on providing a architecture to be used for many security tools aiming to accomplish different security tasks. More details in chapter 5.

6. **A proof of concept to probe chipsets and manage and deploy a SMM-based hypervisor integrity measurement security tool.** Although the generic architecture is designed to fit SMM-based security tools with different functions it is neither convenient nor possible to build a proof of concept for all possible security tasks. Then, this research focus in build a proof of concept to one security tasks: measure the integrity of a hypervisor. In the context of this research to measure the integrity of a hypervisor means to define some portions of code or data or both and check the their integrity. So, this research focus on Xen Hypervisor, version 4, performing on CentOS 5.11, to perform measurement of file xend-config.spx. More details can be found in chapter 6

1.6 List of Publications

1. **Article:** *William de Souza and Allan Tomlinson. Virtualisation Without a Hypervisor in Cloud Infrastructures: An Initial Analysis - PGNet 2013, Liverpool, UK.*

2. **Article:** *William de Souza and Allan Tomlinson. Understanding threats in a cloud infrastructure with no hypervisor - WorldCIS 2013, London, UK.*

3. **Article:***William de Souza and Allan Tomlinson. A Threat Model for a Cloud Infrastructure with no Hypervisor - International Journal of Intelligent Computing Research (IJICR), ISSN 2042 4655, Issues 1/2, Volume 5 (2014), pp. 405-411.*

4. **Article:** *William de Souza and Allan Tomlinson. SMM Revolutions - IEEE BigDataSecurity 2015, New York, USA.*

5. **Article:** *William de Souza and Allan Tomlinson. SMM-based hypervisor integrity measurement - IEEE CSCloud 2015, New York, USA.*

1.7 Overview of the Research

The remain of this work is organised as follow:

Chapter 2 brings the background knowledge of this work, explaining the context and technologies involved in this research and building the definitions necessary to provide the answer to the research questions.

Chapter 3 investigates and details the SMM resources and components in chipsets. It presents related works and SMM-based security tools, analysing opportunities to improve those tools. It also presents the attacks capitalising on SMM and discusses if they are feasible nowadays and ways to thwart such attacks.

From the previous discussions, chapter 4 defines a set of requirements, which should be met for any SMM-based security tool.

Chapter 5 outlines a generic architecture for SMM-based security tools, describing and discussing how the requirements can be meet and the algorithms to implement the solution.

Chapter 6 discusses the implementation and evaluation of the manager module and the agent (the $SBST$ itself).

Chapter 7 presents the conclusion of this work and directions for future works in sub-areas of this research.

Chapter 2

Background

Sorry to be a wet blanket.
Writing a bitcoin description for general audiences is bloody hard.
There's nothing to relate it to.

"SATOSHI NAKAMOTO"

2.1 Introduction

Chapter 1 introduced the problem around security tools, which need to protect themselves, to have high privileges and good view of the system and of their own environment. In summary: strong isolation, high privilege and good view. Those issues motivated research to use SMM for security purposes. However, capitalising on SMM is a challenging task, considering its limitations, constraints and the components.

The present chapter contextualise the security problem addressed in this research, positioning that problem in the big picture of security in the computational system. The chapter presents a set of definitions required to formulate and design the answers for research questions. It discusses the context of security in an abstract view from the highest level considered in this work, a cloud, to the lowest, SMM and its resources in a machine hosting a virtualised environment, by looking at the security issues on cloud computing and virtualisation and describing the principles behind those technologies. We call this contextual issue as vertical security level. Then, it discusses the environmental issues around the SMM: the components in the host machine competing for processor time, memory space and with concurrent security tasks; by analysing and describing the current state-of-art of technologies as UEFI and Trusted Computing. We call this environmental issue as horizontal security level. Finally, the chapter analysis and discusses works related to the system executive software integrity, as integrity measurement, hardening and new architectures, which are considered when building the proof of concept in chapter 6. Section 3.4 addresses security tools capitalising on SMM.

Figure 2.1 presents a overview of the security problem discussed in this research. The context of the problem is a cloud computing environment, comprising of one or more machines represented in the figure by their chipset. The chipset, as defined in section 2.3.3, is a more useful artefact to be considered in this research than the complete machine. Each chipset hosts in its main memory (DRAM) a virtualised environment comprising of one manager virtual machine (VM_0), one or

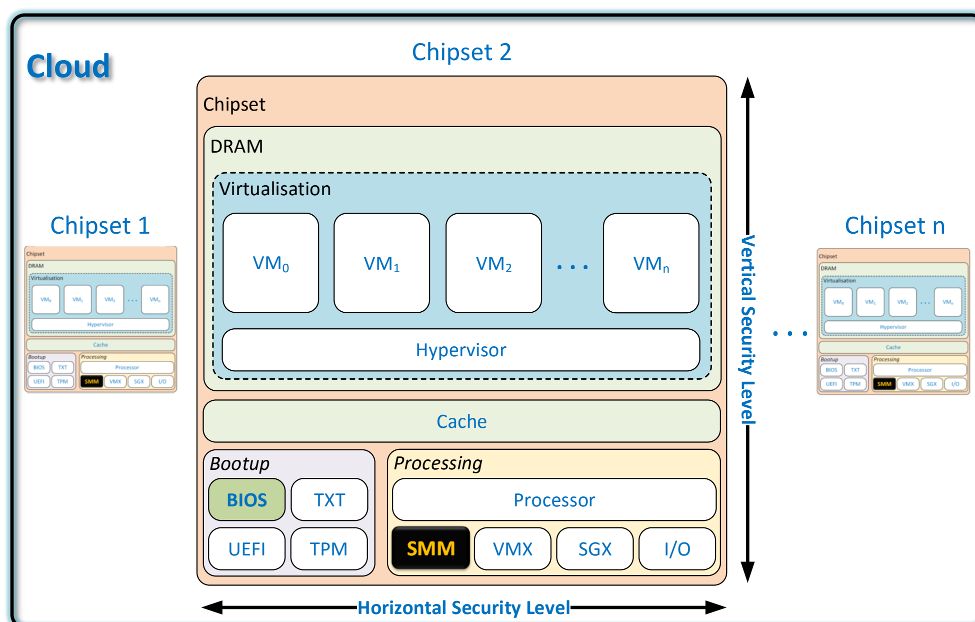

Figure 2.1: **Security Context.** The security context of this research problem comprises of a vertical security level from the more general concept of a cloud computing, which is more visible for users, to the more specific component the SMM, which is chipset specific; and a horizontal security level considering the bootup process and processing issues and the technologies related to them.

more guest virtual machines (VM_1 to VM_n) and a hypervisor. The cache memory takes part of the problem, since some SMM attacks are perpetrated via cache [45]. Then, we have two low-level blocks: bootup, consisting of components and technologies related more to the security issues during bootup process, as BIOS, UEFI, Intel TXT, TPM; and processing, consisting of components and technologies related more to the security issues during the processing tasks. So, the vertical security level puts the research problem in context, relating it to security issues in the: cloud, chipset, memory, virtualisation layer and cache. The horizontal security level considers the research problem in its environment, relating it to security issues in the components of the bootup and processing blocks.

The chapter is organised in four main sections and a discussion and a summary section. The first section presents the set of definitions built to develop the solution for the research problem presented in the chapter 1. Second section addresses the vertical security level around the SMM and third section addresses the horizontal security level around SMM. The fourth section presents the works related the system executive software integrity. After, a discussion and a summary of the work reported in this chapter are presented.

2.2 Definitions

This section presents the definitions used when establishing the requirements and designing the generic architecture.

In this work, the term “Technology” is used to designate an implementation of a technique. For example, a hypervisor is a technique for managing virtual machines. Xen [18, 28, 137, 59] is an implementation of a Hypervisor. So, in that example the hypervisor is the technique and Xen is the technology.

Numerical bases are represented by a letter in the rightmost end of each number, as follow: **H for hexadecimal, B for binary and D or no letter for decimal**.

In this work we use the verb “set” to indicate the action in which a register or a bit in a register (a position in the register, as in figure 3.4) have its value changed to “1” and the verb “clear” to indicate the action in which a register or a bit in a register (a position in the register) have its value changed to “0”.

Definition 2.1 (System Executive Software)
The system executive software ***SES*** *is a code artefact with management functions in a computational system. Examples of system executive software are operating system and hypervisors.*

Definition 2.2 (Code)
Code *is a set of instructions written in any programming language to perform designed actions in a deterministic way.*

Definition 2.3 (Agent)
An agent *is a code artefact designed to take a preprogrammed action autonomously, whenever it perceives a determined change in its environment.*

Definition 2.4 (Basic Code)
A basic code ***bc*** *is a piece of code comprising of basic management functions in a SMM-based security tool. The rationale behind the* ***bc*** *is to perform tasks which are independent of the payload loaded into the SMM-based security tool.*

Definition 2.5 (Atomic Function)
An atomic function *is a set consisting of one or more instructions performed in sequence, without interruption and constrained by a time limit. That time limit must be less than 150* μs*. An example of atomic function is computing the hash code of a memory region.*

Definition 2.6 (Task)
A task t_i *is a well-defined function of a SMM-based security tool, which can be divided into a finite set of subtasks as* $t_i = \{t_{i1}, t_{i2}, ..., t_{nm}\}$*, where each subtask* t_{ij} *performs an atomic function. An example of well-defined task is measure the integrity of essential hypervisor data.*

Definition 2.7 (Set of Tasks)
A set of tasks $T = t_1, t_2, ..., t_n$ *is a set, where each task* t_i *performs a well-defined function. It is not possible to say, a priori, that the set of tasks is a finite set.*

Definition 2.8 (Round)
A round *is the amount of work done by a function related to a task or a sub-task being executed, without crossing the time limit available, which in general is not enough to complete*

*a task or a sub-task. A round starts with when an smi_i is triggered and finishes when a **rsm** instruction is performed.*

Definition 2.9 (Payload)
A payload ***p*** *is a piece of code embedded into* **bc** *to perform a well-defined task. A payload can be thought as a task implementation.*

Definition 2.10 (Set of Payloads)
A set of payloads $P = \{p_1, p_2, ..., p_n\}$ *is a set, where each payload p_i performs a well-defined task. It is not possible to say, a priori, that the set of payloads is a finite set.*

Definition 2.11 (Set of Data)
A set of data $D = \{d_1, d_2, ..., d_n\}$ *is a finite set of data to support the execution of a SMM-based security tool.*

Definition 2.12 (Memory Unit)
A memory unit $mem_{SPACE}[i]$ *is a memory space, such as $mem_{RAM}[i]$, $mem_{SMRAM}[i]$, $mem_{BIOS}[i]$, where i is an index in the memory space, indicating a specific location in that memory space.*

Definition 2.13 (SMI Handler)
The SMI handler ***SH*** *is the* ***SMM*** *executive software, which is implementation, chipset and OEM dependent.*

Definition 2.14 (Set of Registers)
A set of registers $REG = \{reg_1, reg_2, ..., reg_n\}$ *is a finite set of registers related to SMM.*

Definition 2.15 (Resume Instruction)
The resume instruction ***rsm*** *is an instruction to signal the processor to exit from* ***SMM****.*

Definition 2.16 (Set of SMI)
A set of SMI $SMI = \{smi_1, smi_2, ..., smi_n\}$ *is a finite set of interrupts to signal the processor to enter* ***SMM****, which is implementation, chipset and OEM dependent.*

Definition 2.17 (Maximum Latency)
The maximum latency ***ml*** *is the maximum time a SMM-based security tool can spend when executing in* ***SMM****.*

Definition 2.18 (Maximum Memory Size)
The maximum memory size ***ms*** *is the maximum memory size a SMM-based security tool can use in $mem_{SMRAM}[i]$.*

Definition 2.19 (Set of Requirements)
A set of requirements $R = \{r_1, r_2, ..., r_n\}$ *is a finite set of requirements that must be met by a SMM-based security tool.*

Definition 2.20 (Set of Threats)
A set of threats $H = \{h_1, h_2, ..., h_n\}$ *is a finite set of threats identified in the current SMM-based security tool architecture.*

Definition 2.21 (Set of Assumptions)
A set of assumptions $A = \{a_1, a_2, ..., a_n\}$ *is a finite set of assumptions made to deal with complexity and time constraints due to impossibility or inconvenience to tackle all issues or threats identified in the research work.*

Definition 2.22 (Time Elapsed Measurement Function)
The time elapsed measurement function $te(f())$ *is a function to measure the time elapsed of a function* $f()$.

Definition 2.23 (SMM-Based Security Tool)
A SMM-based security tool ***SBST*** *is defined as* $SBST = \{bc, p_i, t_i, D, mem_{SMRAM}, rsm\}$. *Thus, an* ***SBST*** *has a basic code, responsible for management functions and a payload performing a well-defined security task.* ***SBST*** *has a set of data* **D** *to support its execution and it should be laid in the SMRAM; the instruction* ***rsm*** *must be used by* ***SBST*** *to finalise its execution.*

2.3 Context and Technologies

This section explains and contextualises the problem addressed in this research, considering the abstract vertical security level, as showed in figure 2.1. We consider the cloud the highest level because it is a more general concept and more visible to users, whereas the SMM is the lowest level because it is more specific, depending on the chipset, less visible to users and high privileged. As a general rule the lower the component in the vertical security level, the higher the privilege. In the vertical security level, cloud computing is considered the highest entity, comprising of one or more machines, each machine represented for its own chipset. The chipset has many parts, including the virtualisation layer, memories, components of *bootup* and processing. The virtualisation layer are hosted by the DRAM memory and has the virtual machines in its high level and the hypervisor in low level. Cache memory is positioned in a lower level than DRAM, since it is accessed before the DRAM by the processor, so this is a sort of privilege. Finally, we consider the SMM as the lowest level due to its powerful resources and we also consider that an successful attack to SMM can compromise all the higher components In the vertical security level. The components of bootup block are discussed ahead in the present chapter and the components of processing block are discussed in chapter 3.

2.3.1 Cloud Computing

Cloud computing provides an infrastructure for customers to run their applications and store their information. It allows several virtual machines, from different customers, to exist on the same physical machine capitalising on economy of scale, in a dynamic and scalable computational environment at a cost affordable for customers [48, 131]. Although this is the main appeal for a cloud infrastructure it is also the main concern for customers, since the shared environment is prone to threats that can be exploited for a malicious party [30, 38, 63]. For example, a malicious VM can start an attack against the whole infrastructure, targeting components as: another VM running on the same server, the hypervisor or the underlying hardware; and potentially exploiting a wide range of vulnerabilities in the virtualised environment [108, 126].

The importance of cloud computing is more related to the benefits for businesses than its technical aspects. So, the discussion about cloud computing should start in a business environment, where a company needs to decide to keep its own infrastructure or migrate it to the cloud environment [62]. Migrate means to move IT issues and resources to the Cloud Provider, with potential cost reduction, faster time to market, greater market share, innovation and customer loyalty [55].

There are some confusion about the term cloud, specially because the own Internet was referred as a cloud because users access resources through a browser with no idea where such resources are located. To clarify that issue a set of features based on [55, 112, 114, 131] is defined and listed below, so that any pool of resources must have those features to be considered as a cloud computing environment:

- **Internet technologies.** The pool of resources must be available by means of internet technologies.
- **Services-based.** The pool of resources is offered as well-defined services, ready to be used by users.
- **Scalable and elastic resources.** Resources available for any user in the pool must increase or decrease, according to the demands of such a user.
- **Measured use.** The use of resources in the pool, formatted as a services, must be accounted, measured and billed to clearly express their utilisation by any user.
- **Shared resources.** The resources in the pool are used by one user or shared by two or more users, transparently, allowing economy of scale.
- **Fault tolerant.** The resources in the pool must be fault tolerant in such a way that whenever a failure occurs, the resources of users must be migrated from failed resources to working resources, transparently for users.
- **Security.** The pool of resources must be protected against internal and external threats, including the protection and isolation among users.

Once a pool of resources can be recognised as cloud computing and considering the Services-based features, a cloud computing provider can adopter three service models, as described below [55, 112, 131]:

- **Software-as-a-Service (SaaS).** This model has a rich variety of services and offers software as service in the cloud environment. Some examples are: the Google app suite, offering a wide range of application, the Salesforce, offering an applications to manage the sales process and customer relationship, and Symantec, offering security services as anti-virus and firewall from the cloud. Some variations of SaaS are Application-as-a-Service, Security-as-a-Service (as McAfee [7]), Information-as-a-Service and Management-as-a-Service [55].
- **Platform-as-a-Service (PaaS).** In this service model, the cloud provider offers a platform of hardware (network, processor, storage and so on) and software (operating system, compilers, libraries and so on), so the users can deploy

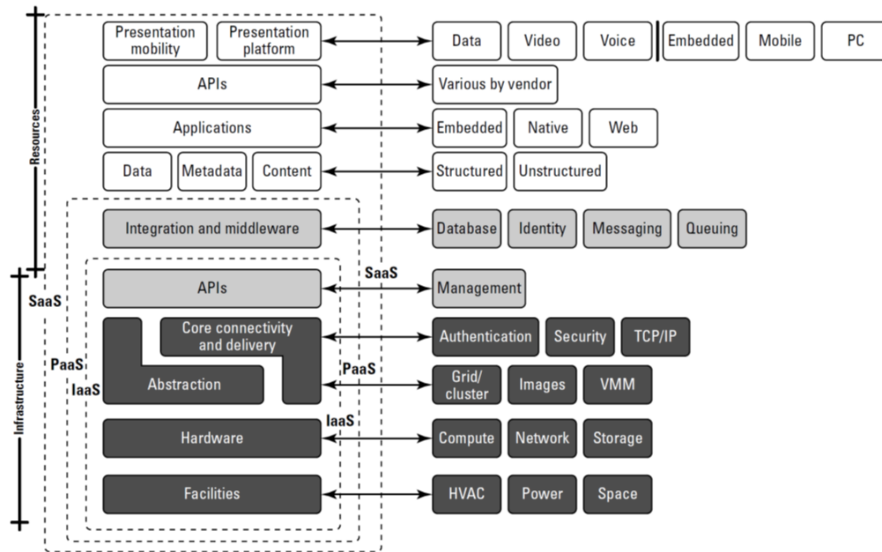

Figure 2.2: **The Cloud Reference Model (figure from [131]).** An abstract view of the cloud computing environment layers.

or migrate their applications to the cloud computing environment without worry about the amount of resources necessary in the moment of migration or in the future. Examples of such service model are: the Microsoft Azure [34], Red Hat [6] and Engine Yard [55].

- **Infrastructure-as-a-Service (IaaS).** This service model offers infrastructure to specific needs of users, as storage, processing power and communication capabilities, so that the user has control over those resources to a certain extent. Amazon Web Services is an example of this model [1]. Some variations of IaaS are Storage-as-a-Service, Database-as-a-Service, Network-as-a-Service and Computer-as-a-Service [55].

Figure 2.2 depicts an abstract view of the cloud computing environment layers, considering the concepts discussed before. This abstract view is also know as The Cloud Reference Model [131].

One last taxonomy is important about cloud computing. It is related to the deployment model, as listed below from the more restrict to the more embracing [55, 112, 131]:

- **Private cloud.** This model refers to a cloud computing environment built to serve a particular organisation. It can be built on or off premises and can be managed or operated by the organisation or a third party.

- **Community cloud.** A community cloud is built for two or more organisations which have common interest, as two or more Universities. It can be built on or off premises of one or more organisations sharing the community cloud and can be managed or operated by the organisations or a third party.

- **Public cloud.** This model refer to a open cloud computing environment, normally built in the public interest and managed and operated on premises by a government, educational organisation or a large company.

- **Hybrid cloud.** A hybrid cloud is formed by two or more of the previous deployment models, which keep their unities and identities, but are linked in some extent to form a unique cloud computing environment.

Most of the discussion about security in the cloud computing environment focuses on the upper layers of Cloud Reference Model (figure 2.2) and discussions about security related to hardware limits this scope to physical access control [30, 63, 108, 126]. However, as we will discuss in details in this research, to have good isolation, high privileges and good view of the system, security tools must capitalise on resources from the low layer in the abstract model (figures 2.1 and 2.2). In this sense, the SMM may be the right place to deploy security tools to defend the cloud computing environment [40].

A huge source of information about cloud is the Cloud Standards Customer Council. Its Wiki contains standards, use cases and a comprehensive set of technical documentation, ranging from "Cloud Customer Architecture for IoT" to "Impact of Cloud Computing on Healthcare" [5].

2.3.2 Virtualisation

Virtualisation is a central technology in data centres, where it has laid the foundation for advances and enabled the cloud infrastructure and cloud computing [129, 131]. Generally speaking, it is a technique used to simulate one or more computers in a single physical machine. By physical machine we mean a hardware device, such as a PC, a server or mobile device. It enables the execution of several and different environments with multiple operating systems on the same physical machine (host hardware).

The term virtualisation can be associated to different concepts. In this section, we will expose some of those concepts. However, this work is focuses on hardware virtualisation (physical machine resources).

2.3.2.1 Virtualisation Layer

Figure 2.3 presents an abstract view of the virtualisation layer. There are two main components in the layer: virtual machines (VM) and the hypervisor.

Virtual machines are software containers enabling operating systems to perform their applications in an isolated environment. In the example of figure 2.3 the VM_0 is a management VM and VM_1 to VM_n are the VMs containing the operating systems. Note that in VM_0 the operating system is called **Host OS**, since it manager some functions of the VM_1 to VM_n, whose operating systems are called **Guest OS**, meaning they are guest of the host OS.

A hypervisor also known as virtual machine monitor (VMM) is a high privileged component that manages the virtual machines in the same virtualised environment [129]. In an abstract view, it consists as a layer between the VMs and the hardware, and controls the guest OSs access to the machine resources. Two required features of a hypervisor are security, since it is a main target for attacks, and

resource scalability on-the-fly, i.e. the hypervisor should be able to allocate more resources from the host system without stopping the VM that needs the resource. The hypervisor manages all external interactions of VMs, including access to the host resource [116]. External interactions are done by means of VM exits. Consequently, the communication among different VMs or VMs and other components in the infrastructure is done indirectly through the hypervisor, by means of VM exits [135]. The exception for that rule is when a management VM is being used, since part of the management function are done by the management VM, as in the Xen Hypervisor [18].

A VM exit is a trap-and-emulate virtualisation implementation, similar to what happens in operation systems. It occurs when the VM code tries to execute a privileged instruction. When this happens, a VM exit occurs, the VM execution is interrupted (trap) and the hypervisor takes over execution to handle the privileged instruction (emulate) [129]. VM exits are rather frequent. As reported in [135], an idle VM running on Xen 4.0, VM exits occur approximately 600 times/s.

Figure 2.3 exposes the communication scheme in the virtualisation layer. VM_0 accesses the hypervisor via special calls (for example, hypercalls or binary translation). VM_1 to VM_n communicate with hypervisor via VM exit and the other way around via VM entry. VM_1 to VM_n access hardware resources by VM_0 through the hypervisor. VM_1 to VM_n can communicate each other and form a virtual network, using virtual network interface cards (NIC) via a virtual switches [116].

The virtualisation layer can be arranged in many ways, depends on the implementation. So, not all implementations have a management VM. One reason for have a management VM is to minimise the hypervisor size. In general, device drivers are located in the management VM. So, management VM (VM_0) and the hypervisor share management functions.

2.3.2.2 The Importance of Virtualisation

Besides the importance for enabling cloud computing, virtualisation allowed advancements in data centres, solving at least a fundamental problem, as follows: in many data centres, the demand for new physical servers had been increasing in a way that approached the physical room limits. This scenario contrasted with the situation of physical servers: while data centres were full of servers, the servers themselves were quite underutilised [116]. The adoption of virtualisation solves this issue by enabling the use of two procedures called **consolidation** (figure 2.4) and **containment** (figure 2.5).

Consolidation is when physical servers are converted into virtual servers, such that each physical server is converted to one equivalent virtual server and all virtual servers are hosted by only one physical server, which can be called virtualisation server. This new physical server adopting virtualisation can host several virtual servers. This brings some advantages, such as: decrease the numbers of physical servers, release room in the data centre, save power and allow a greater use of hardware resources (figure 2.4).

Containment is a similar concept, but with a subtle difference. After, consolidation in a data centre, containment is the next logical step. It consists in, whenever a new physical server is needed in the data centre its workload is virtualised in a new virtual server and hosted in an existent virtualisation server, instead of buying

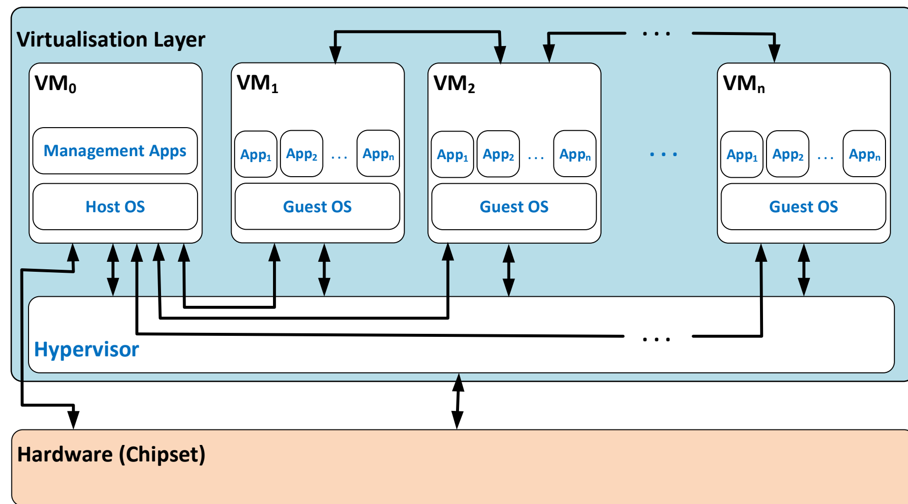

Figure 2.3: **Virtualisation Layer.** This example shows a particular arrangement of a virtualsation layer comprising of a Management VM (VM_0), VM_1 to VM_n (virtual machines containing guest operating systems) and a hypervisor. The arrows express the communication scheme in this particular arrangement.

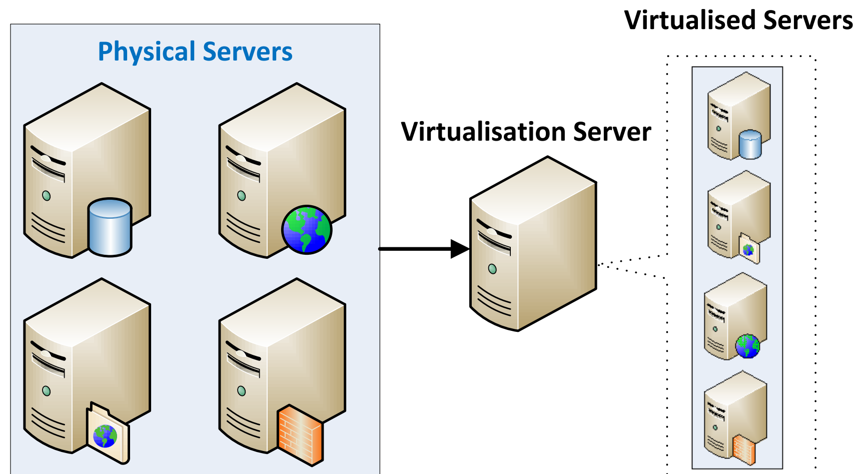

Figure 2.4: **Consolidation.** It consists in virtualising one or more physical servers and migrated them to only one new physical server. For example, four servers offering services of: database, web, web service and firewall; are virtualised and consolidated in just one physical server. That new physical server can be called virtualisation server.

a new physical server. The main advantages of containment are eliminating the necessity for more room in the data centre and decrease the physical servers' base,

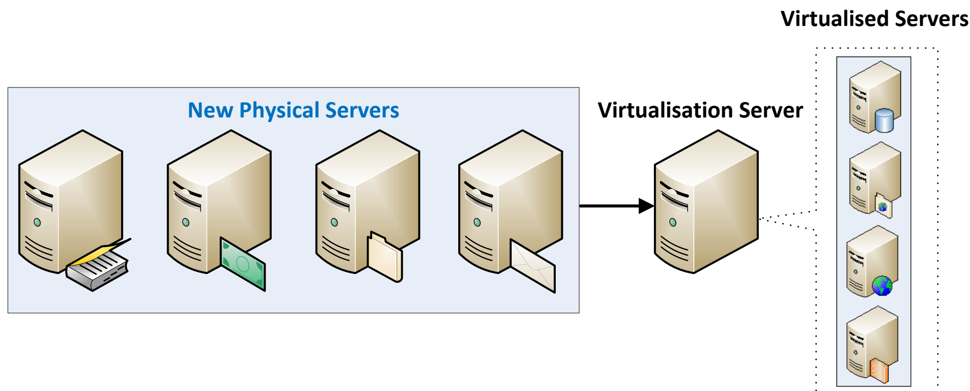

Figure 2.5: **Containment.** It is the next logical step after Consolidation. Whenever a new physical server is required, it is virtualised and hosted in a existent virtualisation server. For example, in a data centre, four new servers are required: name server, payment server, file server and email server. Instead of purchasing new physical servers, they are virtualised and hosted in the existent virtualisation server.

and the consequent hardware upgrades (figure 2.4).

2.3.2.3 Virtusalisation Historical Facts

Virtualisation is by no means a new subject in Computer Science. It arose almost with the advent of digital computer and the first works related to it are dated from the 1950's [116]. To a certain extent, virtualisation arose as natural evolution of time-sharing system [35]. In the early life of virtualisation, the main concerns were to determine whether a computer could be virtualised and understanding that virtualisation was not simply a time-sharing system, but a new technique that did not necessary exclude time-sharing systems [115].

In the past, virtualisation was associated mainly with cost concerns, since computers were expensive and in general could only perform one task at a time, leaving valuable hardware resources underused. Then, it was a paramount matter to look for a way to increase the resources ratio of use and serve more users. Nowadays, virtualisation regains its space and has becoming a growing research area, since it has many possible application besides those ones which motivated it to arise out [116].

2.3.2.4 Formalisation and Fundamentals of Virtualisation

With the rise of virtualisation, one of the main concerns was determining if a computer could support virtual machines, which meant whether a computer could be virtualised or not. In this direction Popek and Goldberg [115] presented formal requirements that test sufficient conditions for a computer architecture to support virtual machines. Although that work refers to virtual machines, in fact all tech-

niques and new components presented in Popek and Goldberg's paper were basically the same as is present in todays virtualisation. That includes the definition of a virtual machine monitor (or hypervisor) and its properties and the classification of instructions for virtualised environments, based on the behaviour of such instructions. The authors also use the term control program to refer to the hypervisor.

Three properties for a hypervisor were defined [115, 56]:

1. **Efficiency property.** Statically proved dominant instructions must be executed directly on the real processor, with no intervention by hypervisor.

2. **Resource control property.** The hypervisor must be in exclusive control over system resources and regain resources previously allocated to virtual machines, whenever necessary.

3. **Equivalence property.** All programs executed under the hypervisor must behave identically when running directly on a physical machine.

Popek and Goldberg proposed a classification of instructions for supporting virtualised environments, based on the behaviour of such instructions, as follow [115, 54]:

1. **Privileged instructions.** They can only be executed if the processor is in a supervisor state, or privileged mode and they cause a trap to the privileged mode, if executed with insufficient privileges.

2. **Control sensitive instructions.** Instructions that affect the resource configuration of a platform.

3. **Behaviour sensitive instructions.** They are those instructions whose behaviour depends on the configuration of resources.

Then, following the properties established for a hypervisor and the classification of instructions above, the follow theorem is defined [115]:

Theorem 2.1 (Virtual Machine Monitor)
For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions .

Although the theorem 2.1 refers to hypervisors, it can be extrapolated to the whole virtualised layer, since by constructing a hypervisor for any computer means enable such a computer to host a virtulisation layer and the hypervisor is a central piece in the virtualised environment. Moreover, it does not make sense to have a hypervisor without the virtualision layer. Also, the "third generation computer", as defined by the authors, can be extrapolated to contemporary commodity hardware.

2.3.2.5 Techniques of Virtualisation

There are different virtualisation techniques, which were developed for specific purposes. Sometimes those techniques can be used together to build resourceful environment, as a private cloud computing. Some of virtualisation techniques are described below. This research focuses on hardware virtualisation.

- **Hardware virtualisation.** This technique virtualises hardware resources from a physical machine. It offers to one or more virtual machines physical resources of a host machine, as memory and processor, giving them the illusion that all physical resources are available to virtual machines. In general, the access to physical resources is controlled by a hypervisor (figure 2.3). Some products that implement hypervisors to virtualise hardware are VMwares virtualization products, Microsoft Virtual Server, Citrix XenServer and the Xen hypervisor.

- **Server virtualisation.** This technique is used to virtualise physical servers. In this sense, a physical server is converted to a software workload. This workload is migrated to a virtual machine and from then on it is a virtual server. A single physical machine can host several virtual servers. This approach has many advantages, as scalability, power saving, eliminate the necessity for room in data centres or in IT spaces, easy to maintain, easy to upgrade, high ratio of hardware resources use in the host physical machines (see figures 2.5 and 2.4).

- **Desktop virtualisation.** Desktops can be virtualised by means of thin clients, which are physical machines with less hardware resources than a common computer (as a personal computer) can have. In this scenario, a virtualised desktop has a minimal set of resources necessary to connect to a server, download its workload and work as a common personal computer, but all heavy processing and storage are done by the server. The virtualised desktop user has limited access to the system resources. For example, to configure their workload, tasks, installing programs or executing anti-virus, must be performed by the server. Commercial solutions for this kind of virtualisation are Citrix's XenDesktop and VMware View.

- **Operating system virtualisation.** Operating systems can be virtualised to create an isolated environment to them. So, a virtual machine can be created and an operating system installed into it, allowing the operating system runs in a full-fledged way, as if it has all physical machine for it (figure 2.3). This kind of virtualisation has many applications (ways of using it). For instance: it can be used as a sandbox, allowing security and performance tests; and it can allow different and incompatible operating systems to execute concurrently in the same physical machine. Those applications take advantage of the strong isolation features provided by the virtualised environment. Another way to look to operating system virtualisation is observing the isolated environment provided by operating systems to their processes, as FreeBSD (jails), OpenVZ, Virtuozzo and Solaris (containers or zones) provide [59, 137], so that a process running in such operating systems can only accesses resources

allocated to it. CentOS, our target Operating System, has many supports to virtualisation, as virtual hosts and the Xen hypervisor and KVM [16].

- **Application virtualisation.** Applications can also be virtualised. This has many applications; for instance, to allow incompatible applications to run in the same physical machine, enabling a desktop user to have access to both applications. It is also useful to deploy and upgrade applications. Some examples of commercial products to virtualise applications are Microsoft's App-V, Citrix's Application Streaming and VMware ThinApp.

2.3.2.6 Virtualisation Categories

The three main categories of virtualisation are: Full virtualisation, Paravirtualisation and Hardware-assisted virtualisation [129, 140]. Those categories are sometimes referred as virtualisation implementations [129] or hypervisor implementations [54]. By design, the x86 architecture does not support virtualisation, since it has 17 non-virtualisable instructions that can be classified as sensitive instructions [115, 118] (see 2.3.2.4). Then, that is the issue to be solved by the categories or hypervisor implementations discussed below.

- **Full Virtualisation.** It provides a complete abstraction of the guest OS (figure 2.3), simulating the underlying hardware in such a manner that the guest OS is not aware about the virtualisation and has the impression that all hardware resources are allocated to it (figure 2.6). This is achieved by a combination of binary translation and direct execution [140]. Binary translation is a technique that replaces non-virtualisable instructions with new sequences of instructions. In order to improve the performance, user level code is directly executed on the processor. No modifications are necessary to either the guest OS or the underlying hardware. VMwares virtualization products and Microsoft Virtual Server are examples of full virtualisation.

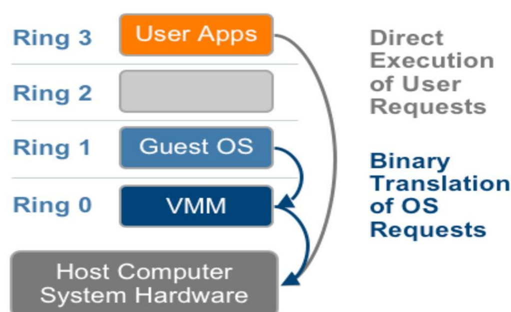

Figure 2.6: **Full virtualisation (figure from [140]).** It simulates the underlying hardware by combining binary translation, a technique that replaces non-virtualisable x86 instructions with new sequences of instructions, and direct execution of user level instructions (non-privileged instructions).

- **Paravirtualisation.** It addresses the problem of non-virtualisable instructions by modifying the guest OS kernel and replacing these instructions with hypercalls that communicate directly with the virtualization layer and provides hypercall interface for other critical kernel operations [140]. Thus, hypercalls play the same role in paravirtualisation that binary translation plays in the full virtualization techniques (figure 2.7). For instance, the instruction IOINSR, used when the guest software attempts to execute an I/O instruction, should be replace with a new sequence of instructions in binary translation, or transformed in a hypercall if paravirtualisation is being used. A hypercall is similar to a system call used in an OS, that is why in order to use paravirtualisation it is necessary modify the guest OS. In this case, commodity OS cannot be used. However, some operating systems as Ubuntu and Red Hat also offer support for paravirtualisation. The Citrix XenServer is an example of paravirtualisation.

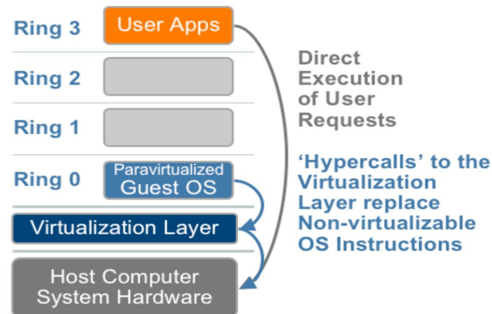

Figure 2.7: **Paravirtualisation (figure from [140]).** It replaces the non-virtualisable x86 instructions with hypercalls, which communicate directly with the virtualization layer. It is required that the guest OS kernel is modified to implement the hypercalls.

- **Hardware-assisted virtualisation.** It refers to a set of features developed by hardware vendors to provide hardware mechanisms to simplify the use of virtualisation. It targets non-virtualisable instructions (privileged instructions) and includes a new CPU feature that allows the hypervisor to run in a new root mode below ring 0, sometimes referred as ring -1 [45, 46, 145]. Thus privileged and sensitive calls are set to automatically trap to the hypervisor, eliminating the need for either binary translation or paravirtualization (figure 2.8). Examples of this technology include Intel Virtualization Technology (VT-x) and AMDs AMD-V [140].

2.3.2.7 Hypervisor

The hypervisor, also known as virtual machine monitor or control program [56, 115], is the virtualised environment component that manages virtual machines and controls the access to the system resources. Considering an abstract view, the hypervisor is inserted between the virtual machines and the hardware (figure 2.3).

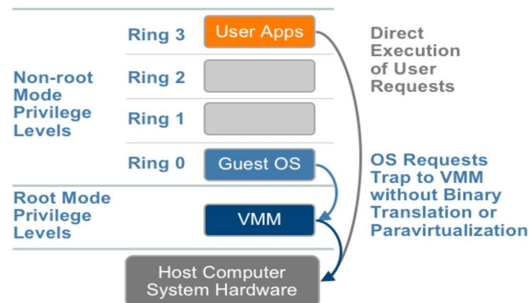

Figure 2.8: **Hardware-assisted virtualisation (figure from [140]).** It provides hardware mechanisms to simplify the use of virtualisation, targeting non-virtualisable instructions and includes a new CPU feature that allows the hypervisor to run in a new root mode below ring 0. It needs neither binary translation nor paravirtualization.

Hypervisors can classified in one of three types [116, 118]:

- **Bare-metal hypervisor (type 1).** It executes on top of the hardware platform. It is a kind of thin operating system and controls the hardware, handles resource scheduling and access and monitors virtual machines. It is normally the choice when performance is a strong requirement. Examples of type 1 hypervisor are: VMware ESX, Citrix XenServer and Microsoft Hyper-V.

- **Hosted hypervisor (type 2).** It executes on top of an operating system environment, as a process. System resources are presented by the underlying host operating system. Examples of type 2 hypervisors are: Parallels workstation, Microsoft virtual server, QEMU, VMware server and VMware workstation.

- **Hybrid hypervisor.** It has a control structure running directly on hardware and employs a high privileged virtual machine to management functions (VM_0 in figure 2.3). Example of management function is the access to device drivers. In a hybrid hypervisor, device drivers are located in the management virtual machine and it can have access to the hardware without interference of the hypervisor 2.3. Examples of hybrid hypervisors are: Xen hypervisor and Microsoft's Viridian.

2.3.2.8 The Xen Hypervisor

Xen is a hypervisor implementation, based on paravirtualization (section 2.3.2.6). However, paravirtualization implemented in Xen is different from other one implemented in other works, since Xen was designed to deal with popular and standard applications and services, overcoming limitations present in other works, as in Denali Project [8], which among other things does not support x86 segmentation [18].

Operating systems running as guests in any Xen virtual machine need to be modified to run over Xen [28]. Although this can be a drawback at first glance, Xen

achieves high-performance and strong resource isolation because of those modifications required by their guest operating systems [137, 18].

In Xen context, virtual machines are called domains. So, Xen has a domain 0 (dom0) and one or more unprivileged domains (domU). The dom0 contains resources necessary to manage Xen and the other domains and also possesses privileged access to the underlying hardware. Then, whenever an unprivileged domain needs to access the hardware resource, this is made by means of domain 0 (figure 2.9) [28, 59]. It also contributes to minimise the hypervisor size. Thus, general speaking, dom0 manages accesses to disks, network and other devices and the hypervisor manages accesses to the CPU, memory and handle interruptions [137]. Xen interacts with domains by means of hypercalls.

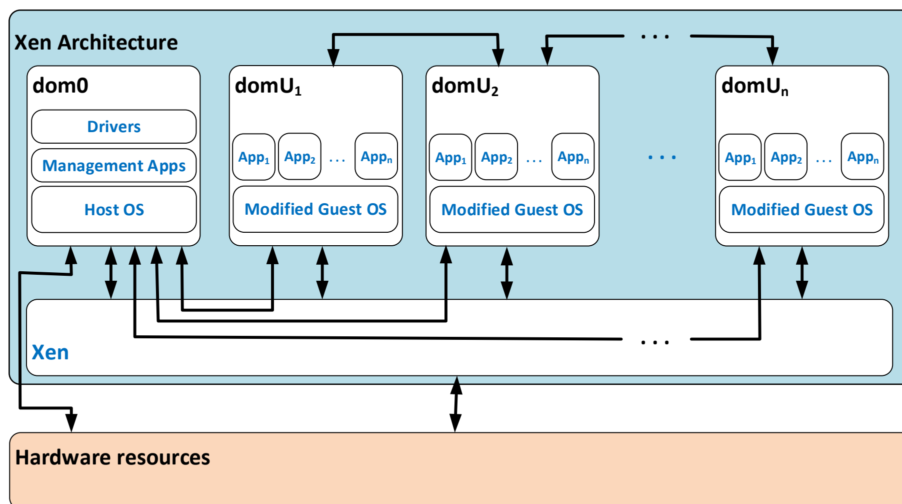

Figure 2.9: **Xen Architecture.** In Xen architecture virtual machines containers are called domains. So, Xen comprises of a privileged domain 0, to manage hypervisor and the other domains, and one or more unprivileged domains. This arrangement contributes to minimise the hypervisor.

2.3.3 Chipset: Intel x86 Architecture

Chipset is an important concept in the computational system. According to [57], a chipset is a coordinated specific unit comprising of CPU, Memory Controller Hub (MCH), also known as Northbridge and Input/Output Controller Hub (ICH), also known as Southbridge. This is a simplification of a chipset architecture since chipsets have other components, including special purpose chips and circuitry [85]. It is noteworthy that this arrangement does not mean a hierarchy: there is not a chip in the chipset to control the CPU and the CPU does not control all the chipset components. Instead, the chipset components work in a coordinated way.

However, the Intel x86 architecture has evolved since the introduction of SMM into Intel 386SL processor. Then, contemporary chipsets have an architecture comprising of more components than CPU, MCH and ICH, which are organised to deal

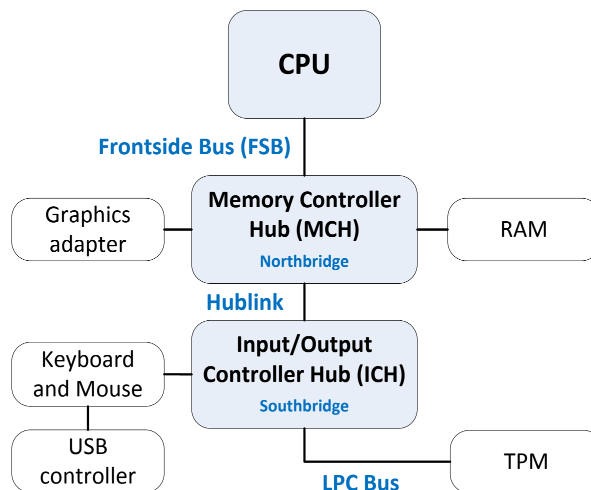

Figure 2.10: **Intel Hub Architecture**, based on [57]. The key components in the Hub Architecture, which are generally present in machines based in Intel processors released up to 2008. According to [57], LPC is the Intel's choice to connect ICH with TPM chips, when a TPM chip is in use.

with issues related to the previous platform. Such issues are described in section 2.3.3.2.

Thus, this section explains the Intel Hub Architecture (1999 - 2008) and the Platform Controller Hub (from 2008 on), presenting the components of both architectures, considering the interaction with SMM. Also, this section explains the processor (CPU) operating modes and the security issues related to them.

2.3.3.1 Intel Hub Architecture

The basic components of Intel Hub Architecture are presented in figure 2.10 in a simplified block diagram. It introduces the interactions of the main components of that platform.

The CPU is responsible for performing the main tasks of processing in the system. By CPU, we mean one or more processor cores. The Memory Controller Hub (MCH), also known as Northbridge, controls the CPU access to the RAM. The I/O Controller Hub (ICH), also known as Southbridge, connects the input/output devices to the system.

Three buses are present in figure 2.10: The Front Side Bus (FSB), the Hublink and Low Pin Count (LPC). The FSB is the way to connect CPU and the MCH. In turn, the Hublink connects MCH and ICH. Both are high speed buses. The LPC connects low-bandwidth devices to the ICH and it is a slow bus. According to [57] the LPC is the Intel's choice to connect ICH with Trusted Platform Module (TPM), when TPM is in use.

TPM is the main component of the trusted platform proposed by the Trusted Computing Group (TCG) [138] and it is generally deployed in a chip. By defini-

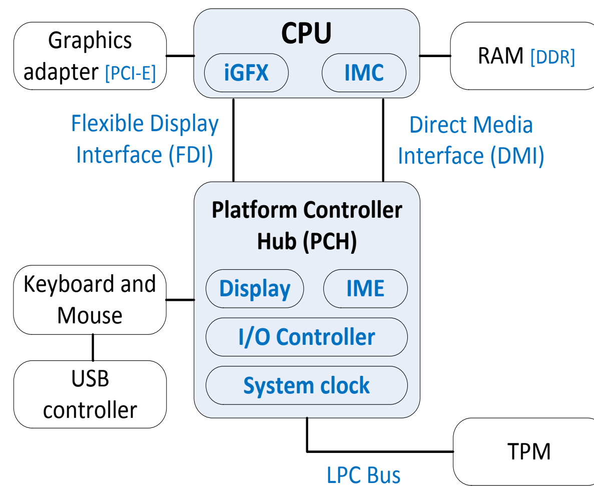

Figure 2.11: **The Intel Platform Controller Hub.** This platform is basically a rearrangement of functions and components from the Hub Architecture. However, improved components where deployed.

tion, a trusted platform is the one that can be trusted by a user of this platform. This user can be any entity local or remote. The trusted platform should be able to prove its identity and its security an entity which challenges the trusted platform. This process requires some attestation process, normally using cryptographic algorithms and protocols [42, 57, 138].

2.3.3.2 Intel Platform Controller Architecture (PCH)

Intel had released in 2008 the Platform Controller Hub (PCH) (Figure 2.11) [81, 82, 89], which was introduced in the Intel 5 Series processors and supersedes the Intel Hub Architecture. Observing figure 2.11, there is a simple block diagram with two main components: CPU and PCH.

Comparing figures 2.10 and 2.11, they are about the same, but some rearrangement of functions and components were done. The motivation behind that improvement was the bottleneck between the CPU and the MCH, caused by the FSB bandwidth limitation. To avoid FSB bottleneck, in the new architecture the MCH components were distributed between the CPU and the PCH blocks and the ICH components were embodied into PCH block[89].

The CPU embodies the Intel Graphics Accelerator (iGFX) and the Intel Memory Controller (IMC) components, which are improved versions from the previous MCH video and memory management components.

The CPU and PCH are connected by means of the Flexible Display Interface (FDI) and the Direct Media Interface (DMI). The FDI connects the graphic component (the engine) in the CPU and the graphic component (the interface) in the PCH [81] as shown in figure 2.11. A detailed discussion about FDI and other display technologies used in Intel's platforms can be seen in [51]. The DMI is a high

Table 2.1: **Intel chipsets.** Those are the main Intel chipsets produced, since the release of PCH architecture. Among those main chipsets, other one were produced and released [89].

Codename	Series	Released	Processor
Ibex Peak	Intel 5	2008	Nehalem
Cougar Point	Intel 6	2011	Sandy Bridge
Panther Point	Intel 7	2012	Ivy Bridge
Lynx Point	Intel 8	2013	Haswell
Wildcat Point	Intel 9	2014	Broadwell
Sunrise Point	Intel 100	2015	Skylake
Union Point	Intel 200	2017*	Cannonlake

* To be released.

speed communication interface between CPU and PCH, allowing the newest and legacy software to operate transparently and concurrently. The Intel Management Engine (IME) is a multiple components resource to execute functions of control and management in the platform [81].

Intel chipsets are under ongoing evolution. Table 2.1 exhibits the Intel chipsets models, since the first release of PCH. The table presents the chipset codename, the chipset series, the year of releasing and the main processor (microarchitecture) codename that use the chipset.

It is noteworthy that each chipset has its own variations, as workstation, server, desktop, mobile and so forth. Besides, those chipset codenames are the main ones and among them there are other chipsets released. For example, between the release of Ibex Peak and Cougar Point chipsets, there were also the release of chipsets Langwell (smartphone), Tiger Point (netbook) and Topcliff (embedded system) [89].

Then, table 2.1 exposes partially the complexity associated to low-level security: each chipset has its one particularity that must be taken in account when addressing security.

2.3.3.3 Processor Operating Modes

The CPU operates in five modes: protected mode, real-address mode, virtual-8086 mode, IA-32e mode and system management mode (SMM) [85]. In machines that support Virtual Machine Extensions (VMX), the processor can operate in VMX root operation (privileged and used by the hypervisor) and VMX operation (non-privileged and used by guest VMs) (more on that in section 3.3.4).

The protected mode is based in four rings. Those rings are numbered from zero to three (0 to 3), zero being the most privileged and three the least privileged. Although that protected mode was designed by Intel to reinforce security, there is a difference between the Intel designed ring scheme and what is really implemented for OS vendors [57, 85, 86].

Figure 2.12 (left side) exposes the design proposed by CPU architects [57]. Since ring 0 is the most privileged, only the kernel should be there. Drivers, as essential complementary part to the work of operating system, should stay in ring 1. Operating system services do not need such a privileged status, so they should stay in

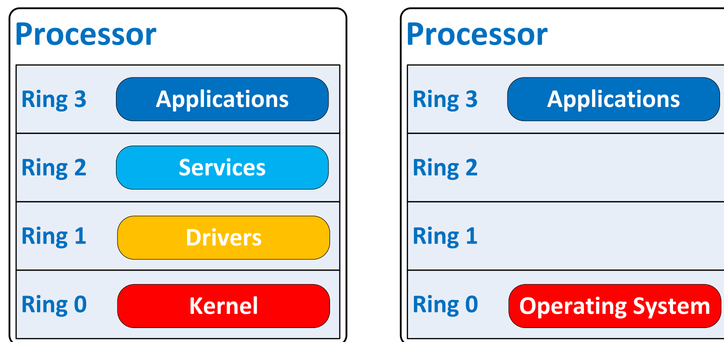

Figure 2.12: **Ring security scheme, based on [57].** The first figure presents the ring scheme design proposed by Intel CPU architects and the second one presents as this scheme is really used by operating systems vendors, according to [57].

ring 2, just a layer below the applications, which should be in the least privileged ring 3.

The ring scheme in figure 2.12 (left side) was designed to provide isolation among components in the system by separating the kernel from the non-fundamental parts of an operating system (or hypervisor). It means, ring 0 should contain just the minimal essential functions to guarantee the correct and safe work of the operating system. However, as exposed in figure 2.12 (right side), operating system vendors just use ring 0 and ring 3.

The consequence from the usage in figure 2.12 (right side) is that operating system components (kernel, drivers and services) have the same privilege, which compromises the designed ring scheme security. For instance, any device driver in the system is granted the ring 0 security level, the same as the kernel, allowing a great avenue of attacks, using those drivers.

Real-address mode offers an Intel 8086 processor 16 bits environment with extensions. Those extensions allow, for instance, its use with other operating modes.

Virtual-8086 mode allows software written to real-address mode being executed in a protected environment. In essence, Virtual-8086 is not a processor mode [85]; it is the protected mode enabled to execute multiples Intel 8086 real-address mode software [24].

IA-32e mode was introduced in 64-bit Architectures. It comprises of two submodes: Compatibility mode and 64-bit mode. Compatibility mode allows most legacy 16-bit and 32-bit software to run without re-compilation under a 64-bit operating system. 64-bit mode permits software written to access 64-bit linear address space to run in 64 bit architectures and with 64-bit operating systems [85].

The SMM will be detailed in Chapter 3.

2.4 Environment and Technologies

This section discusses the environmental issues around the SMM: the components in the host machine competing for processor time, memory space and with concurrent security tasks; by analysing and describing the current state-of-art of technolo-

gies as UEFI, Trusted Computing and Intel Software Guard Extensions (Intel SGX). We call this environmental issue as horizontal security level.

The SMM will be addressed in chapter 3 and Virtual Machine Extensions (VMX) in section 3.3.4.

2.4.1 Coreboot

Coreboot is an open source project which can be an alternative to the proprietary BIOS. It initialises the hardware and executes additional boot logic, called payload, in a fast and secure way on modern computers [36].

We consider in this work the coreboot version 4.4. It has 58 SMI handler implementation, with a average size of 6.57 Kbytes. The smaller size is 743 bytes and the biggest one is 22.62 Kbytes. Those numbers are important for the desing of the $SBST$.

We find a suitable SMI handler is this release for our chipset 2 (section 1.2) whose source code size written in C language is 9.97 Kbytes. Thus, we estimate that its executable code is estimated in 5.5 Kbytes. So, it leave around 25 Kbytes available for our $SBST$ source code.

2.4.2 Unified Extensible Firmware Interface (UEFI)

UEFI is an interface specification providing interaction between personal computer operating systems and platform firmware, which acts in the bootup process and passes control to next component in the system. Normally, this next component is an operating system loader. That specification does not impose the way such a firmware should be built.

It provides a standard environment for secure booting process and for running pre-boot applications. Legacy booting processes, based on BIOS, continue to be supported for platforms using UEFI [2, 44]. UEFI also provides a secure boot. UEFI Secure Boot aims primarily improve security in the pre-boot environment [144]. Another fundamental part of UEFI is the Platform Initialisation (PI) specifications. In fact PI is the part that specifies how to build the components to initialise the platform and it is largely platform dependent [139].

UEFI and PI specifications are under responsibility of The UEFI Forum [3].

Figure 2.13 shows where UEFI and PI specifications fit into the platform boot flow.

2.4.3 Intel Trusted Execution Technology

The Intel Trusted Execution Technology (Intel TXT) provides secured system start (or restart), such that the system executive software can be loaded in a trusted way [57]. In the context of Intel TXT, the operating system or hypervisor are called generically Measured Launched Environment (MLE). Intel TXT uses and extends the Virtual Machine Extensions (VMX) and Safer Mode Extensions (SMX) to enter and exit from the secure environment provided by Intel TXT [57, 58].

SMX instructions are implemented by GETSEC CPU instructions set. Thus, Intel TXT uses the GETSEC instruction SENTER (secure enter) to enter and SEXIT (secure exit) exit from the secure environment. Basically, Intel TXT execution flow

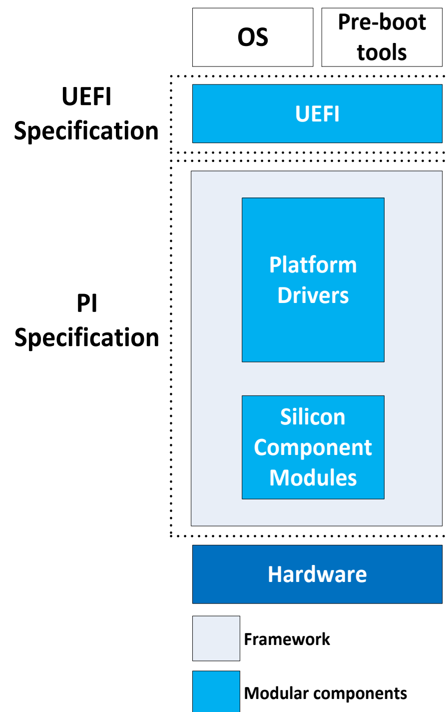

Figure 2.13: **The PI and UEFI layers, based on [153].** The Platform Inicialisation (PI) and the Unified Extensible Firmware Interface (UEFI) together forms an interface between the platform (chipset) firmware and the operating system, providing an environment for secure booting and for running pre-boot applications.

is: it starts using GETSEC [SENTER], load the MLE and pass control to the MLE; by finishing the execution, it removes the MLE using GETSEC [SEXIT] [57].

Intel TXT uses Trusted Boot (tboot) [95] to launch the MLE and it takes in account the use of TPM to perform and store the measurements and verifications [57, 58].

The Intel TXT ability of issuing a SENTER instruction at any time is called **late launch** [57].

2.4.4 Trusted Computing

The notion of trust can vary depends on the person using it. Even when talking about trusted computing, this notion can vary, for example, depending on the kind of resources that are in use and whether the stakes are high or not. Besides, define trust can be a hard work. Then, we chose to follow the definition proposed by the Trusted Computing Group (TCG) [57, 138]: "An entity can be trusted if it always behaves in the expected manner for the intended purpose".

It noteworthy the fundamental work on trust computing by Bell and LaPadula [20] which proposed a mathematical model for security systems. A comprehensive work on trusted computer systems, criteria for evaluation and how to use such a kind of systems can be found in the Orange Book [42].

2.4.4.1 Trusted Platform

By definition, a trusted platform is the one that can be trusted by a user of such a platform. This user can be any entity local or remote. The trusted platform should be able to prove its identity and that it is safe to an entity which challenges the platform. This task requires some attestation process, normally using cryptographic algorithms and protocols. In the context of TCG, the platform takes in account the use of a Trusted Platform Module (TPM) and three so-called roots of trust to provide that attestation and other security services.

The TCG established three roots of trust [57]:

- **Root of Trust for Measurement (RTM).** It provides an entity, which is implicitly trusted, with the ability of making reliable integrity measurements. An RTM can be static or dynamic. The static RTM (SRTM) on PCs is a piece of code embedded into the BIOS and is normally started on the bootup process. That piece of code is called Core Root of Trust for Measurement (CRTM). The dynamic RTM (DRTM) is started whenever the platform needs to be measured for an event that is not a platform reset. The task of measurement is conducted by the platform CPU.

- **Root of Trust for Reporting.** It provides an entity, which is implicitly trusted, with the ability to report information about the platform to local or remote entity. This information can be verifiable by the attestation process. This process is conducted by the TPM.

- **Root of Trust for Storage.** It provides an entity, which is implicitly trusted, with the ability to store critical information in way that this information cannot be tamper or leaked. Some examples of that critical information are the integrity measurements and cryptographic keys used in the attestation process. This process is conducted by the TPM.

2.4.4.2 Trust Computing Base

The Trusted Computing Base (TCB) is all platform components, technologies and services that an entity needs to trust in, so that platform can be considered trusted. The TCB is responsible for the correct functionality and security of a computing platform [42, 54, 57, 138].

2.4.4.3 Trusted Platform Module (TPM)

The TPM is the main component of the trusted platform proposed by the Trusted Computing Group. It is generally deployed in a chip. Figure 2.14 presents the main components of the TPM. Those components enable the roots of trust.

In this work, we are considering TPM v1.2, where applicable. The new standard (TPM v2.0) was released in 2013 with better support for virtualisation and more flexibility in cryptographic algorithms.

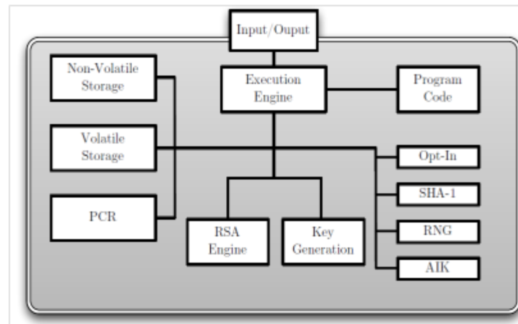

Figure 2.14: **TPM Overview (figure from [54]) [57]**. The Trusted Platform Module (TPM) based on the version 1.2 specification.

2.5 Data Integrity with Hash Functions

Data integrity is a cryptographic service that can be provided by the cryptographic mechanism of Hash Function. Hash functions take some data and compute a "fingerprint" of such a data [134]. That fingerprint can be also called: message digest or hash code or hash value [132].

We can define a hash function as a three-tuple (X, Y, H), where the conditions below are satisfied (based on [134]):

1. $X = x_1, x_2, ..., x_n$ is a set of possible messages (or data).
2. $Y = y_1, y_2, ..., y_n$ is a finite set of possible message digests.
3. $H = h_1, h_2, ..., h_n$ is a finite set of possible hash functions.
4. There is a hash function $h_i \in H$, where $h_i : X \to Y$ such that $y_i = h_i(x_i)$

Hash functions are used in insecure environments where we can protect Y, but not X. Thus, to use it for data integrity, we compute a hash code $h_i(x_i) = y_i$, with $x_i \in X$ and $y_i \in Y$, and protecting y_i. If x_i has changed to x_i', we can compute the hash code again $h_i(x_i') = y_i'$ and show that $y_i' \neq y_i$. So, considering that the hash function is secure and having y_i safely stored, we can always compute the hash code again and compare the stored one with the fresh computed hash value to check the integrity of x_i [134].

The hash code or fingerprint or message digest is called this way because it generates a fix length of y_i. Thus, x_i has variable size and y_i is a fix length and, in general, $x_i > y_i$ [132].

Some examples of hash functions are MD5 ($y_i = 128-bit$) [117], SHA ($y_i = 160-bit$) [49], SHA-256 ($y_i = 256-bit$), SHA-384 ($y_i = 384-bit$), SHA-512 ($y_i = 512-bit$) [50], Whirlpool ($y_i = 512 - bit$) [19].

The security of hash functions are in general associated to three features: preimage resistant, second preimage resistant and collision resistant [132, 134]. For more details about those features and about hash functions, see [113, 123, 132, 134].

2.6 Related work: System Executive Software Integrity Issues

This research is investigating the issues around using SMM resources to build a security tool. It is intending to propose a general architecture where any security tool can fit. However, this research aims built a proof of concept to validate the proposed architecture. This proof of concept is tackling the issue of measuring the integrity of a system executive software, more specifically the Xen hypervisor. Thus, this section investigates some of the main works published about measuring the system executive software integrity. In chapter 3, section 3.4 are discussed security tools capitalising on SMM.

Integrity measurement is the process where some entity can obtain information from a platform, based on a metric or in an algorithm [57]. Another important related concept is re-measurement, which is the process of determining if a previous measurement is still the same or not [57]. Therefore, an integrity measurement tool must be able to measure and re-measure its target and spot possible differences. So, the aim of system executive software integrity measurement is to verify that its code and data remains the same since the last measurement.

It is noteworthy that sometimes defending the integrity of the system executive software might mean eliminate the cause of problem or even the measurement target or reduce it.

2.6.1 Integrity Measurement

Bringing those concepts on Integrity Measurement to hypervisor security context, we can state that the aim of integrity measurement on hypervisors is to guarantee that the hypervisor, its code and data, remains the same since the last measurement.

Some research in that direction has been successfully presented to measure the integrity of the hypervisor, its components and related entities in the virtualised environment. Those approaches may add extra code to the hypervisor or to the kernel.

IMA [121] is an integrity measurement architecture used to measure executable code in Linux operating systems on load-time, capitalising on the use of the TPM and following the TCG standards. Using IMA measurement agents and the services provided by the TPM, a remote agent can attest the integrity of a target platform. However, measurement during the load-time does not accurately reflect runtime behaviour, if someone is considering load-time measurements alone.

Since measurement during the load-time does not accurately reflect runtime behaviour, if one is considering load-time measurements alone, **PRIMA** [99] is an extension of IMA and was built on IMA code, but takes a new approach to the problem by basing the measurements on information flow integrity. However, both IMA and PRIMA are vulnerable to attacks that have the kernel as the target. Isolation is a fundamental security principle for an integrity measurement agent to protected itself against being tampered by malicious code. Besides, from their architecture, one can note that they lack good isolation.

Since the virtualised environment has become more prevalent and since virtualisation, by definition, needs to offer strong isolation, new approaches showed up

to take advantage from this characteristic. For example, **Terra** [52] provides strong isolation by means of a trusted virtual machine monitor (TVMM), a trusted hypervisor, which offers an open-box VM, for VMs with low security requirements, and a close-box VM, for VMs with high security requirements. Terra provides integrity measurement for the VM, but not runtime measurement. Moreover, although it implements an trusted hypervisor, there are no measurements for the integrity of TVMM itself.

By trying to reinforce the security in virtualised environment, an architecture called **vTPM** [21] was designed, which virtualises the TPM (more on TPM in section 2.4.4.1), allowing each VM access to its own vTPM, as a physical or unique TPM. Although it can reinforce security by making use of the TPM resources inside of a VM, the VM itself continues to be vulnerable to many others classes of attack, especially attacks that tamper with the hypervisor.

In a different direction, some approaches try to identify rootkits or malwares [130] in systems using hypervisors. That is the case of **Patagonix** [107] that aims to identify covertly executing binaries. Patagonix guarantees that any executing binaries will be reported to the system administrator. In fact, it realises its objective by measuring the integrity of those binaries. Its architecture comprises of a Patogonix VM, which plays the main role in the system, a component inserted in the hypervisor. That component eliminates the semantic gap [27] between the monitored VM and the hypervisor. However, we believe that the main drawback of Patagonixmight be its strong assumption that the hypervisor will provide self-protection and it will defend itself against attacks from rootkits and malwares, while also defending Patagonix from such attacks. It noteworthy to mention **Subvirt** [101] that conversely uses a hypervisor and virtual machines to implement a new class of malware called virtual-machine based rootkit (VMBR).

HIMA [13] is an integrity measurement agent residing in the hypervisor and with a part of its architecture inserted in the management VM (Dom0), since HIMA is implemented over Xen Hypervisor. It aims to measure the integrity of VMs. It offers strong isolation and a solution for the problem of 'Time of Check to Time of Use' (TOCTTOU). That solution is basically a re-measurement, as described by [57]. HIMA is to some extent an evolution of the previous work of IMA and PRIMA, but which is suitable for the virtualised environment and which capitalise on the privileges of hypervisors. However, like Patagonix, HIMA assumes a trusted hypervisor and platform.

HyperSafe [141] is a lightweight approach that provides lifetime control-flow integrity for hypervisors. Specifically, it protects the hypervisor's code and static data from being compromised, even if memory bugs are present. However, HyperSafe just consider attacks that exploit vulnerabilities in the hypervisor's code. Attacks that come from low level, such as cache exploitation ("Cache poisoning") [148], SMM exploitation [47, 45], Intel TXT attack [147] or hardware attacks, can compromise the security of HyperSafe.

Those aforementioned approaches insert extra code in the hypervisor, enlarging the attack surface.

2.6.2 Minimising the Attack Surface

Conversely, other approaches minimise the hypervisor in order to diminish the attack surface, leaving just the essential functionality in the hypervisor. For example, **SecVisor** [124] is a tiny hypervisor that ensures that only approved code can execute in kernel mode, guaranteeing code integrity (kernel integrity). **TrustVisor** [109] is a minimised hypervisor that provides code and data integrity, with a small code base. **BitVisor** [128] is a thin hypervisor, introducing and implementing *parapass-through* drivers, which allow remove device drivers from the hypervisors, since device drivers are more prone to errors than other hypervisor components and degrade the general level of security [29]. The main limitation of those works is the fact that they just support one single VM and they are not able to run over a multi-core or multi-CPU platform. **NOVA** [133] minimises the amount of hypervisor code and reduces the TCB of VMs, by moving most functionality to user level. Different from the last approaches, NOVA supports multi-core and multi-tenancy.

2.6.3 Hardening the Hypervisor

The most intuitive approach is hardening the hypervisor. For example, by improving the security of VMs allowing them to run in potentially insecure management OSs [26]. A management OS, According to [26], is the OS in the management VM in a type 1 hypervisor environment (as in Xen hypervisor). **Hyperwall** [136] assumes that a hypervisor is malicious and provides protection to its guest VMs, by proposing and using Confidentiality and Integrity Protection tables (CIP tables) and hardware mechanisms. **sHype** [120] inserts a trusted hypervisor layer below the Guest OS to control the sharing of virtual resources based on security policies.

2.6.4 No Hypervisor Approach

The 'no hypervisor' strategy proposes a new approach to tackle the security issues of hypervisors: rather than defending, just remove the attack surface by getting rid of the hypervisor, but preserving the semantics of virtualisation [100, 135]. NoHype identifies the main roles of a hypervisor and searches for some other manner to do the same thing, in order to eliminate the hypervisor. It is a feasible system that can be implemented on commodity hardware and is focused on cloud computing infrastructure. However, NoHype does not mitigate all threats that a hypervisor is prone to in cloud architecture, and can introduce new kinds of threats [39]. Besides, it has drawbacks that can hinder its use in a real world scenario. For instance, it compromises scalability, which is one the stronger appeals to use cloud computing [38].

2.6.5 Intrusion Detection System on Virtualisation

Other approaches try monitoring the virtual environment to detect intrusion and the presence of malicious code. In this sense, a technique called Virtual Machine Introspection (VMI) has been used to monitor the VM and for building intrusion detection systems in virtualised environments [17, 53]. For example, **Livewire** [53] allows IDS to monitor the state of a guest OS, but it was not built for taking in account distributed systems. On the other hand, **HyperSpector** [103] is a virtual dis-

tributed monitoring environment to allow secure intrusion detection in distributed computer systems, using virtual machines and virtual network, performing isolated monitoring without any additional hardware. Those techniques use measurements similar as integrity measurement. However, they do measurements just to detect intrusions and prevent attacks on the system and evasions from the IDS. Both Livewire and HyperSpector capitalise on the use of virtualisation to secure the IDS, but not necessarily to improve the security in virtualised environments. For this purpose, the approach used in [110] does a better job. By capitalising on multiprocessing environment, they employ multiple observers to perform measurements on the system, including self-measurement. We can define an observer as an agent following our aforementioned definition for agents (see definition 2.3). Each observer is instantiated in a thread, and then multiple observers are employed to perform the same measurement, using multiple communication channels to convey their findings. Those findings are then compared to look for inconsistencies. Since, this measurement is also performed over the observers themselves; it creates an additional self-defence mechanism. The difficulty to attack such a system is related to the number of observers. Let n be the numbers of observers. Since all of them are performing the same measurement and communicating their findings to each other, an attacker needs to subvert $n(n-1)$ individual communication channels to be successful. That paper [110] presents insights about formalisation and defence and self-defence mechanisms for our work.

2.7 Discussion

This chapter exposed the complexity of the investigation performed in this work. To dominate that complexity, this chapter has approached the security problem considering a vertical and horizontal security level and describing the components of each one.

We can observe that the technology developed to offer more computing power and facilitate the use of computational resources have introduced new threats. That is the case o cloud computing and virtualisation. Those technologies enabled users to increase their computing power capacities and reduce their costs, but made them vulnerable to news classes of attack, as those one perpetrated by malicious virtual machines. To certain extent, those technologies create a single failure point: the hypervisor, which became a main target for attacks. In fact, successful attacks have been performed against hypervisors, both commercial hypervisors[102] and open source ones[145].

Because of the importance of hypervisors, a great deal of research has been done in securing them. In general, those works approaches lack of: isolation, good view of the system and enough privileges to protect the hypervisor. Also, they address the problem of measure the hypervisor integrity, but do not take action when a violation is identified. So, the protection of hypervisors comprises of just detecting an integrity violation, in most of the cases. We observed that in some cases defend the hypervisor integrity means eliminate the cause of problem or even the measurement target, as the own hypervisor. That is the case of NoHype [135]. However, it eliminates some security issues, but it introduces other security issues and drawbacks in the cloud environment, as described in [38].

A main threat against hypervisors and potentially all the cloud comes from the chipset. The cloud and the virtualised layer are sort of the big picture in the computational security. However, in the single machine, the chipset is yet the platform from where attackers can lunch the more devastating atttacks and take over of the more powerful resources. For example the SMM, a resourceful component of the chipset, which can enable isolation and high privileges for security tools, can also be used for deploying resourceful SMM-based malwares, as described in section 3.5.

The chipset is specially a complex component in this scenario. Different versions of them are around and some part of the chipsets, as firmwares and device drivers, are OEM specific, which tangling even more the relation among components.

2.8 Summary

This chapter discussed the security context whereupon this research is investigating. We determined a set of definitions required to formulate and design the answers for the research questions and then we established a vertical and a horizontal security level to deal with the complexity of the security problem we are addressing. Then, the components of those security levels were presented and discussed. By investigating the chipset, we found the complexity related to address security concerns related to chipsets, since there are many different versions of it installed into contemporary machines. Finally, we investigated and discussed the main works addressing issues around measurement and protection of the system executive software, since this research aims to build a proof of concept to measure the integrity of a Xen hypervisor.

Our analysis of the main works on measure the integrity of the system executive software identified that in general works lack of: good isolation, good view of the system and enough privileges to address that problem. As we stated in chapter 1 those capabilities are fundamental in such a security context. So, this motivates the search for those capabilities by using SMM resources.

Thus, in the next chapter we will deeply investigate the system management mode, presenting its components, investigates the SMM related registers, exploring the registers directly and indirectly related to SMM, investigating SMM operation and some of its relationship with components in the system and other Intel technologies. The chapter also presents the motivations to use the SMM resources and security tools capitalising on SMM, discussing issues around those implementations. On the other hand, it describes the attacks using SMM as a platform and attacks against the SMM itself, offering ways to thwart such attacks.

Chapter 3

The System Management Mode (SMM)

Sorry to be a wet blanket. Writing a description for bitcoin for general audiences is bloody hard. There's nothing to relate it to.

''Satoshi Nakamoto"

SMM is one of the operating modes that a CPU can operate in (section 2.3.3.3). Whenever the processor is in SMM, a set of high privileged resources is available to its executive software (called SMI handler) to perform priority management functions. In general, code artefacts trying to capitalising on those resources must emulate the SMM executive software. So, SMM provides privileged and isolated environment where priority system code can perform and sensitive related data is stored. Those codes and data are related to system management and hardware functions. The implementation of SMM components are platform, chipset and Original Equipment Manufacturer (OEM) specific. It means that, even among Intel processors, a processor family may have different SMM components when compared with another family and an OEM is free to implement those components in its own way, provided that the components work as specified by Intel [87].

SMM and its resources were designed and are implemented to be use by system firmware (generally OEM code), not by "general-purpose" systems software [87]. For example, an SMM-based security tool is a use case of "general-purpose" software. So, any use different of the original design and purpose is considered a workaround and must be carefully planned to avoid or minimise side effects. That recommendation aims to preserve the important SMM functions to avoid that a "general-purpose" software violates the strict limits and constraints of SMM resources and disturb the correct function of SMM components, as by overwriting the SMI Handler (details in sections 3.2.4 and 3.4). So, this research establishes a set of requirements, which a security tool must met to overcome those limits and constraints, allowing the use of SMM for "Security-purpose".

3.1 Introduction

Chapter 2 has contextualised and positioned the problem addressed in this research, considering the vertical and horizontal security level in computational systems and specified a set of definitions required to formulate and design the answers

for the research questions proposed. As exposed, there are many security implications related to SMM and system components and technologies, in both vertical and horizontal levels, as malicious virtual machines, oversized hypervisors and non-flushed cache memories. Also, most of the found works addressing issues on system executive software measurement and protection lack one or more of: good isolation, privilege and good view; to accomplish their security task.

This chapter discusses the SMM components, explaining how each one contributes to the SMM operations. It takes in account the evolution of SMM according to the chipsets considered in this research (section 1.2) [66, 71, 87, 97]. It also investigates the SMM related registers, exploring the registers directly related to SMM, as the System Management RAM Control (**SMRAMC**), an 8-bit register working as an access control mechanism to the SMRAM, and the register indirectly related to SMM, as the Advanced Configuration and Power Interface (ACPI) Base Address Register (**PMBASE**), a 32-bit register setting the base address for ACPI I/O registers and other ones, including the SMM register **SMI_EN** and **SMI_STS**. After, it investigates the operation of SMM, its relationship with main components in the system, as cache memory, and relationship with other Intel technologies, as Intel Virtual Machine Extensions (**Intel VMX**) and Intel Software Guard Extensions (**Intel SGX**). The chapter also presents security tools capitalising on SMM and discusses the issues around those implementations. On the other hand, it describes attacks to SMM and offers procedures to thwart such attacks.

The chapter is organised in four content sections and a discussion and summary section. The first one investigates and describes the SMM components. The second section investigates SMM operation, relations and associated issues. Third section deals with SMM security tools pointing out opportunities for improving them and section four discusses SMM attacks and ways to thwart them.

3.2 Components

The System Management Mode comprises of a processor operating mode (SMM), a memory space (SMRAM), a set of registers (SMBASE, SMRAM control register, PMBASE, SMI_EN, SMI_STS and other), an executive software (SMI handler), an interruption (SMI), processor pins (SMI# and SMIACT#) and a processor instruction (RSM). All those components are spread over the chipset (figure 3.1). The rationale behind including, for example, an SMI pin as a component of SMM is that such a component could not exist if SMM does not exist. Some of those components are discussed ahead and other ones, as the system management interrupt (SMI), are addressed in together with other components.

3.2.1 The mode System Management

In SMM, the processor changes the system to an alternate environment, that has its own and separated memory space (SMRAM). Such an environment is transparent and inaccessible for any other code in the system, even high privileged ones as code executing in ring 0, as operating system kernels. The BIOS contains SMM related code and data, loading them into the SMRAM during the boot up process [57] and, since the SMRAM content is volatile, it needs to be loaded every time the system

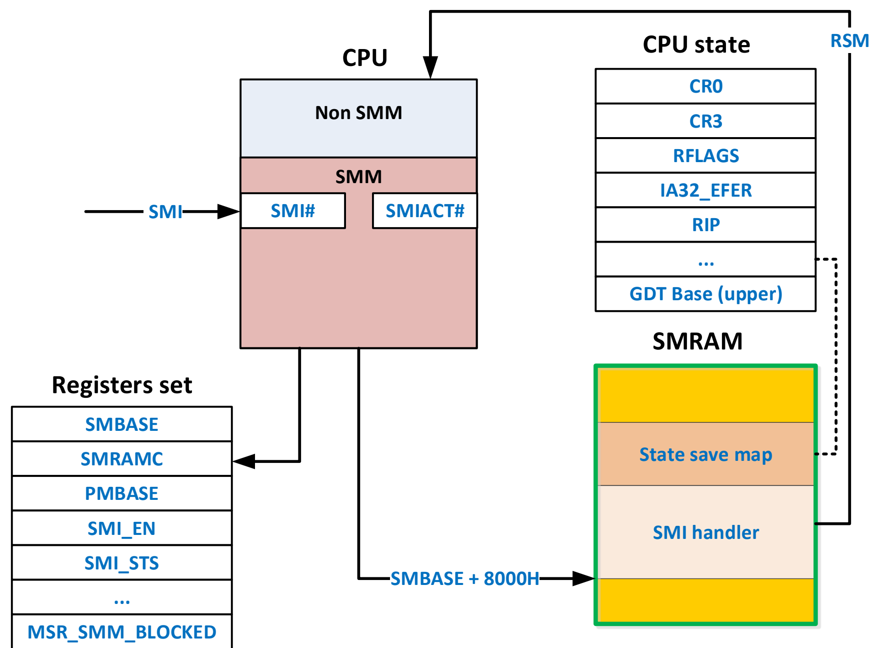

Figure 3.1: **SMM components.** Those components are available when the processor enters in SMM: A processor operating mode, a memory space, a set of registers, an executive software, an interruption, processor pins and a processor instruction.

start (or restart). Because of so powerful resources, some authors refer to SMM as ring -2 (ring -1 would be hardware hypervisors) [148].

3.2.2 System Management Memory (SMRAM)

SMRAM is the work memory used when the processor enters into the SMM. Upon SMM, paging is disabled. Then, all memory access is mapped to the low 4 GBytes of the processor's physical address space.

Figure 3.2 exhibit the SMRAM scheme for the Intel 32-bit architecture. There are basically two areas: SMI handler area, which contains SMI handler code and data, and the state save area, which contains the processor context (or processor state) just before an SMI is triggered. The **SMBASE** register contains the base address for the SMRAM (section 3.2.3.1). The default SMRAM size is 64 KBytes, but it can range from 32 KBytes to 4 GBytes. Figure 3.2 shows that the minimum 32 KBytes (FFFFH - 8000H = 7FFFH = 32768) refers to the SMI handler area (from

SMBASE + 8000H to SMBASE + 8000H + 7E00H) plus the state save area (from SMBASE + 8000H + 7E00H to SMBASE + 8000H + 7FFFH). Then, the SMI handler area is equal to 32,256 bytes (7E00H = 32,256) and the state save area is equal to 512 bytes (7FFFH - 7E00H = 01FF = 512). The beginning of SMRAM is indicated by the register SMBASE, which default value is 30000H, after boot up process or hardware reset [66, 71, 87, 97].

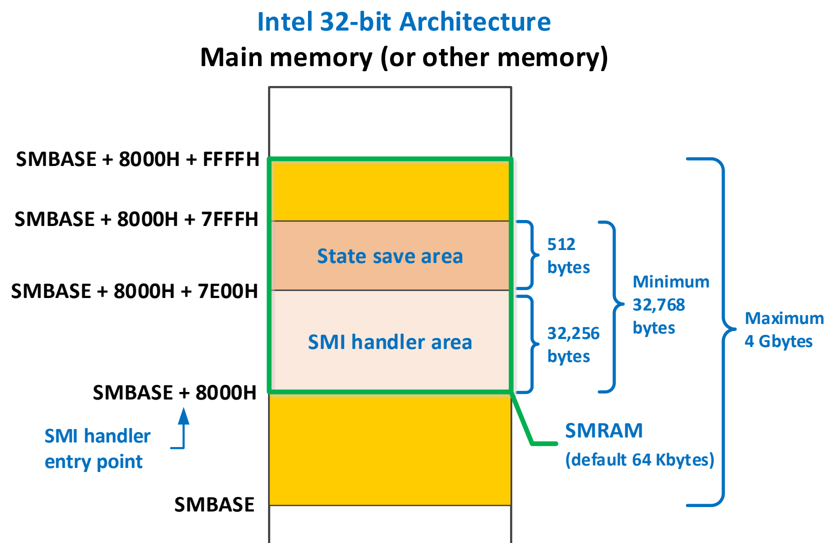

Figure 3.2: **SMRAM space for 32-bit machines.** The SMRAM space can be located in any part of the chipset, as specified by Intel [66, 71, 87, 97]. It is the OEM's choice where to put it. It can lay on the main memory or in other memory. The processor always looks for the SMI handler first instruction at SMBASE + 8000H address in the SMRAM.

Figure 3.3 exhibit the SMRAM scheme for the Intel 64-bit architecture. The difference between the 32-bit and 64-bit architecture is basically the size of the state saved map, which is equal to 512 bytes (7FFFH - 7E00H = 01FF = 512) in the 32-bit architecture and it is equal to 1024 bytes (7FFFH - 7C00H = 03FF = 1024) in the 64-bit architecture and [87, 97].

By entering in SMM, the processor looks for the first instruction at the address SMBASE + 8000H (by default 38000H), using registers CS = 3000H and EIP = 8000H. The CS register value (3000H) is due to the use of real mode memory addresses by the processor when in SMM. In this case, the CS is internally appended with 0H on its rightmost end [24, 87]. The SMI handler can change the value of SMBASE by altering that value at the address SMBASE + 8000H + 7EF8H in the state save map (Intel 64-bit Architecture), but this change would have practical effect just after the processor enters in SMM again, which is known as SMBASE relocation [87, 97]. Many works cite that relocation is done to A0000H [69, 70, 45, 23, 47].

The "relocation" indicates that some registers values stored in the SMRAM state

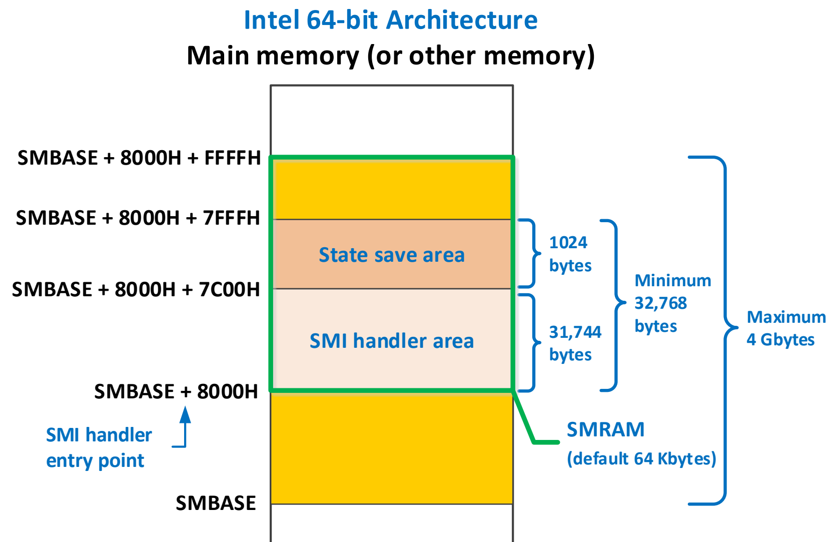

Figure 3.3: **SMRAM space for 64-bit machines.** For the 64-bit architecture, the SMRAM space can be located in any part of the chipset, as specified by Intel [87, 97]. As in 32-bit architecture, it can lay on the main memory or in other memory. However, the state save area is bigger than in 32-bit architecture, since there are more registers and tables to be saved. The processor always look for the SMI handler first instruction at SMBASE + 8000H address in the SMRAM.

save map are changeable, including the SMBASE value. Basically, the state save area stores the value of registers in addresses (or positions) in the SMRAM, so the SMI handler can alter the registers values by writing to those addresses. Then, the processor will operate with the new values when exiting from SMM. However, some values are read-only and change them would cause a processor unpredictable behaviour [87, 97]. For example, for Intel 64-bit architecture: register CR0 is stored at SMBASE + 8000H + 7FF8H and may not be altered; register RFLAGS is stored at SMBASE + 8000H + 7FE8H and may be altered; register RIP is stored at SMBASE + 8000H + 7FD8H and may be altered; and register RAX is stored at SMBASE + 8000H + 7F5C8H and may be altered [87, 97].

SMRAM can receive information other than the processor state or SMI handler code and data. However, any code and data handling should be performed by the SMI handler. This capability is useful to use SMM for “general-purposes” or for our objective for “security-purposes”. About using SMM for “general-purposes” see details on the opening of this chapter and in contribution 4.

3.2.3 SMM Related Registers

There are several registers related to the SMM in the chipset. This section investigates and describes those registers and clarify issues about their functions and

values. This section is based on our target chipsets, described in section 1.2. More on SMM register can be find at (appendix A).

In general, registers with a suffix "EN" are used to enable functions and devices, registers with a suffix "STS" are used to report the status of functions and devices and registers with a suffix "CNT" are used to control functions and devices in the system.

The Model-Specific Registers (MSR) can be read with the instruction RDMSR and written with the instruction WRMSR. They are supported by a finite number of families and/or models of Intel processors. The instruction CPUID is used to verify the available of MSR for any families and/or models of Intel processors.

3.2.3.1 SMBASE register

This register contains the base address for the SMRAM. It is a internal processor register and contains the default value of 30000H and it is can be relocated to A0000H [45, 69, 70, 92, 93, 66, 71].

The next sections describe the registers related to SMM in our two target chipsets.

3.2.3.2 Chipset 1 System Management RAM Control

Probably the more important register to the SMM is System Management RAM Control (**SMRAMC**) [69, 70, 92, 93]. This 8-bit register is a sort of access control mechanism to the SMRAM (figure 3.4). In our target chipset 2, it is located at PCI device 0, address offset 88H [92, 93].

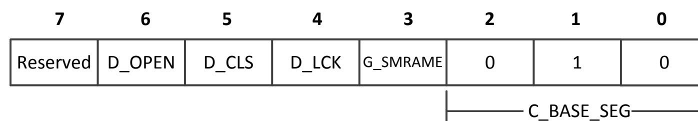

Figure 3.4: **SMRAM control register.** It acts as access control to the SMRAM. The bit D_OPEN indicates if the SMRAM space is accessible or not. The bit D_CLS indicates if the SMRAM is accessible for data references. Bit D_LCK blocks or unblocks writing access to the SMRAMC register. Bit G_SMRAME enables the use of the three previous bit. Bit 7 is reserved and bits 0 to 2 are hardwired to 010B.

Bit 7 is reserved. The bit 6 (D_OPEN) indicates that if the SMRAM space is accessible (D_OPEN = 1) or not (D_OPEN = 0). The bit 5 (D_CLS) when set (D_CLS = 1) makes the SMRAM inaccessible for data references, although code references still accessible. Bit 4 (D_LCK) blocks all writing access to the SMRAMC register when it is set (D_LCK = 1). Bit 3 (G_SMRAME) enables the use of D_OPEN, D_CLS and D_LCK, when it is set (G_SMRAME = 1). The field C_BASE_SEG is composed by the bits 0, 1 and 2. It indicates the location of SMRAM and it is hardwired to 010B.

Figure 3.5 shows the SMRAMC state before BIOS initialises the SMRAM space. As C_BASE_SEG is hardwired, its value remains the same during all time. G_SMRAME is set to allow the BIOS operate over the SMRAM. D_OPEN is set to allow

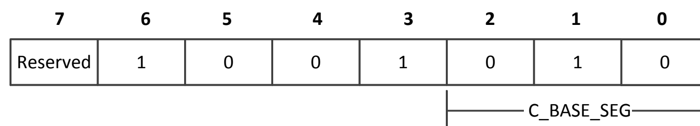

Figure 3.5: **SMRAMC state before SMRAM initialisation.** Bit D_OPEN is set to allow the BIOS to upload the SMI handler code and data to the SMRAM. Bit D_LCK is cleared to allow the BIOS to change the register values after finishing the SMI handler upload process. Bit D_CLS is cleared to allow data references.

BIOS writes the SMI handler and all the necessary code and data for SMM operation in the SMRAM.

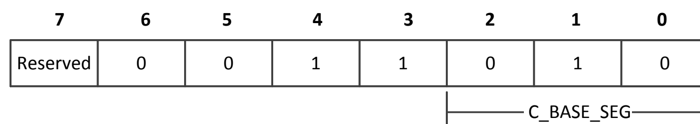

Figure 3.6: **SMRAMC state after SMRAM initialisation.** After BIOS finishes the SMI handler upload, it clears bit D_OPEN, set bit D_CLS and D_LCK. So, now SMRAM and SMRAMC are inaccessible and no data reference is allow to the SMRAM.

Figure 3.6 shows the SMRAMC state after BIOS initialises the SMRAM space. D_LCK is set and the D_OPEN is cleared (necessarily) and then SMRAM space and SMRAMC register are both locked.

In machines released about 2004 or before, it is quite common the situation where D_LCK is cleared and the D_OPEN is set after the boot up process [69, 70]. Then, considering that SMBASE was relocated to A0000H, if D_OPEN is set and the processor is in SMM, the access using address A0000H is routed to the SMRAM space. If D_OPEN is set and the process is in protected mode, the access is routed to the SMRAM space too. If D_OPEN is cleared and the process is in protected mode the access is redirected to the legacy VGA memory space [45, 69, 70, 92, 93].

3.2.4 SMI handler

The SMI handler is the SMM executive software. There are several events that can trigger an SMI, each event requesting a different action. For example, in our target chipset 1 there are 38 reasons to cause an SMI [81] and in the target chipset 2 there are 35 reasons to cause an SMI [64].

SMI handler is a powerful software. So, it is necessary to understand its details and features before developing a SMM-based security tool. Since the BIOS can be updated, and it loads the SMI handler in the SMRAM, the SMI handler can be used as a workaround to patch bugs on hardware [143].

The first issue is the SMI handler **size**. This depends on the OEM or BIOS manufacturer that creates it, as established by Intel, it is important to handle the system event which causes an SMI, but not how [66, 71, 97]. In fact, the SMI handler has 32256 bytes available for its code, data, heap and stack (figure 3.2) in the 32-bit architecture and 31744 bits in the 64-bit architecture (figure 3.3). So, when developing a SMM-based security tool, it is mandatory to stick to those sizes.

We have performed some compilation experiments with the SMI handlers in Coreboot 4.4 2.4.1. We could not get a precise size of the SMI handlers executable code, since there are many external code dependencies in the SMI handlers codes and all of them call other functions in many other codes around Coreboot code.

Another issue is the quantity of SMI handlers in the system. This depends on the OEM or BIOS manufacturer strategy to deal with the system management events. But, even if there is more than one SMI handler deployed in the SMRAM, the processor will always search for the SMI handler at SMBASE + 8000H address. Probably in this case an SMI handler would call the other one. In [143] is described some experiments on disassembling a BIOS of ASUS P5Q motherboard, which is based on AMIBIOS 8. Those experiments report more than one SMI handler present in the BIOS static code, each one seeming to deal with a different set of tasks, but it is not clear if all of them are loaded in the SMRAM space during the boot up process or if a specific SMI handler is chose depending on the host chipset.

It is not clear if the SMI handler call another code in the system, which would be located out of SMRAM space. Nonetheless, since SMI handler can access any part of the system, with full privilege, it is plausible that it calls some firmware to complete specific management tasks, depending on the strategy planned to deal with the system management events by the manufacturer.

A critical issue to the SMI handler is how many times are SMIs triggered in an interval "*t*"? There are several factors that can influence the numbers of SMI triggered in the system, for instance: chipset, architecture, platform, OEM and BIOS manufacturer. It is hard to determine a general or global figure to answer this question. However, it is possible to probe systems to get clues. The Intel BIOS Implementation Test Suite (BITS) [90, 98] provides a set of functionalities to test a system endowed with Intel processors. BITS tests the SMI latency and frequency among others things. The next section presents an experiment with BIOS tool.

An important issue is how long does the SMI handler take to accomplish its tasks? Since each event triggers an SMI to deal with a specific task, the time SMI handler spends to accomplish it depends on the kind of task to be performed. The next section presents an experiment with BIOS tool to shade some light in that matter.

From that comes another issue: what would be acceptable in terms of time spend in SMM? Since the system executive software stands still when the processor is in SMM, the SMI handler must be as fast as possible. But, how fast? The BITS tool defines that the SMI latency must be less than 150 μs to minimise the risk of system executive software time-out [90, 98]. Some experiments conduct in [41] demonstrated that different devices are affected differently depending on the SMI latency. For instance, latency-sensitive applications, like USB audio, can be affected by SMI with duration ranging from 20 to 50 *ms*.

3.2.4.1 SMI Experiments

In table 3.1, it is showed an experiment done in this work with for 13 machines, using BITS. Considering the recommended limit of 150 μs, machines from 1 to 8 failed the test. Again, it is noteworthy that BITS was developed for Intel x86 processors.

Table 3.1: **SMI latency.** In the latency experiment, we considerer the maximum latency time of 150 μs as established by Intel in the BITS tool [98]. Values are tabulated in microseconds.

OEM	Processor	MCH	ICH	Max	Ref
1 - Gigabyte	Dual Core E2160	945P	ICH7	853 μs	[74, 75]
2 - Gigabyte	Dual Core E2160	945P	ICH7	395 μs	[74, 75]
3 - Gigabyte	Pentium 4	945P	ICH7	320 μs	[74, 75]
4 - Gigabyte	Pentium 4	945P	ICH7	318 μs	[74, 75]
5 - Asus Tek	Pentium 4	not id	ICH7	350 μs	[75]
6 - Dell	i5-2500	Sandy Bridge	H67	191 μs	[83, 84]
7 - Asus	Atom N270	82945GSE	ICH7-M	597 μs	[77, 88]
8 - Foxconn	Core 2 Duo	E7500	ICH3-S	290 μs	[68, 76]
9 - HP	i5-650	Clarkdale	P55	0.031 μs	[78, 79]
10- Fujitsu	Pentium 4-M 548	82845	ICH3-M	7.8 μs	[72, 73]
11- HP	i7-2670QM	Sandy Bridge	HM65	0.069 μs	[83, 84]
12- Acer	i5-3450	Ivy Bridge	B75	0.039 μs	[80, 81]
13- Compaq	Pentium III-M	82830M	ICH3-M	69 μs	[64, 65]

Table 3.2: **SMI average latency.** measured by BITS tool [98]. Values are tabulated in microseconds. We can notice that most SMIs occurs in the 0 to 1 μs.

Machine	0 to 1 μs	1 to 10 μs	10 to 100 μs	100 μs to 1 ms
1 - Gigabyte	0.051	0	0	781
2 - Gigabyte	0.051	0	0	388
3 - Gigabyte	0.037	0	0	318
4 - Gigabyte	0.037	0	0	312
5 - Asus Tek	0.051	0	0	347
6 - Dell Inc	0.008	3.64	14	169
7 - Asus	0.046	0	0	0.583
8 - Foxconn	0.019	0	0	245
9 - HP	0.011	0	0	0
10 - Fujitsu	0.038	5.355	0	0
11 - HP	0.016	0	0	0
12 - Acer	0.011	0	0	0
13 - Compaq	0.066	0	0	0

Table 3.3: **SMI frequency.** Obtained by Intel BITS tool [98]. Values are tabulated in unites, we have reported in this table the raw numbers produced by BITS tool. But not all of those unites are properly an SMI. AS one of the author clarified by e-mail communication: "The SMI latency test runs **rdtsc** repeatedly in a loop, and looks for long gaps between iterations". So those numbers are not necessarily all trigerred SMIs. Again, we can notice that most SMIs occurs in the 0 to 1 μs.

Machine	**0 to 1** μs	**1 to 10** μs	**10 to 100** μs	**100** μs **to 1** ms
1 - Gigabyte	291696327	0	0	20
2 - Gigabyte	292672876	0	0	5
3 - Gigabyte	395342184	0	0	5
4 - Gigabyte	396355043	0	0	5
5 - Asus Tek	340636951	0	0	702
6 - Dell Inc	1686431294	243	15	30
7 - Asus	324413121	0	0	58
8 - Foxconn	768134799	0	0	20
9 - HP	1282771330	0	0	0
10 - Fujitsu	295877313	669040	0	0
11- HP	882002283	0	0	0
12- Acer	1271119669	0	0	0
13- Compaq	223088353	0	0	0

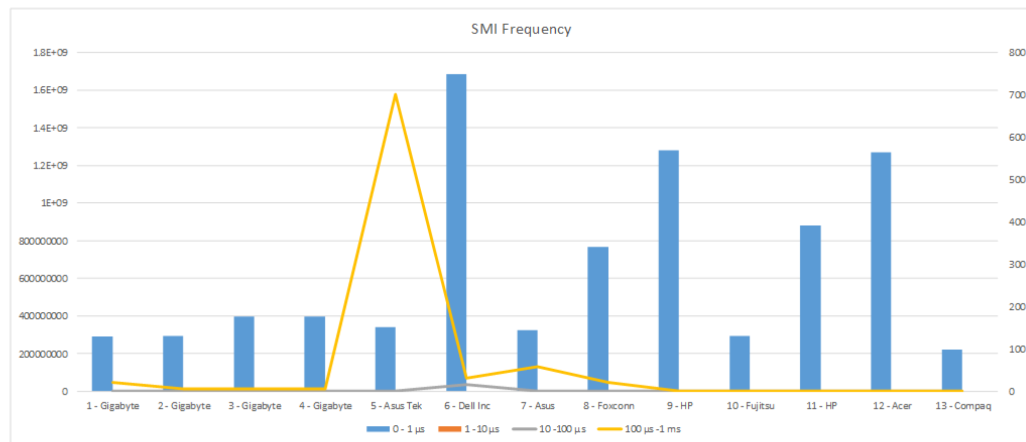

Figure 3.7: **SMI Frequency Graph.** In general, the graphics shows that newer machines trigger SMI more often.

3.3 SMM operation and relations

3.3.1 Entering and exiting from SMM

Figure 3.9 explains the process of entering in and exiting from SMM. To enter SMM, a system management interruption (SMI) must be generated. There are several reasons to an SMI be generated, depending on the chipset in use. But, all of them

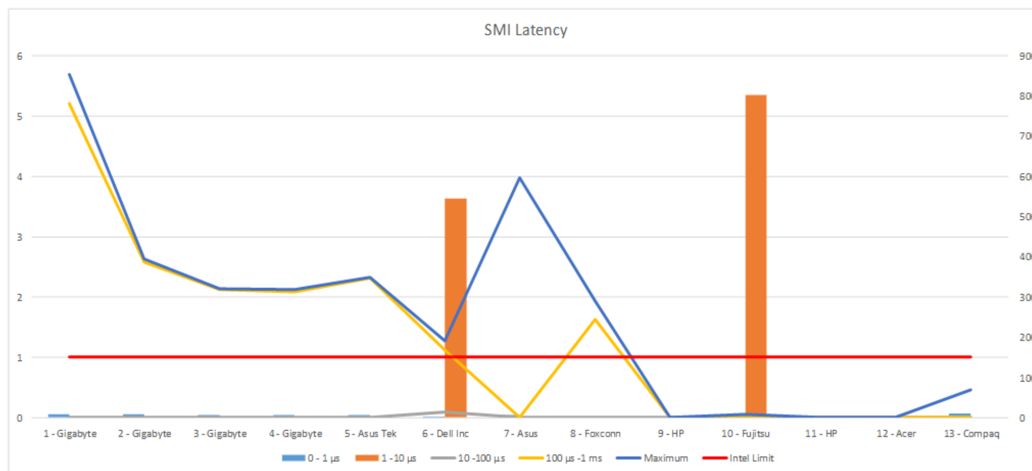

Figure 3.8: **SMI Latency Graph.** Observing this graphic, we can see that newer machines have, in general, smaller latency time.

are related to system management functions (1). No matter the reason, the only way to generate an SMI is by signalling it in the SMI# pin on the processor, which is referred as SMI triggered by hardware, or by an SMI message received through the Advanced Programmable Interrupt Controller (APIC) bus, which can be referred as SMI triggered by software (2). Then, the processor informs external hardware that an SMI handling has begun, by generating an SMI acknowledge transaction on the system bus (P6 family processors) or by asserting the SMIACT# pin (Pentium and Intel 486 processors) (3) [87, 11]. SMIACT stands for SMI active [11, 127].

Upon entering SMM, the current processor state is saved in the SMRAM (4) and no other interruptions can affect or be handled in the system, including another SMI. Then the processor looks for the first SMI handler instruction at the address SMBASE + 8000H (5). SMI handler set or clear SMM related register, which just can be manipulated in SMM (6) (section 3.2.3).

To exit from SMM, the SMI handler executes its exclusive RSM instruction (7). There is no other way to exit from SMM. Upon exiting from SMM, the processor always comes back to the mode it was before enter in SMM (8), restoring the state saved (9) and passing control to the interrupted process before entering to SMM or dealing with other interrupts which might be generated during the SMM time (10) [87, 24, 127].

The BIOS uploads the SMI handler code and data to the SMRAM during bootup process (0).

3.3.2 Caching

IA-32 processors do not invalidate their caches before entering or exiting from SMM, which can allow information leakage. The main recommendation is avoid make SMRAM space cacheable (section 4.4). So, Intel recommends three methods,

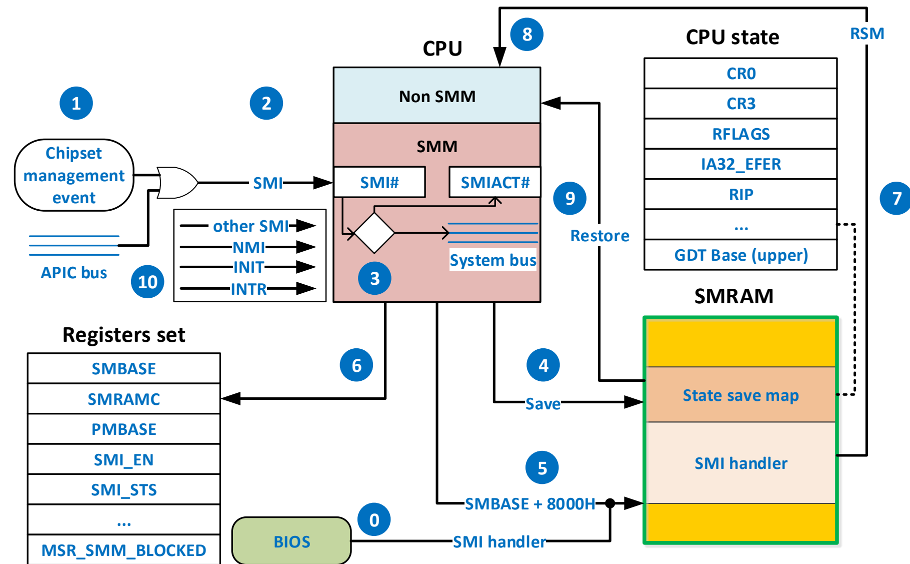

Figure 3.9: **Entering and exiting from SMM.** This figure exposes the process of entering and exiting from SMM.

that can be adopted by BIOS manufacturers and OEM in general when developing SMM resources to deal with this caching issue [87]:

1. Place the SMRAM in a memory region inaccessible for operating system and applications and allow this region be cacheable. This method might avoid that a malicious code jumps to the SMRAM space, after successful attack the cache. However, it does not prevent leakage of information stored in the cache if a successful attack occurs.

2. Place the SMRAM in a memory region that can be overlapped for operating system and applications and encode SMRAM as uncacheable (see section 3.2.3 and [86, 87] for more on memory encode). This will reduce the SMI handler performance, impacting in latency time.

3. Place the SMRAM in a memory region that can be overlapped by operating system and applications, but flushing the caches before entering and exiting the SMM. It can be done by asserting the FLUSH# pin and then the SMI# pin after an SMI has been triggered. This will cause some overhead due to two flush operations.

3.3.3 Multiple-processor systems

In a system with multiples processors, one or more processors can be in SMM at the same time. When more than one processor is in the SMM, each one needs its own SMRAM space. However, this space can be overlapped and code and static data can be shared [87]. Nonetheless, each process needs its own state save map and dynamic data storage areas. Because of this requirement and since the state saved map is fixed at SMBASE + FE00H, the SMI handler needs to initialise the SMBASE register for each processor.

3.3.4 SMM and Virtual Machine Extensions (VMX)

VMX provides the processor support for virtualization. VMX operates with two mode of operation: VMX root operation, that is privileged and used by the hypervisor; and VMX operation, that is non-privileged and used by guest VMs. The Virtual Machine Control Structure (VMCS) controls several functions related to VMX and it can only be accessed by the hypervisor [57].

Processors supporting VMX can deal with SMM in Default Treatment or Dual-monitor Treatment. The Default Treatment is quite similar to the SMM operation showed before.

In the Dual-Monitor Treatment, the system uses two hypervisors (or virtual machine monitors or VMM): an executive monitor (as Xen [18]), which operates outside the SMM and provides virtualisation services for guest virtual machines (guest VM); and the SMM-transfer monitor (STM), operating "inside SMM" and offering SMM functions to guests VM executing in VMX operation mode [87].

Then, in the dual-monitor treatment there is a cooperative work between the executive monitor and the STM. The control is transferred from executive monitor (or from its guests) to STM by means of a SMM VM exit and from STM to executive monitor by means of a SMM VM entry [57, 91]. Note that, when using STM, the processor does not use a RSM instruction to exit from SMM, but a SMM VM entry. STM operates in VMX root operation mode.

With VMX, the SMM code used to be loaded by using three aliases memory addresses: HSEG, TSEG and CSEG [57]. Those aliases specify regions in the memory. However, the contemporary platform does not support HSEG and CSEG anymore [93]. Nonetheless, the only alias important to STM is TSEG. TSEG is divided to include an area called monitor segment area (MSEG). The BIOS loads STM code and data in the MSEG area inside of SMRAM [87] and the remaining SMM code in the TSEG area [57].

Not all processors support the Dual-Monitor Treatment. It is necessary to consult the IA32_VMX_BASIC register to check information about support [87].

3.3.5 The Intel Software Guard Extensions (SGX)

The reason behind to use SMM for security purposes is because it has powerful resources in the system.

In fact, using SMM for security purposes is a sort of workaround. The real necessity is something in the system that can provide the same resources as SMM provides, especially a fair amount of protection and isolation for the security so-

lution code and data themselves. Possibly, with efficient protection and isolation even a security tool would not be necessary: applications could perform on such protected and isolated environment directly.

In this context, the potential answer for security purposes can be the Intel Software Guard Extensions (SGX). SGX is a Intel architecture that allows applications to execute in the native operating system in an isolated environment, enabling confidentiality and integrity. SGX provides an isolated and protected container, called enclave, established in the application's address space [111].

This new architecture is enabled by means of 18 news instructions added to the Instruction Set Architecture (ISA), a new processor mode, called enclave mode and new data structures [111]. It can provide security for different kinds of applications, ranging from streaming of video to high security demand financial transactions [60]. SGX also provides strong mechanisms for attestation and sealing. The attestation can be local, between two enclaves executing on the same platform, or remote, when it is necessary to proves identification to a third party [10]. The Intel SGX is not yet available commercially.

From the previous description, we can see that the enclaves are like virtual machines, managed essentially by instruction in the processor. Then, we might understand SGX as the feasibility of a "hypervisor in processor". Since SGX is embedded in the CPU it can be very safe.

3.4 Security implementations using SMM

The current SMM-based security architecture is presented in figure 3.10. In the next subsections, we present SMM-based security tools and point out opportunities to improve them. Although AMD processor are not targeted in this research, we believe it is useful to discuss some of the SMM-based security tools developed for AMD processors.

3.4.1 Measuring the Hypervisor Integrity

In [141] is presented HyperCheck, a hardware-assisted framework to protect the integrity of hypervisors, using SMM. HyperCheck comprises of three components: memory acquiring module, analysis module and CPU register checking module.

The memory Acquiring Module reads the contents of the physical memory and sends them to a remote machine where the analysis module checks their integrity. It is implemented as a network interface card (NIC) driver and placed into SMRAM. The CPU register checking module reads the registers and validates their integrity. This former module is in the SMRAM. The acquiring module is located in a PCI NIC and the analysis module is located in a remote machine.

Like HyperCheck, HyperGuard [146] uses the SMM to provide hypervisor integrity protection. It combines the SMM with chipset integrity scanning and takes a different approach to integrity measurement: rather than measure the code and data of the hypervisor to guarantee that they are correct, it ensures that there is no malicious code in the hypervisor. HyperGuard requires the cooperation of BIOS developers.

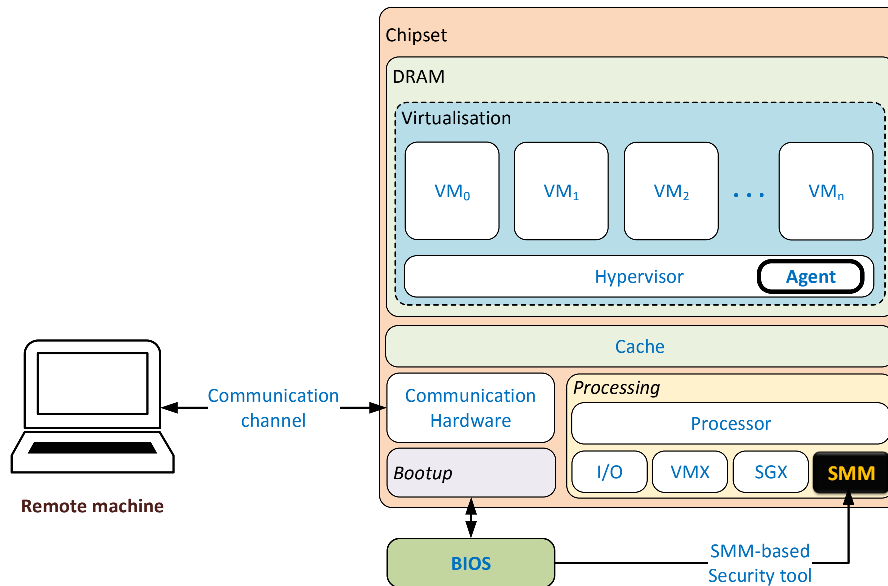

Figure 3.10: **Current SMM-based security tools architecture.** Any tool capitalising on SMM must somehow emulate SMI handler. In general, tools count on an agent inside of DRAM and a remote machine to analyse collected data.

According to [14] HyperCheck and HyperGuard cannot always check the state of the hypervisor since the hypervisor context can be hidden, depending on the moment an SMI is triggered. This is the case for machines with VMX capabilities, since the CPU can be in a root mode or in a guest mode [87, 57]. When in guest mode, if an SMI is triggered, the CPU context would be invisible for those tools. Moreover, another potential issue indicated by [14] is that HyperCheck and HyperGuard do not provide a way to start the integrity measurement without alerting the hypervisor. This might allow an attacker to perform a scrubbing attack (like deleting evidences of the attacker actions).

HyperSentry [14] capitalises on the hardware architecture and the SMM to implement a framework supporting stealthy in-context integrity measurement for hypervisors. The framework uses a measurement agent running in the hypervisor code base to check the integrity of the running hypervisor. HyperSentry uses a remote verifier machine, which triggers the measurement agent. To keep the invocation of HyperSentry stealthy, the remote verifier uses a Intelligent Platform Management Interface (IPMI) as communication channel. IPMI connects to a Baseboard Management Controller (BMC) in the target machine, which triggers an SMI. After SMI triggering HyperSentry, HyperSentry invokes the measurement agent and executes an RSM, in such a way that the next instruction to be executed will be a measurement agent instruction. It is not clear in [14] how that is done, but, it is likely to be done by writing the measurement agent address to the EIP (or RIP) register. Besides isolation and stealthy, HyperSentry tries to address the issue of hypervisor integrity measurement during runtime; but it addresses that issue in superficial way, since the main purpose of the work is to provide the framework.

3.4.2 Other security solutions

The Strongly Isolated Computing Environment (SICE) [15] is another framework that capitalises on SMM to provide an isolated execution environment. Its architecture involves only the hardware, the BIOS and the SMM. SICE is implemented on AMD processors and its main function is to manage isolated environments. The idea behind SICE is to execute in parallel with the SMI handler. Then, SICE operates under two modes: time-sharing mode and multi-core mode. In time-sharing mode, SICE uses time multiplexing to share the hardware platform between the system executive software and the isolated environments. In multi-core mode, one or more core is used by the isolated environment and the remains are used by the system executive software. The system executive software and its drivers are required to be modified to provide services to the isolated environments.

Essentially, SICE uses SMM resources to prepare the isolated environments. So, the isolated environments are put into the SMRAM, within the limit of 4GB; and, by setting the SMM_Addr and SMM_Mask AMD registers, SICE shrinks the SMRAM space in such way that they stay out of the SMRAM space. Thus they can be executed out of SMM, but keeping the same level of isolation provided by SMRAM.

In [61] a scheme is presented for auditing cloud computing systems, making use of SMM, TPM and HyperSentry [14]. It uses the same mechanism described above, in HyperSentry, to provide stealthy measurement aiming to detect persistent and non-persistent threats in the system.

AppCheck [142] is a similar tool to HyperCheck, implemented in the Intel platform. However, it aims to protect applications by inspecting their code in the physical memory. In this sense an application means a user level process.

MUSHI stands for Multiple level security cloud with strong hardware level isolation [151]. It is a framework that uses SMM and TPM to provide isolation for guest Virtual Machines in cloud environments. MUSHI, a modified version of an SMI handler, copies certain parameters to SMRAM, measures them and sends the results to a remote party, which verifies the integrity of the data sent. As far as we noticed from [151], MUSHI was not implemented.

HypeBIOS [152], like MUSHI, aims to provide isolation for Virtual Machines and is implemented on AMD platforms. It reduces the virtualised environment's TCB by eliminating some components from the boot-up process. Then, HypeBIOS includes two additional layers in the system: a master layer (modified SMI handler) in SMRAM to perform measurements in general; and a slave layer positioned between the master layer and the hypervisor, intercepting and mediating some communication between VMs and the master layer.

BIOS Chronomancy [25] capitalises on SMM, TPM and time-based attestation [104] to implement a Core Root of Trust for Measurement (CRTM) to demonstrate and fix fails on the implementation requirements described in TPM recommendations.

Another proposed framework is SPECTRE [149]. However, its objective is to inspect the system state, using techniques of virtual machine introspection (VMI) [53]. VMI makes use of on virtualisation resources to obtain visibility of a monitored host system, while maintaining a good level of isolation. In this sense, VMI balances the power of visibility achieved by host intrusion detection system (HIDS) with the isolation and protection provided by a network intrusion detection system

(NIDS). So, the main goal of SPECTRE is detecting malicious code running in the host machine.

In this system, the host machine periodically triggers an SMI by means of a timer in the chipset, so that the SMI handler (SPECTRE) monitoring module is performed. After concluding, a status message is sent to a remote monitor machine. This architecture is reported as being able to identify different kinds of attacks, such as heap spray, heap overflow and root-kits The implementation was done using an AMD machine and, as with HyperCheck, SPECTRE uses a custom NIC driver.

IOCheck [150] is a framework to improve security of I/O devices at runtime by checking their configuration and firmware integrity. Its architecture is closely related to SPECTRE: the host (or target) machine triggers an SMI, using a random-polling (triggered periodically) or an event-driven approach (triggered whenever particular events happen). Then IOCheck (a modified SMI handler) verifies the integrity of configuration and firmware files on the target device. If a potential attack is detected, the host machine emits a sound alert and sends an alert message to a remote machine. After concluding its tasks, IOCheck executes an RSM instruction.

3.4.3 Brief Analysis of the Security Implementations

HyperCheck [141], HyperSentry [14], auditing tool [61], AppCheck [142], MUSHI [151], SPECTRE [149] and IOCheck [150] use a similar architecture, taking in account a remote machine to analyse the collected data and for other management functions.

These tools assume that SMRAM is tamper-proof and that an attacker has no access to the physical machine. This is a quite strong assumption, since the reported attacks (see section 3.5) have shown how to compromise the SMM. HyperSentry, auditing tool [61], HyperCheck, SPECTRE and IOCheck further assume that the components involved in communications (as the channel, the NIC and the remote machine) are safe.

The tools use modified versions of the original SMI handler. HypeBIOS, SPECTRE and IOCheck use SMI handler from Coreboot [36]. The SMI handler has crucial system management functions. However, it is not clear what happens to the original functions of the SMI handler after the modifications have been made, or to the level of cooperation between tools and the original SMI handler. Moreover, these tools need some form of cooperation to be installed, since they require changing the BIOS. HyperGuard, for example, is a project supported by a BIOS manufacturer. No further information has been found about HyperGuard.

In HyperCheck, the memory Acquiring Module is implemented as a NIC driver deployed into SMRAM. However, it is not clear how the SMI is triggered to start that module.

In HyperSentry architecture the weakest security point is the measurement agent, since it resides in the hypervisor. After an SMI has triggered the HyperSentry's SMI handler, the handler invokes the measurement agent and executes an RSM in such a way that the next instruction to be executed will be a measurement agent instruction inside the hypevisor. It is not clear how this operation is performed, but it is likely to be done by altering the EIP register. Then, that agent performs the measurement. So another SMI is triggered and the measurement results are stored in SMRAM space.

There are two more unclear issues in the operation above. Firstly, how does the measurement agent trigger the SMI to store results in the SMRAM? Secondly, since the measurement agent is not an SMM code, it cannot store data in SMRAM space at all.

Although HyperSentry is focused on the framework, the measurement agent deployment strategy might prevent HyperSentry from delivering the proposed security. Firstly, the agent is common code in the hypervisor, so it can be tampered with by the same hypervisor's threats. Secondly, If the agent cannot trigger the SMI in the same way as the remote verifier does, it will prevent the proposed stealthy operation. Finally, since the agent has no way to write to the SMRAM space, it will probably deliver the measurements to some unprotected memory space for subsequent collection by HyperSentry's SMI handler. This memory space could be tampered with by malicious code.

Another issue is the execution time for the tools. As seen in section 3.2.4 the SMI latency must be less than 150 μs [90]. However, table 3.4 shows that most of the tools have execution time superior to that limit.

It is noteworthy that the time report by SICE (67 μs) is the time required to manage the isolated environment (create, enter, exit and terminate) and not the execution time for any workload. This means that the workload in SICE executes in an isolated memory space, but the processor is no longer in SMM mode. So although the workload benefits from the memory isolation property, it cannot capitalise on any other SMM resources.

Table 3.4: **Reported execution time.** Almost all tools were not able to keep their execution time under the maximum latency time recommended by Intel: 150 μs [90]

Tool	Time	Objective	CPU
HyperCheck	40 ms	Check integrity of hypervisor	Intel
HyperSentry	35 ms	Check integrity of hypervisor	Intel
SICE	67 μs*	Provide isolated workload	AMD
SPECTRE	62.5 ms**	Virtual machine introspection	AMD
IOCheck	10.4 ms	Check integrity of I/O devices	AMD

(*) Time to prepare the isolated environment.
(**) Worst case scenario.

3.5 Launching attacks using SMM resources

In our research, we have not identified any work addressing the issue of establishing requirements to use SMM for security purposes. But, there are works attacking, misusing and implementing security tools capitalising on SMM. We describe briefly these works in the following subsections. An attack targeting SMM is essentially neither a hardware attack nor software attack. But it is classified closer to the hardware than to the software, since it needs sort of physical access and low-level system components access.

3.5.1 The first wave of attacks

Duflot et al. [45] published one of the first known works on exploiting SMM. That work explains how to use SMM to circumvent operating system protection mechanisms, permitting privilege escalation. The authors establish and demonstrate a scheme comprising of six sequential steps: 1) enable SMI; 2) open SMRAM space; 3) replace default SMI Handler by a customised one; 4) close SMRAM space; 5) trigger SMI and 6) gain full access to the system.

Although this attack is practical, it is operating system dependent, mainly in Linux-like systems since Linux allows more control over low-level system functions. Moreover, the attack assumes an attacker with system administration privileges.

Branco [23] presents a more generic attack and shows how to use the SMM to deploy malwares. Essentially this follows the same steps described in the work of Duflot et al. [45]. However, the way some steps are performed differs. For instance, in the step "open SMRAM space" the SMRAMC register is manipulated to accomplish this in both [45] and [23]. However, Duflot et al. use the operating system functions to do it, while Branco uses libpci [12, 106] to manipulate the register, which makes this attack more generic than that described in [45].

Embleton et al.[47] presents a generic attack using SMM. They demonstrate their ideas by implementing a key-logger and a network back-door which work together to capture keystrokes and send them to a remote machine using the UDP protocol. They trigger an SMI by software via the APIC bus in such a way that each time a key is pressed an SMI is triggered and the key-logger is launched. The key-logger then logs the key, sends it to the remote machine and triggers the normal interruption for the keystroke as would be expected by the CPU.

That attack has limitations, some of which were reported in the paper. One limitation is the high overhead. The paper reports that the overhead is negligible. However, since any key pressed triggers an SMI and at least three steps are needed to be accomplished after that, it will generate a high overhead, potentially causing the operating system or hypervisor to freeze.

Another issue is that is not clear what happens with the original SMI handler. Considering the design of the SMM, the original SMI handler was most likely overridden by the key-logger code. If the original SMI handler is not launched to handle an SMI, the system may crash. This would be a denial of service, which has value to an attacker, but was not intended by the attacker. One more issue is what would happen when a legitimate SMI is triggered. Most likely, in the advent of a legitimate SMI, since the CPU always searches for the SMI handler code at SMBASE + 8000H, the key-logger would be executed. This would try to log a keystroke and would launch a keystroke interruption, which may cause an unpredictable error on the system.

3.5.2 The second wave of attacks

In 2009, Duflot et al. [46] presented another paper reporting a new attack using a technique called "caching poisoning" to circumvent the D_LCK bit protection, allowing access to the SMRAM. It consists of making the SMRAM cache-able, by encoding it to "writeback" (WB) triggering an SMI to have the SMI handler in the

cache and then replacing the memory content at SMRAM address with a malicious SMI handler (for example), modifying its image in the cache.

The tough point in this attack is “making the SMRAM cachable”. However, since the attacker must have system administrator privileges, the attacker just needs to execute a set of assembler instructions to manipulate the MTRR registers to accomplish this task.

Another attack uses the same cache poisoning technique was published in [148]. The steps described there are quite similar to those ones published in [46]. The paper [148] provides detailed information about the attack, emphasising that it is hardware-dependent. In particular, that attack is based on Intel DQ35JO motherboard with 2 Gbytes of RAM. This motherboard was reported in [119] as having a bug, which would allow bypassing the Xen hypervisor protection. That attack was called ”Q35 remapping bug” [147]. The paper also presents a variant of the attack, which allows the attacker to obtain more information about the SMI handler by reading its code. This attack variant crafts the malicious code to read data cached without polluting it with new and undesirable data.

In [147] an attack against Intel TXT is successful in demonstrating the power of malicious code capitalising on SMM resources. This attack uses the techniques described in [148] to exploit, via malicious code inserted in SMRAM, a Xen hypervisor loaded by Intel TXT.

One limitation in attacking the SMM is reported in [147]. They argue that there is no way to get access to the SMI handler code image running inside SMRAM, since it is locked for non-SMM code. In this context, [143] describes how to analyse the SMI handler by means of reversing engineering the BIOS firmware. This exploit allows also the SMI handler to be modified to embed malicious code in it. The paper presents an SMM keystroke logger and its respective detection code, which is more sophisticated and with more details than that one described in [47]. However, it is hardware specific, designed to exploit the BIOS of ASUS P5Q motherboards, based on AMIBIOS 8 [143].

A more recent misuse of SMM is reported in [122]. Using an AMD platform, the authors introduce a root-kit into the SMM to intercept events and capture data from USB devices. Thus, they are able to intercept, replace and inject keystrokes stealthily. The reported time for each keystroke is 61μs.

The authors gained an advantage, compared to the previous key-loggers [47, 143], by intercepting the communication before it reaches the OS kernel. To do that, they take advantage of a feature in the Host Controller Control Register (HC Control register) defined in the Open Host Controller Interface (OHCI) specification, supporting the USB 1.1 devices [33]. Generally speaking, the Host Controller (HC) is the hardware that controls the USB devices connected in a host machine and the HC control register is used to define the operating modes for the HC [33]. Its 8th bit is the interrupt routing bit (IR). When cleared, interrupts are routed to the host bus interrupt mechanism. When set, interrupts are routed to the SMM. Since the keylogger is a modified version of the SMI handler, by setting the IR bit, the key-logger is able to intercept all USB events.

Some limitations of this work [122] are: 1) the ex-filtration technique, which is done by saving the logged keys in a memory region outside of SMRAM; 2) As discussed previously, the modification of the original SMI handler could cause several

problems in the system; 3) The installation process needs physical access to exploit possible bugs present in the chipset [148] or to use a customised BIOS, such as coreboot [36, 23]; 3) The attack is based on an old specification. Even the old Enhanced Host Controller Interface Specification (EHCI) for Universal Serial Bus, supporting USB 2.0 devices has a complex scheme to allow SMI and has a set of register to deal with that, such as the USB Legacy Support Control/Status (USBLEGCTLSTS), which enables SMIs for every EHCI/USB event the BIOS needs to track [67].

3.5.3 Thwarting the Attacks

The attacks in the first wave assume that bit D_LCK is cleared or that the attacker is able to manipulate SMRAMC register. Because of that, BIOS manufacturers have been deploying BIOS and setting the D_LCK bit after the SMM code and data are loaded into SMRAM thus blocking SMRAMC register access.

One way to overcome this protection, as suggested in [23], is replacing the BIOS by an open source BIOS, as Coreboot [36], or circumventing the BIOS protection mechanism to install the malicious code in the original BIOS space, taking advantage of BIOS updating features, as described in [22]. BIOS replacement attacks are hardware attacks [57] and can be avoided by access control to the hardware.

The attack described by Embleton et al. [47] might be detected by checking the APIC table integrity and might be avoided by preserving the APIC table integrity, since it triggers an SMI by means of the APIC bus.

The "cache poisoning" attack is possible, because IA-32 processors do not invalidate their caches before entering or exiting from SMM. This allows information leakage. To mitigate these attacks, Intel recommends in [87] that BIOS manufacturers and OEM should deal with the issue by placing the SMRAM in a memory region that:

1. is inaccessible for operating system and applications, and allow this region be cache-able

2. can be overlapped for operating system and applications and encode SMRAM as not cache-able.

3. can be overlapped for operating system and applications, but flushing the caches before entering and exiting SMM.

Another improvement that mitigates "cache poisoning" attacks is the System-Management Range Register (SMRR) Interface. This restricts access to the memory address range in the SMRAM used by the SMI handler code and data. Thus, cacheable address references to SMI handler are limited. To check if the processor supports the SMRR interface, one needs to check if the bit 11 of IA32_MTRRCAP register is set. The SMRR interface contains two Model Specific Registers (MSR): The IA32_SMRR_PHYSBASE defines the base address for the SMRAM and the memory type used to access it; the IA32_SMRR_PHYSMASK determines the SMRAM address range protected. They can only be written to when the processor is in SMM mode [87].

Two other registers must also be considered to improve security: the MSR_ SMM_FEATURE_CONTROL register, which is used to restrict the SMI handler address range; and the MSR_ SMM_MCA_CAP register, which offers additional write protection to the MSR_ SMM_FEATURE_CONTROL register [87].

3.6 Discussion

This chapter described the SMM components and discusses how each one contributes to the SMM operation. We investigated and mapped the registers related to SMM and how those register influences the SMM operation. Some registers have great influence on SMM, as the System Management RAM Control (SMRAMC), which working as an access control mechanism to the SMRAM and other are only flags to indicate the SMM components status as the SMI Status register (SMI_STS). There are SMM related register related to control (_CTN), enabling functions (_EN), indicating status (_STS) and so forth. We described the SMM operation and why it has timely resources to be used by security tools.

We investigated the SMI handler since it is potentially the more powerful software artefact in Intel architecture. In fact, any security tool (or any software) capitalising on SMM must emulate the SMI handler in some way. The SMI handler is in a certain extent obscure and some questions related to it are fundamental to develop SMM-based tools, for instance, the size of SMI handler, how many times an SMI occurs in a space of time and how long it can stay on execution without compromise the host machine operation. About this last subject, we conducted an experiment in this chapter with 13 Intel based machines using the BITS tool [90] to determine the SMI latency and we find that eight machines failed in the test (table 3.1).

Intel architecture comes with different architectures to improve the capacities, ability or security of the Intel-based machines, as Virtual Machine Extensions (VMX), Intel Trusted Execution (TXT) and Intel Software Guard Extensions (SGX). Those technologies have interactions with SMM. Then, this chapter identified and discussed such interactions.

The security tools capitalising on SMM were discussed. An analysis of their design and opportunities of improvement were indicated in this chapter. On the other hand, published attacks against SMM were reported and analysed. We presented ways to thwart those attacks and discussed if they are feasible nowadays.

3.7 Summary

The SMM-based security tools discussed in this research were not able to fully use the main resources of SMM, as isolation and transparency on execution. They also were not able in keeping their execution time under the maximum latency time for the SMI handler recommended by Intel, according to [90]. We say, SMI handler because SMM was not design for general-purpose. Moreover, Intel recommends not using SMM for general-purposes [87]. That recommendation aims to preserve the important SMM functions to avoid that a “general-purpose” software violates the strict limits and constraints of SMM resources and disturb the correct function of SMM components, as by overwriting the SMI Handler (details in sections 3.2.4

and 3.4). So, this research establishes a set of requirements, which a security tool must met to overcome those limits and constraints, allowing the use of SMM for "Security-purpose". About using SMM for "general-purposes" see details on the opening of this chapter and in contribution 4. It is important to notice why security tools need to capitalise on the SMM. And that is Because the system environment is insecure; the tools need to be protected themselves and SMM can offer an isolated and protected environment. On the attacks, they only work on specific chipsets and when certain conditions are satisfied, as when the bit D_LCK in the SMRAMC register is cleared.

SGX may be the Intel's answer for the security demanded by the SMM-based tools. Applications using SGX Essentially use the set of new instructions provided by SGX to handle the data structures and then manage their enclaves. An enclave itself is similar to a virtual machine in concept and the process of managing the enclave is similar to that used by hypervisors to manage virtual machines. Since, the instructions to manage enclaves comes from the processor, SGX could be thought as a "hypervisor in processor" or as a "hypervisor on a chip". Naturally, enclaves from SGX and hypervisors have different threat models, but this comparison is useful to understand the general idea behind SGX.

It should be emphasised that SGX does not supersede the functionality of SMM. In fact, SGX instructions are only available when the processor is in protected mode and Intel clearly defines the interactions among Intel SGX and SMM. Note that SGX is a future feature to be added in Intel processors, therefore, it is not available commercially yet [94].

This chapter successful investigated and described SMM components and SMM operation. It identified the SMM relationship with other Intel technologies, as Intel: VMX, TXT and SGX; and analysed security tools and attacks capitalising on SMM. Then, in the next chapter, we analyses all findings and results from the present chapter to build a threat model for an SMM-based security tool. From the threat model, we make the assumptions considered when formulating and designing the answers for the research questions proposed and when building the proof of concept. Finally, we establish a set of requirements for using SMM for security purposes.

Chapter 4

Requirements

4.1 Introduction

The last chapter investigated and detailed SMM components and how those components work together during SMM operation. The SMM related registers were investigated and mapped. We have learned that SMI handler is potentially the more powerful software artefact in Intel architecture, due to the resources available to it when the processor is in SMM. Also, SMM needs to co-exist with different Intel technologies, as Virtual Machine Extensions (VMX), Intel Trusted Execution (TXT) and Intel Software Guard Extensions (SGX). The interaction among those technologies were discussed, emphasizing that SGX is not available commercially yet. Finally, last chapter analysed the security tools capitalising on SMM and pointed out opportunities of improvement for those tools. Also, it has analysed known attacks against SMM, suggesting ways to thwart those attacks and discussing their feasibility nowadays.

In this chapter, we examine the findings in chapters 2 and 3 to construct a threat model to the current SMM-based security tools architecture. From that threat model, we make assumptions and then we establish a set of requirements which SMM-based security tools must meet to overcome the limits and constraints of SMM, while dealing with the threats. By meeting the requirements, an $SBST$ can take more from SMM resources, ensuring strong isolation, high privileges and good view of the system. Finally, we indicate what requirement (or assumption) mitigates each threat.

This chapter is organised in three content sections and a discussion and a summary section. In the first section, we analyse the platform around SMM and built a threat model, defining 10 threats in the platform. Section two discusses the assumption made, considering the complexity and the time frame available for this work. Then, third section establishes and discusses the requirements that must be met by SMM-based security tools.

4.2 Threat model

To understand the threats around SMM, we have analysed the current architecture for SMM-based software (not just security tools), considering the attacks reported, the characteristics of the x86 platform and the SMM. By analysing them, we noticed that most of those tools are modularised and their architecture addresses the SMM constrains and limitations, as limited space and maximum SMM latency (execution time limit) with that modularised architecture. However, such an architecture enlarges the TCB, by adding a communication channel, a remote machine and drivers

and devices, as those ones used to enable communication. A **remote machine** is generally used in that architecture to analyse the data collect in the target platform, so the analysing task is done by the remote machine eliminating the overhead of analysing using the local processor in the target platform (figure 3.10).

Thus, we have identified the following threats (figure 4.1):

- ***h1* - Tamper with the analyser.** An attacker can tamper with the remote machine and compromise the result of measurements, so no action would be taken in the measured machine with its integrity is violated. Counting on a remote machine is a common strategy to deal with SMM constraints and limitations, as used in HyperCheck [141], HyperSentry [14], auditing tool [61], AppCheck [142], MUSHI [151], SPECTRE [149] and IOCheck [150]. An attacker can also impersonate the remote machine to receive the collected data when an authentication service is not present between the analyser and the target machine.

- ***h2* - Information disclosure.** The communication channel used to send information to the remote machine, as described in the previous threat, can be monitored and measurement information disclosed or the channel can be disrupted avoiding that collected data can arrive at the analyser. Yet, a man-in-the-middle attack can tamper the collect data, compromising the result of measurements. Some tools as Hypersentry and hypercheck use attestation to address the **Information disclosure** issue.

- ***h3* - Denial of service.** An attacker can disable the hardware used to establish the communication channel between the analyser and the target machine, as described in the previous threats, denying the communication service.

- ***h4* - BIOS replacement.** An attacker may replace the BIOS by accessing the target machine or try replacing the BIOS software during a BIOS update process, removing the security tool and inserting or not a SMM-based malware in the BIOS to upload it in the SMRAM.

- ***h5* - Security tool replacement.** An attacker can try replacing the security tool after the bootup process have started and before the BIOS upload the tool in the SMRAM. So, the attacker would have their one tool in the SMRAM.

- ***h6* - Cache poisoning.** It consists in modifying the SMI handler image present in cache memory, as described in section 3.5.2 and in [46] and [148]. For that attack works, the attacker needs to make SMRAM cachable, by encoding it to "writeback" (WB) and the target machine is not following the strategies suggested in 3.3.2.

- ***h7* - Tamper with the hypervisor.** An attacker may tamper with the hypervisor and take over of it and all the code present in it. So, an agent or any software in the hypervisor environment can be tampered with or denied of working. There are successful attacks against hypervisors, as in the case of a successful execution of code on the host machine from a guest OS in a VMware environment [102] and an exploitation of the Xen hypervisor that allows including a backdoor functionality inside of it [145].

- *h8* **- Tamper with the management VM.** The management VM share with the hypervisor the management function in the virtualised layer. Thus, it is as important as the hypervisor. An attacker may tamper with the management VM and launch attacks against the hypervisor and the other virtual machines. So, the attacker can escalate its exploitation to another parts of the virtualised layer or the host machine.

- *h9* **- Malicious Virtual machines.** An attacker using a malicious VM could force a VM exit to occur, trying to simulate an execution of privileged instructions, and than the attacker could try either to inject malicious code , as described in the vulnerability **CVE-2007-5497** [31] or to cause a malfunction in the hypervisor [135, 31]. By doing this, an attacker might violate confidentiality, integrity and availability of other VMs, the hypervisor and ultimately compromise all the cloud infrastructure.

- *h10* **- Malicious chipset.** A malicious chipset normally requires attackers with a big amount of resources, which can anticipate their attacks by inserting malicious firmware or hardware trojans in the chipset of machines, which may become a target in the future. A malicious chipset is a very powerful platform to launch any kind of attack and may compromise the whole infrastructure around it.

In general, our threat model assumes that an attacker might have complete control over any software and can violate the hypervisor or any other software outside SMRAM.

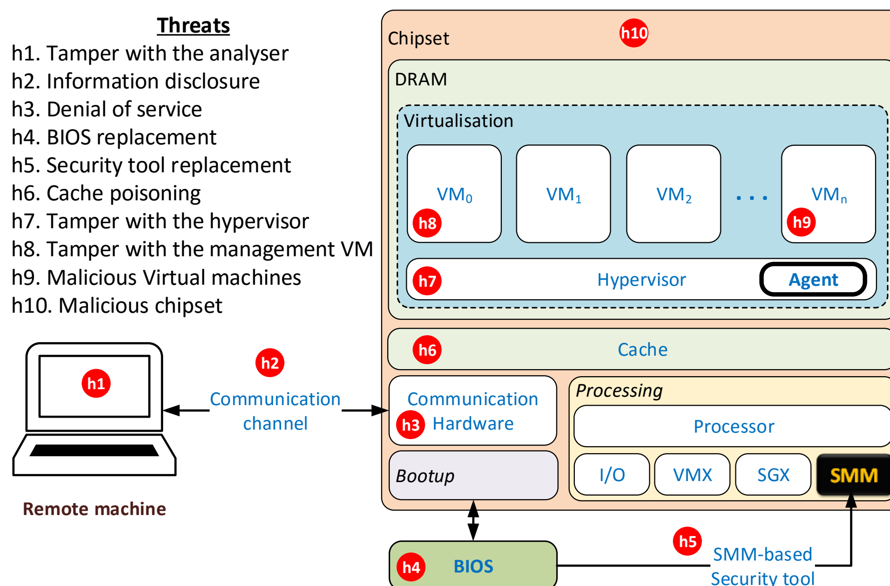

Figure 4.1: **Threat model.** Threats identified in the current SMM-based security tools architecture.

4.3 Assumptions

Those assumptions are related to security issues. Then, since it is neither possible nor convenient to address all threats related to environment considered in this research we make assumptions to render this research problem treatable. Some threats are out of the scope of this research and other ones are difficult to address in the time frame available for this work. In this context, the following assumptions are made:

- ***a1* - No physical access.** *The attacker does not have physical access to the machine and cannot replace the BIOS.* An attacker with physical access to the machine can perform hardware attacks or replace the BIOS. By replacing the BIOS, an attacker could be able to inject an SMM-based malware or simply use a pristine BIOS version, without the SMM-based security tool.

- ***a2* - Trustworthy BIOS.** *At the target platform, the BIOS is trustworthy and safely uploads and sets up the SMI handler in the SMRAM.* a trustworthy BIOS contributes to mitigate attacks against the SMM-based security tool before it is deployed in the SMRAM. In fact, many BIOS manufacturers implement some sort of authentication when updating their BIOS [143, 153].

- ***a3* - Trustworthy chipset.** *The chipset in the target platform is not malicious and cannot be tampered with.* We assume that the chipset will work always in a trustworthy way and attackers cannot tamper with firmware and other components in the chipset.

4.4 Requirements for using SMM for security purposes

Based on what was investigated up to now, some remarks about SMM and its resources are possible. Thus, SMM is: chipset specific, platform specific, OEM specific, executive software specific, limited in execution time (however, there is no mechanism to pre-empt it), limited available space for code and data, high privileged, important to manage the system, code and data non-persistent and real-address mode.

Then, using SMM for security purposes means to use its components to perform tasks, which they were not designed for, according to the Intel's manuals [66, 71, 87, 91, 92, 93, 97], as the previous works did. Because of this, all reported works exhibit some limitation. To take more advantage from SMM, an SMM-based security tool needs to deal with the limits and constraints imposed by SMM. Then, this research establish a set of requirements which must be met by an $SBST$ to optimise it for the SMM.

This research establishes a set of requirements for an $SBST$, considering the threat model, the assumptions, the architecture and analysis of security tools presented and the analysis of attacks reported in this research. These requirements aim to overcome the limits and constraints of SMM and allow an SMM-Based security tools take more advantage from SMM resources. Therefore, an SMM-based security tool must be **small, fast, persistent, cooperative, isolated, resistant, SMI-independent and complete**, as discussed ahead.

- ***r1* - Small.** There are 32512 bytes available for the SMI handler code, data, heap and stack (section 3.2.2). In fact, SMRAM can be up to 4 Gbytes, but it is recommended to keep the minimum size [87]. Observing HyperCheck [141], HyperSentry [14], auditing tool [61], AppCheck [142], MUSHI [151], SPECTRE [149] and IOCheck [150], we notice that those tools use a similar architecture, taking in account a remote machine to analyse the collected data and for other management functions. That architecture addresses the issue of limited space, modularising the tools. Although this remedies the problem, it opens new venues of attack, as the communication channel, the drivers and devices used to enable communication and as the remote machine. Moreover, they enlarge TCB, since all those items should be in the TCB, so they lose the SMM isolation protection by using other modules of the tool outside of SMRAM. Thus, a SMM-based security application should be small enough to fit in the minimum size available.

- ***r2* - Fast.** Intel specifies that the SMI latency must be less than 150 μs to minimise the risk of system executive software time-outs [90]. The modularised architectures of HyperCheck [141], HyperSentry [14], auditing tool [61], AppCheck [142], MUSHI [151], SPECTRE [149] and IOCheck [150] deals with this problem by leaving just a part of their tool in SMRAM. This architecture minimises the time spent executing the tool when in SMM. Anyway, the execution time reported when in SMM is 35 ms for HyperSentry and 40 ms for HyperCheck, which is much greater than the time specified by Intel.

- ***r3* - Persistent.** The SMRAM is volatile. A reboot or system restart will clean the whole SMRAM content. So, the SMM related code and data need to be loaded again. Thus, the design of a SMM-based security tool needs to consider that the tool must be embedded in the BIOS (or equivalent entity), since the BIOS contains the SMM related code and data and loads them into the SMRAM during the boot up process (figure 3.9) [57].

- ***r4* - Cooperative.** The SMI handler functions need to be preserved since they have important tasks to perform. The SMM-based security tools reported in section 3.4 use modified versions of the SMI handler. But the implications and the extension of modifications to the original SMI handler code are not specified. So, the SMI handler integrity cannot be guaranteed. Any SMM-based security tool must preserve the original SMI handler functions, by adding its own code to the SMI handler and not overwriting any part of it. Since when entering SMM the processor looks for the first instruction to be executed at the address SMBASE + 8000H (by default 38000H) in SMRAM, where the SMI handler is located, this implies that any SMM-based security tool must be a modified version of the SMI handler.

- ***r5* - Isolated.** SMI handler, and consequently any SMM-based security tool, performs its tasks without notifying or being recognised by the system executive software [57]. Moreover, since the system executive software stands still during the whole time the processor is in SMM, its execution is transparent for other software in the system. Then SMM-based security application using

memory outside of SMRAM is sort of counter-intuitive, since the main motivation to use SMM is to benefit from its powerful resources, such as isolation and transparency. Moreover code and data outside of SMRAM can be tampered. Then, a SMM-based security tool needs to be protected by isolation and its code and data, even temporary, should be kept in the SMRAM.

- ***r6* - Resistant.** The "cache poisoning" attack is an effective attack against SMM [46, 148]. This attack is possible by manipulating MTRR registers to make SMRAM cacheable. To thwart this attack, the SMRR Interface should be used to protect the related MTRRs registers (section 3.2.3). Then, any resource in the system available to reinforce security in SMM must be compulsorily used.

- ***r7* - SMI-independent.** To start any SMM-based security tool, an SMI needs to be generated. A common approach to trigger an SMI to start such a software is writing to the Programmed I/O Port 0xB2H [23, 45]. Since this is a well-known approach an attack might aim to thwart such action by identifying the code signature [142] and then denying the use of that port. Thus, a SMM-based security tool should take advantage from any SMI generated to start executing its job. Conversely, whenever the tool needs to start, it should be able to use different ways to trigger an SMI, considering the APIC table, which contains the SMI trigger events.

- ***r8* - Complete.** As discussed in requirement *r1* and *r5*, HyperCheck [141], HyperSentry [14], auditing tool [61], AppCheck [142], MUSHI [151], SPECTRE [149] and IOCheck [150] need a remote machine to analyse the collected data and for other management functions. Some tools need to keep part of their code in the system executive software. For example, HyperSentry uses an agent deployed in the hypervisor code base. While that model of architecture overcomes some SMM limitations, they lose the main benefits from using SMM as: isolation and transparency. Then, the SMM-based security tool must have all functionalities to execute its tasks and all needed data completely deployed in the SMRAM.

4.5 Discussion

In this chapter, we built a threat model to understand the threats to the current $SBST$ architecture. Then, we made assumptions and established a set of requirements which an $SBST$ must meet to overcome the limitations and constraints of SMM and to improve the security of an $SBST$. By meeting the requirements, those tools can take better advantage from SMM resources, ensuring strong isolation, transparency, high privileges and good view of the system.

The requirements were established to deal with the limitations and constraints of SMM and to mitigate the threats identified in the threat model (figure 4.1). Thus, we can classify requirements $r1$, $r2$, $r3$, $r4$ and $r7$ (small, fast, persistent, cooperative and SMI-independent, respectively) as functional requirements, since they are related to functioning of $SBST$ and must be met to overcome the limitations and constraints of SMM. Requirements $r5$, $r6$ and $r8$ (isolated, resistant and complete,

respectively) can be classified as security requirements, since they must be met to mitigate the threats $SBST$s are prone to. In the same direction, the assumptions were made considering the complexity and time frame available for this work, since it is neither possible nor convenient tackle all threats identified. Therefore, figure 4.2 indicates the requirement or assumption mitigating a specific threat.

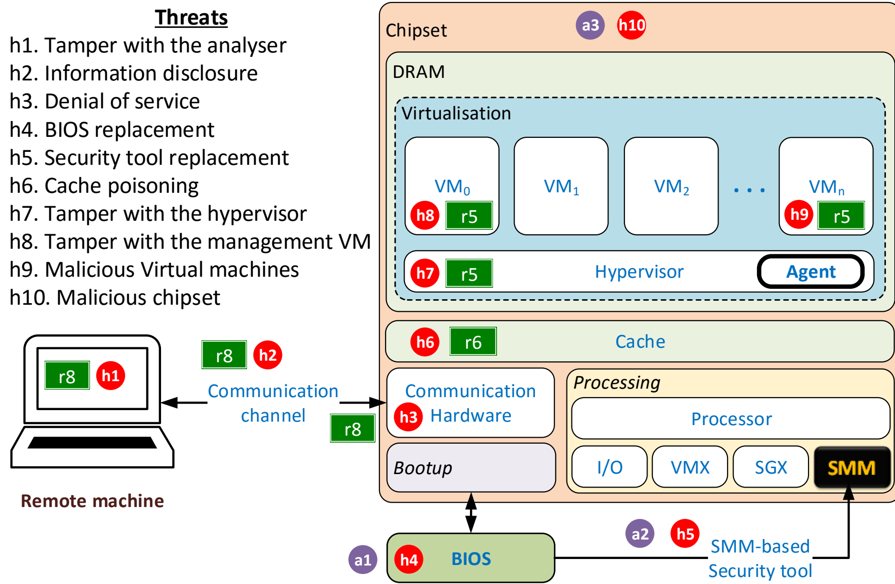

Figure 4.2: **Mitigating threats.** Threats $h4$, $h5$ and $h10$ are eliminated by the assumptions made and threats $h1$, $h2$, $h3$, $h6$, $h7$, $h8$ and $h9$ are mitigate by the requirements $r5$, $r6$ and $r8$.

Assumption $a1$ (no physical access) eliminates threat $h4$ (BIOS replacement), since an attacker would not have access to the target machine. Assumption $a2$ (trustworthy BIOS) eliminates threat $h5$ (Security tool replacement), since $SBST$ would be uploaded in the mem_{SMRAM} by a trustworthy BIOS; and assumption $a3$ (trustworthy chipset) eliminates threat $h10$ (malicious chipset), since the chipset would be trustworthy.

Requirement $r5$ (isolated) mitigates threats $h7$ (Tamper with the hypervisor), $h8$ (Tamper with the management VM) and $h9$ (Malicious Virtual machines), since by deploying the $SBST$ in the mem_{SMRAM}, its code is unreachable by code outside mem_{SMRAM}.

Requirement $r6$ (resistant) mitigates threat $h6$ (Cache poisoning), since by setting the appropriated values in MTRR registers we can make mem_{SMRAM} uncacheable, avoiding any kind of attack related to the cache memory while in SMM.

Requirement $r8$ (complete) mitigates threats $h1$ (Tamper with the analyser), $h2$ (Information disclosure) and $h3$ (Denial of service), since by having all $SBST$ functionalities laid out in mem_{SMRAM}, we do not need neither the communication hardware and channel nor the remote analyser machine.

Requirement $r1$ (small) does not directly mitigate threats, but it contributes to security by diminishing the TCB.

An $SBST$ may require characteristics and capacities to work properly and to enforce security, which are not mandatory as requirements but can be desirable in the tool. For instance, the characteristic of being "unique" in the SMRAM. This means that a $SBST$ would need to prevent another SMM-based code (except SMI handler) to be deployed in the SMRAM, controlling the SMRAM space and registers related to SMM, as those ones related to the SMRR Interface to limit cacheable addresses references to the SMRAM and to restrict access to the memory address range of SMI handler code and data in the SMRAM (section 3.2.3).

A desirable capacity is "self-cleaning". For example, the Intel Trusted Execution Technology (Intel TXT) (section 2.4.3) provides secured system start (or restart at any time), such that the system executive software (Measured Launched Environment (MLE) in Intel TXT context) can be loaded in a trustworthy way. The capacity of launching a restart at any time is called **late launch** [57]. A $SBST$ may be endowed with the capacity to perform a "late launch", a system restart or similar operation to load a pristine version of itself, if something goes wrong, as in the case of the attack called "Memory Sinkhole", which exploits a flaw in the CPU designed to compromise SMRAM [43] or when an "SMM Handler Code Access Violation", is detected in one of the **IA32_MCi_STATUS** registers, which means that "an attempt was made by the SMM Handler to execute outside the ranges specified by SMRR" (see section A.1.1 in this research and section 15.9 and 16 in [87, 97]). We are not assuming that a target machine is endowed with Intel TXT capabilities. However, if the target machine has such a capability, it may be used.

Those characteristics and capacities are considered supplementary and they are just discussed in this work, as they are considered in the architecture design discussion. However, they are not implemented in our proof of concept.

4.6 Summary

The threat model designed in this chapter contributes to establish the requirements, since some requirements deal with security issues and other with functional ones. The threat model pointed out some issues, which might be subject of a complete new research, as the issue of a malicious chipset. So, we made assumptions to address those issues which are neither possible nor convenient to be addressed in this research.

In the same way, the characteristics and capacities an $SBST$ might have, besides the requirements, can be the subject of a completely new research. For example, the characteristic of being "unique" can be very difficult to implement, since it implies the capacity of identifying executable code in the SMRAM and block those "other" codes to be written to the SMRAM. Also, the feature of being "self-cleaning" is also difficult to implement, since that implies simple actions as check some registers (as the **IA32_MCi_STATUS** registers), but also complex ones as the capacity of self-measurement and measure the SMRAM space.

SMM has limitations and constraints which have imposed severe performance and security penalties to SMM-based security tools. However, by investigating those tools, attacks, chipsets and the SMM components and operation, we were able to build a threat model and make assumptions to establish a set of requirements,

which if met, can ensure, strong isolation, transparency, high privileges and good view of the system.

In this chapter, we built a threat model to the current $SBST$ architecture and successful established a set of requirements that should be met by an $SBST$. Then, in the next chapter we will specify each requirement, detailing how they can contribute to build a general architecture for $SBST$s. We will propose a general architecture and then design our answer to the research questions, step by step, by inserting one requirement at a time in such an $SBST$ general architecture.

Chapter 5

A Generic Architecture for SMM-Based Security Tools

5.1 Introduction

The previous chapter built the threat model to the current $SBST$ architecture, considering the findings in chapters 2 and 3. Then, assumptions were made to deal with the complexity of the problem and the time available. After that, we establish a set of requirement, which must be met to overcome the limitations and constraints of SMM and to mitigate the threats identified in our threat model described in chapter 4.

This chapter specifies the requirements detailing how they can be built to fit in the general architecture proposed. Then, we propose that general architecture, offering a global view of our solution. After, we design the proposed architecture to answer the research questions (section 1.4). The design is made step by step, by inserting the requirements in the $SBST$ general architecture. Finally, we specify the algorithms to implement the $SBST$ functions by means of a payload study case.

This chapter is organised in three content sections and a discussion and summary section. First section specifies and discusses the requirements. In the second section, we present the general architecture (figure 5.1) to answer the research questions proposed in section 1.4 and the third questions design the general architecture and specify the algorithms to implement the $SBST$.

5.2 Requirements Specification

In this section, we analyse the requirements and elaborate how they can be specified to build the generic architecture for an $SBST$.

Thus, considering: the definitions in section 2.2; $ml = 150\mu s$; $ms = 32512$ bytes; and $R = \{r_1, r_2, ..., r_8\}$ the requirements described in section 4.4. To meet the set of requirements R, $SBST$ should be built in an architecture, as follows:

5.2.1 Requirement r1: SMALL

Specification: $|SH| + |bc| + |p_i| + |D| <= ms$.

Discussion. The solution to make $SBST$ small is to divide it in two parts: a basic code as defined in definition 2.4 and a payload as defined in definition 2.9. Then, $SBST$ comprises of a bc (definition 2.4) to deal with management functions, a payload p_i (definition 2.9) to perform a well-defined task and set of data $|D|$. $SBST$ should be small and flexible to receive a different payload p_{i+1} to perform

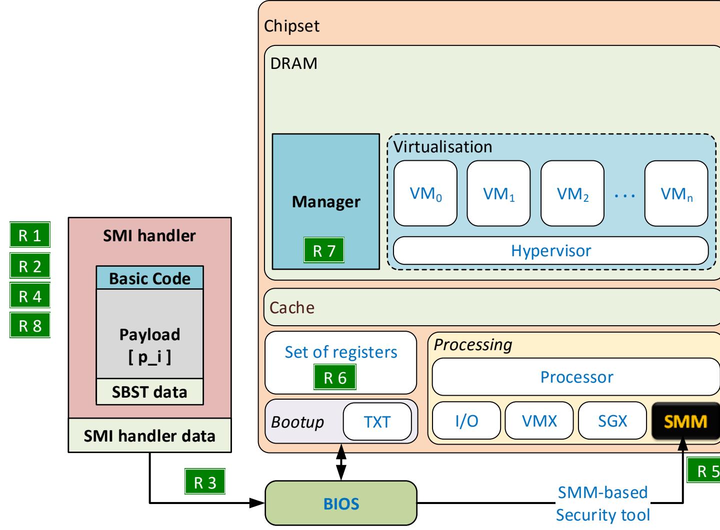

Figure 5.1: **General architecture.** The requirements are met by implementing an $SBST$, deploying it in the BIOS and uploading it in the SMRAM. A second software artefact is implemented: a **Manager** that contribute to meet the requirements and to probe and to learn about the target platform. Note that the **Manager** does not need to reside in the target machine.

a different task as often as required. So, the summation of the size of: SH, bc, p_i and D must be smaller than ms. We have included SH (definition 2.13) due to the strategy chosen to meet the requirement $r4$.

5.2.2 Requirement r2: FAST

Specification: $te(SBST : X \rightarrow Y) < ml$.

Discussion. The $SBST$ execution time must be smaller than ml, with $ml = 150\mu s$, whenever transforming some X in Y. For example, supposing the $SBST$ is loaded with a payload p_i to measure the integrity of the Xen hypervisor essential data [18], what can be an simple file, as the file used in this research: **xend-config.sxp**. So, when measuring that file (X) to produce a measurement value (Y), the total time spent on that should be less than ml. The use of payloads contributes to reduce the execution time. Considering that each p_i executes a task t_i and t_i is divided in subtasks $t_{i1}, t_{i2}, ..., t_{nm}$, to avoid exceeding the maximum latency limit, each time $SBST$ starts by an $smi_i \in SMI$, it performs a subtask t_{ij}, saves the result in the mem_{SMRAM} and finishes its execution. Next time $SBST$ starts, it performs another subtask t_{ij+1} and must concatenate the result with the previous one in mem_{SMRAM}. $SBST$ may keep this procedure up to complete a whole task t_i. The algorithm 5.1 describes how to meet the requirement $r2$.

5.2.3 Requirement r3: PERSISTENT

Specification: $mem_{BIOS}[i] \leftarrow SBST$.

Discussion. $SBST$ must be deployed in the mem_{BIOS}, considering requirement $r4$. Every bootup process in the target machine will load the $SBST$ into $mem_{SMRAM}[i]$. Essential data in D is loaded from mem_{BIOS} during the bootup. Other data must persist among $SBST$ executions, during the time the target machine is executing, to complete a task as described in requirement $r2$ and in the algorithm 5.1.

5.2.4 Requirement r4: COOPERATIVE

Specification: $SH \leftarrow SH \cup SBST \cup bc \cup p_i \cup D$.

Discussion. SH should be modified to embed the bc, p_i and D, in such a way that all previous functions of SH remains integral. Then, when any smi_i (requirement $r7$) trigger SH the original code portion is executed and, after that, it passes control to the instructions implement bc and p_i.

5.2.5 Requirement r5: ISOLATED

Specification: $mem_{SMRAM}[i] \leftarrow SBST \equiv mem_{SMRAM}[i] \leftarrow SH;$

Discussion. The $SBST$ must be restricted to the mem_{SMRAM}. After meeting requirement $r4$, $SBST \equiv SH$. The whole set D must remain in the mem_{SMRAM}, no part, even temporally, can be stored outside mem_{SMRAM}.

5.2.6 Requirement r6: RESISTANT

Specification: $SBST : REG_x \rightarrow REG_y$

Discussion. $SBST$ must have control of the REG and set the appropriate values, whenever they are not set by the chipset, such as enforcing the use of SMRR interface. It noteworthy that some functions in the chipset may set values in the registers without $SBST$ control, as discussed in section 3.2.3. In some situations $SBST$ can change the values of some registers and in other it cannot be changed, as, once bit SMI_LOCK (bit 0) in the SMI_EN register is set, it cannot be cleared.

5.2.7 Requirement r7: SMI-INDEPENDENT

Specification: $bc_{smi_i} : SBST_x \rightarrow SBST_y$.

Discussion. bc is designed to start at any $smi_i \in SMI$, which take $SBST$ from a status x to a status y, where x is the stopped status and y is a continuous execution status, as describe in the requirement $r2$.

5.2.8 Requirement r8: COMPLETE

Specification: $\forall p_i \in P, p_i \in SBST$ and $SBST \subset mem_{SMRAM}$

Discussion. All functions designed to be performed by an SMM-based security tool must be implemented in the $SBST$ and it must be deployed in the mem_{SMRAM} after the bootup process. $SBST$ must not depending onto any code or data outside of mem_{SMRAM}. Note that after meeting requirement $r4$, $SBST \equiv SH$.

5.3 General Architecture

Based on the previous section, in this section we propose a generic architecture as described into the figure 5.1. According to the figure: requirements $r1$, $r2$ and $r4$ are met by coding them in the $SBST$ as specified in the last section; requirement $r3$ is met by inserting $SBST$ in the mem_{BIOS}; requirement $r5$ is met by uploading $SBST$ from mem_{BIOS} to mem_{SMRAM}; requirement $r6$ can also be codified in the $SBST$ since it is about to handle registers values; requirement $r7$ is met by coding it in the $SBST$ and by using another software artefact called "Manager", which contributes to discover the event that can trigger an SMI in the target platform. The manager resides in the DRAM, but it is not necessarily in the target machine. So, there is not isolation problem in this concept; and requirement $r8$ is met by designing $SBST$ to be a monolithic code, deployed into mem_{SMRAM}.

The functional requirements $r1$, $r2$, $r3$, $r4$ and $r7$ (small, fast, persistent, cooperative and SMI-independent, respectively) are implemented as follow: requirements $r1$ (small) and $r2$ (fast) are met by implementation techniques and should be verified to not allow an $SBST$ bigger than the size limit, and controlled during the execution time to pre-empt the tool whenever the time limit is reached. Requirement $r3$ (persistent) is met by embedding the $SBST$ in the mem_{BIOS}. Requirement $r4$ (cooperative) is met by inserting the basic code bc and a payload p_i code in the SMI handler code, without disable or override any SMI handler function. Requirement $r7$ (SMI-independent) is met by designing and implementing the tool to trigger the $SBST$ whenever an SMI occur in the system and by using another software artefact called "Manager", which contributes to discover the event that can trigger an SMI in the target platform.

The security requirements $r5$, $r6$ and $r10$ (isolated, resistant and complete, respectively) are implemented as follow: requirement $r5$ (isolated) is met by uploading the whole $SBST$ code and data from the mem_{BIOS} to the mem_{SMRAM}. Requirement $r6$ (resistant) is met by setting the appropriated registers in the set of SMM related registers (section 3.2.3) to use the SMRR interface. Requirement $r8$ (complete) is met by implement $SBTS$ as a monolithic software artefact, which does not take in account any other code out of SMRAM.

5.4 Architecture Design

In this section we design the architecture, adding step by step the requirements to the architecture. As presented in figure 5.1, the architecture consists of two software artefacts: an agent (the $SBST$), designed to perform a security task, consisting of a basic code bc and a payload p_i, where some steps are implemented into the bc, other in the p_i and other considering the whole $SBST$, as described ahead. As discussed in the definition 2.4, bc exists to perform tasks which are independent of the payload loaded into the $SBST$; and a manager module to help the payload developer to understand the particularities of a target chipset. The manager module is used to probe and research a target machine and then understand the capabilities and limitations of such a machine, so it must be laid out in DRAM. The machine probed and research is not necessarily the target machine, but it must have the same chipset family or a similar chipset. Then, there is not isolation problem caused by

the manager module deployment.

The index i used in the algorithms in this chapter represents any position in the memory unit considered, respecting and preserving the previous occupied memory indexes. For example, for $mem_{SMRAM}[i]$, the index i can be any value between SMBASE + 8000H and SMBASE + 8000H + FFFFH, respecting and preserving the memory space of SMI handler and state saved area (figures 3.2 and 3.3).

5.4.1 Step 1: $SBST$ Small and Cooperative

Requirements $r1$ and $r4$ are met in step 1, by making $SBST$ small and cooperative. Realise this is a matter of how $SBST$ architecture is designed and how it is implemented. This step in implemented considering the whole $SBST$.

We designed the $SBST$ architecture to use payloads, where each payload realises a well-defined task (definitions 2.6 and 2.9). This architecture contributes to meet most of the requirements and allow flexibility to the security tool designer who might change the purpose of the tool by replacing payloads. It also contributes to make the tool faster since it just execute one well-defined function.

When planning the $SBST$, it is required to plan the functions into bc and payloads in P to keep the tool under the size limit discussed. Also, it is necessary to find an SMI handler code small enough to embed the bc and a payload p_i, while keeping $SBTS <= ms$ (section 5.2.1) and preserving the SMI handler original code (section 5.2.4). Note that SMI handler implementation depends on the manufacturer, consequently its size too and, more important yet, an SMI handler implementation is chipset specific. The general approach in this research is to use an open source SMI handler, as that one released with Coreboot [36] (section 2.4.1). As we saw in section 3.2.4 and 2.4.1, we found our target SMI handler source code in the coreboot 4.4 set of source code in the file *smihandler.c* which is 9.97 Kbytes size and its executable code is estimated in 5.5 Kbytes. So, we have established that there are around 25 Kbytes available for the proof of concept of our $SBST$ code.

The bc can implement functions like those described in the algorithm 5.8 in step 5 5.4.5 that must be applied for any payload. A payload p_i may implement any well-defined task according to definition 2.9 provided that it not make the $SBST$ size surpasses the limit imposed. The total D is the summation of the SMI handler and bc and p_i data, if and when they exist. The size of SMI handler data depends on the OEM implementation and the bc and p_i data depends on the task considered, remembering that they need to obey our specification $|SH|+|bc|+|p_i|+|D| <= ms$.

5.4.2 Step 2: $SBST$ Fast, SMI-independent and Complete

Requirements $r2$, $r7$ and $r8$ are met in step 2. Those requirements are implementation dependent too. Therefore, we need a study case on a security task to help us designing this part of our generic architecture, because a payload implementation (its set of instructions) depends on the task to be performed, since a payload might execute any well-defined task (definitions 2.6 and 2.9). This step in implemented as a payload.

Let's assume that this well-defined task is: checking the integrity of some data related to a hypervisor, comprising of a unique table (in a file, as the **xend-config.sxp** file), and that it takes three subtasks (definitions 2.6 and 2.5) to be accomplished.

It is import to emphasize that this integrity check task is just to make our proof of concept feasible and to demonstrate it. So, we are not too concerned about the integrity check task itself, but in how to fit a security task in the proposed architecture by means of our proof of concept. So, to make $SBST$ fast, we design it to depend on the time elapsed when executing a subtask. So, the $SBST$ needs to execute a pre-emptive rsm instruction before the subtask reaches the time limit. However, since monitoring a subtask execution adds overhead to the elapsed time and it is required to count on another system component (e.g. a clock) to measure the time and it adds some complexity to the development of the $SBST$, we decided that the better approach in this case is to estimate the time needed to perform each subtask before implement it. In this way, we control the execution by job done and not by execution time, in that way, we assure that the time constraints are respected.

To deal with step 2, it is necessary to understand the features of the host machine where the $SBST$ will be deployed, as CPU family and memory technology, to discover the memory timing cycles to read and write information in the main memory and the CPU timing cycles to compute the required tasks. Cache memory will not be considered, since it is required to make memory uncacheable to meet requirement $r6$. It is also necessary to know the amount of data or code to read and write and the amount of computation needed. It is mandatory to sum up the time elapsed during the SH execution, according to the requirement $r4$.

The technological details discussed above are not imperative during the architecture design. Those issues are addressed in the next chapters. Then, we can use values of our experiments with SMI handler (section 3.2.4) and use approximated values in the other cases.

Let's consider our chipset 2 and assume the following example of configuration to make our algorithm simpler for designing and presenting the architecture:

- SH **execution time:** The SMI handler latency time for any management function is $69\mu s$ triggered after an SMI (according to table 3.1). So, any subtask can spend up to $81\mu s$ performing its instructions.

- **SHA_256( ):** A function implementing algorithm SHA-256 to compute a hash code [4, 50].

- **xend-config.sxp:** One of the configuration files of Xen hypervisor, where some important configurations are done. For instance, configure a IP address, enable migration and scripts to run, change memory configurations and so on. It is located in /etc/xen [137]. Let's consider this file has 12 Kbytes (it was a common size found for that file in our research).

- **LoadData( ):** A function able to read data from main memory and write it to the SMRAM at a rate of 200 bytes per $1\mu s$.

- **ComputeHash( ):** A function able to compute the hash for any code or data at a rate of 70 bytes per $1\mu s$.

- **VerifyHash( ):** A function able to compute the hash for any code or data at a rate of 70 bytes per $1\mu s$ and compare a pair of hash digest at a rate of $10\mu s$.

From the configuration above, we beginning to design the algorithms to meet $r2$, $r7$ e $r8$. We designed the $SBST$ to work non-stop, starting from the bootup process, when the tool is uploaded into the $mem_{SMRAM}[i]$ and after the first SMI occurs in the system, and finishing when the target machine is restarted, rebooted or switched off. However, when the time limit is reached, $SBST$ saves the state of its current execution and executes a rsm instruction. When a new SMI is triggered the $SBST$ recovers the **state saved** (figures 3.2 and 3.3) and restarts its execution. Then, to meet $r2$ (section 5.2.2) we control the execution of each subtask, focusing on the amount of data processed. To meet $r7$ (section 5.2.7) we design the algorithms to start executing by any SMI. To meet $r8$ (section 5.2.8) we design all functions in the bc and p_i to be embedded in a monolithic software artefact, laying it out in $mem_{SMRAM}[i]$ and not allowing it to count on any other software outside of SMRAM. Figure 5.2 presents the execution flow of the algorithms.

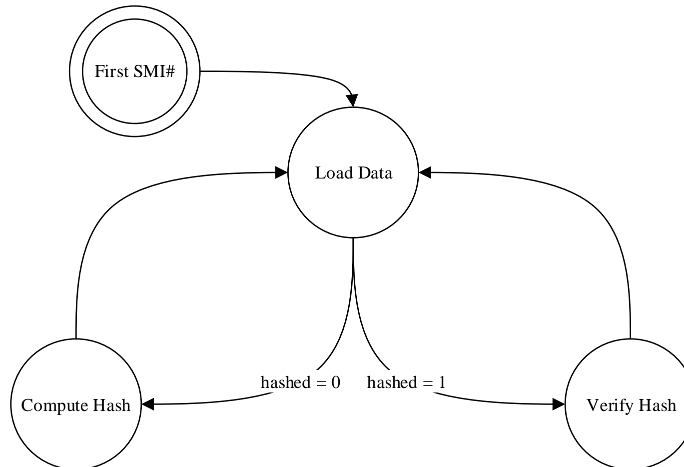

Figure 5.2: **Execution flow of the algorithms.** When the first SMI occurs, the $SBST$ calls function load data, then calling functions compute hash or verify hash, depending on the value of "hashed". It works non-stop from that point on.

Algorithm 5.1 represents the main functions and calls the other algorithms to perform the integrity measurement. Thereby, the $SBST$ should work in a continuous base, in the sequence: **load data** (algorithms 5.2 and 5.3), **compute hash** (algorithms 5.4 and 5.5), **verify hash** (algorithms 5.6 and 5.7), according to figure 5.3. Then, considering our execution times, file **xend-config.sxp** can be copied into $mem_{SMRAM}[i]$ in one $SBST$ invocation, but it requires three invocations to measure and three invocation to remeasure the file. In this way, the file in our study case is measured and remeasured in two data sections of 5670 bytes and one data section of 948 bytes, one by one, totalising 12 Kbytes. So, the $SBTS$ try to identify and report inconsistencies section by section. To recap, the $SBST$ invocation is done every time an smi_i is triggered.

Algorithm 5.1 has two constants: $DATA_BY_ROUND$ = 16200 and $HASH_BY_ROUND$ = 5670; those constants reflect the previous information assumed

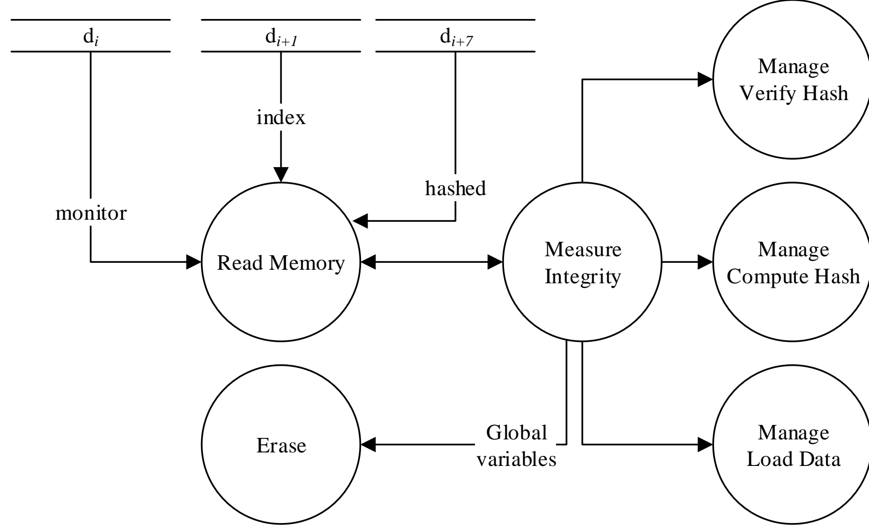

Figure 5.3: **Algorithm Measure Integrity.** This function represents a payload to measure the integrity of a file. It calls functions LoadData( ), ComputeHash( ) and VerifyHash( ) to measure the integrity and ReadMemory( ) during the execution. In the end, all variables are erased to clean the $mem_{DRAM}[i]$

that the functions to integrity check are able to deal with 16200 bytes and 5670 bytes size before reach the time limit. It means that at each round (definition 2.8), $SBST$ can deal with 16200 bytes of data, since the LoadData( ) function can read data from $mem_{DRAM}[i]$ and write it to the $mem_{SMRAM}[i]$ at a rate of 200 bytes per $1\mu s$ or 16200 bytes per $81\mu s$; and the ComputeHash( ) function can compute the hash for any code or data at a rate of 70 bytes per $1\mu s$ or 5670 bytes per $81\mu s$.

For the efficiency and simplicity sake, all variables are global. Thus, algorithm 5.1 holds all variables used in the integrity check process. All variables, their type and persistence are discussed below (tables 5.1 and 5.2).

The **integer** variables are: *size*, non-persistent and containing the amount of data remaining to be loaded; *round*, non-persistent and indicating the amount of rounds required yet; *previous_round*, persistent (stored in d_{i+2}) and indicating the order of last round performed; *monitor*, persistent (stored in d_i) and indicating which function is called in a certain point; *index*, persistent (stored in d_{i+1}) and indicating the position in $mem_{SMRAM}[i]$ where the data read from $mem_{DRAM}[i]$ will be stored; *hashed*, persistent (stored in d_{i+7}) and indicating if the data is already hashed or not. If $hashed == 0$ the data in $mem_{SMRAM}[i]$ was not hashed or it was verified yet and the execution flow can go through the hash computing process. If $hashed == 1$ the data in $mem_{SMRAM}[i]$ was hashed; *size_section* indicates the total of data already loaded in a round. This variable is only persistent in algorithm 5.5, where it is stored in $mem_{SMRAM}[d_{i+6}]$;

The **long** variables are: *address*, persistent (stored in d_{i+3}) and storing the read memory address in the current round; *previous_address*, non-persistent and storing

Algorithm 5.1: Measure the integrity of hypervisor dynamic data. This is the main algorithm and calls functions: LoadData( ), ComputeHash( ) and VerifyHash( ); to perform integrity measurement, according to the variable $monitor$ value.

Input : Table xend-config.sxp.
Output: The integrity measurement of file xend-config.sxp.

1 constant integer $DATA_BY_ROUND = 16200$;
2 constant integer $HASH_BY_ROUND = 5670$;
3 integer $size, round, previous_round, monitor, index, hashed, size_section$;
4 long $address, previous_address$;
5 string $data, total_data, digest, stored_digest$;
6 $Function \quad Main$
7 **Begin**
8 $monitor \leftarrow ReadMemory(mem_{SMRAM}[d_i])$;
9 $index \leftarrow ReadMemory(mem_{SMRAM}[d_{i+1}])$;
10 $hashed \leftarrow ReadMemory(mem_{SMRAM}[d_{i+7}])$;
11 **if** $monitor == NULL$ **then**
12 | $monitor \leftarrow 0$;
13 **if** $index == NULL$ **then**
14 | $index \leftarrow 0$;
15 **if** $hashed == NULL$ **then**
16 | $hashed \leftarrow 0$;
17 **if** $monitor == 0$ **then**
18 | ManageLoadData( );
19 **else if** $monitor == 1$ **then**
20 | ManageComputeHash( );
21 **else if** $monitor == 2$ **then**
22 | ManageVerifyHash( );
23 Erase($size, round, previous_round, monitor, index, hashed, size_section,$ $address, previous_address, data, total_data, digest, stored_digest$);
24 rsm
25 **End-Main**

the read memory address from the previous round.

The **string** variables are: *data*, non-persistent and storing the read data from memory, either $mem_{DRAM}[i]$ or $mem_{SMRAM}[i]$, during one loop iteration. *total_data*, non-persistent and concatenating the data variable content, during all loop iterations. *digest* is persistent (stored in d_{i+5}) and stores the hashes computed by data section; *stored_digest* is non-persistent and stores a hash read from mem_{SMRAM} $[d_{i+5}]$, during the hash verification process.

Since each time an smi_i is triggered that algorithm starts and it may not be able to accomplish its task in only one round (by reading 16200 bytes or by computing hash for 5670 bytes), those variables are demanded to persist in $mem_{SMRAM}[i]$. So, the variable $monitor$ is stored in d_i and the variable $index$ is stored in d_{i+1}, according to definition 2.11 and in compliance with requirement $r3$. The variable d_i represents the data stored into $mem_{SMRAM}[i]$, in the index i, and variable d_{i+1}

Table 5.1: **Persistant variables.** These variables must persist along the $SBST$ life cycle. To comply with the established requirements, they must be stored in the $mem_{SMRAM}[i]$. The index i indicates the first free position in $mem_{SMRAM}[i]$ to be used for storing the persistant variables. So $i + n$, indicates the next ones, where n, indicates a number between 1 and 7.

Location	Variable	Description
d_i	$monitor$	Which function should be called
d_{i+1}	$index$	Where to store file data in $mem_{SMRAM}[i]$
d_{i+2}	$previous_round$	The last round
d_{i+3}	$address$	Memory read address in the current round
d_{i+4}	$total_data$	Data read from $mem_{DRAM}[i]$
d_{i+5}	$digest$	The hashes already computed
d_{i+6}	$size_section$	Total of data already loaded
d_{i+7}	$hashed$	Indicating if data was hashed or not

Table 5.2: **Non-persistant variables.** These variables are non-persistant, so their content can be lost among $SBST$ calls. However, to comply with the established requirements, they must be erased before getting out of SMM.

Variable	Description
size	The amount of data remaining to be loaded
round	The amount of rounds required yet
previous_address	The read memory address from the previous round
data	The read data from memory during one loop iteration
total_data	The data concatenated during all loop iterations
stored_digest	A hash read from $mem_{SMRAM}[d_{i+5}]$

represents the data stored into $mem_{SMRAM}[i]$, in the index i plus the size of the data stored into d_i. From this point on, all variables d_i or d_{i+j} will follow that previous scheme. The algorithm call functions to load data, compute hash or verify hash, according to the $monitor$ value.

The control structure chose to control the algorithm execution flow imposes one or another kind of penalty to the $SBST$ execution: cross the time limit or spend more rounds to complete a task. For example, consider the nested **If** control structure like below:

if $monitor == NULL$ **then**
  $monitor \leftarrow 0$
if $monitor == 0$ **then**
  ManageLoadData($index$)
if $monitor == 1$ **then**
  ManageComputeHash($index$)
if $monitor == 2$ **then**
  ManageVerifyHash($index$)

The nested **if** above would make the $SBST$ to **cross the time limit** in at least two cases: when the variable $round == 0$ in algorithms 5.2 and 5.4, those algorithms update the variable $monitor$ value, clean values in $mem_{SMRAM}[i]$ and fi-

nalise their execution, passing control to algorithm 5.1. Then, the execution flow continues, calling ManageComputeHash($index$) or ManageVerifyHash($index$), according to the case. But, some time will have passed and these functions will have less time then what was planned.

Now, let's consider the nested **If** control structure like below:

if $monitor == NULL$ **then**
    $monitor \leftarrow 0$
else if $monitor == 0$ **then**
    ManageLoadData($index$)
else if $monitor == 1$ **then**
    ManageComputeHash($index$)
else if $monitor == 2$ **then**
    ManageVerifyHash($index$)

In this case, when the same situation as above happens and the execution flow continues in algorithm 5.1, it will call function *Erase(list variable)* and the rsm instruction. Thus, one round is used just for transition between functions and **spend one more round to complete a task**.

The integer variables $size, round, previous_round$ are global variables and they are used by all algorithms called by algorithm Algorithm 5.1.

Function *Erase(list variable)* erases variables created in the the algorithm, before leaving the SMM.

Algorithm 5.2 prepares the environment to call function LoadData( ) (algorithm 5.4). Information from the previous rounds should persist along all executions of $SBST$. For example, d_{i+2} persists information about the current round of the load data process. To control de number of rounds, we use the equation $round \leftarrow \lceil size/DATA_BY_ROUND \rceil - previous_round$, which gets the file size, divide it by the maximum data size by round ($DATA_BY_ROUND$), rounding (ceiling) the result, which is subtracted by the previous round. This equation will work even if *xend-config.sxp* size increase or decrease.

If $round == 0$ it means that no more rounds are needed. So, the algorithm passes control to the next function, ComputeHash( ), by writing 1 to the $monitor$ (d_i) and clearing: the index (d_{i+1}), the previous round (d_{i+2}) and the previous address (d_{i+3}); by writing 0 to them. The previous address (d_{i+3}) stores the reading memory position from the previous round during load data process. All those data are located in $mem_{SMRAM}[i]$, as demanding by requirement $r5$.

If more rounds are required yet, it calls LoadData( ), passing $index$ to the function; and, then, it increments $index$, to position the correct place where the next data blocked will be stored into $mem_{SMRAM}[i]$ and $previous_round$, since a new round was completed. Finally, the functions write $index$ and $previous_round$ to $mem_{SMRAM}[i]$.

Function Increment( ) adds a new unit to the memory space considered, according to the variable type.

Although $previous_address$ is not explicitly updated in the algorithm 5.2 pseudocode, the value of $previous_address$ is updated during the execution of this algorithm. Note that line 14 calls function LoadData( ) (algorithm 5.3) and that function reads from and writes to the variable d_{i+3}.

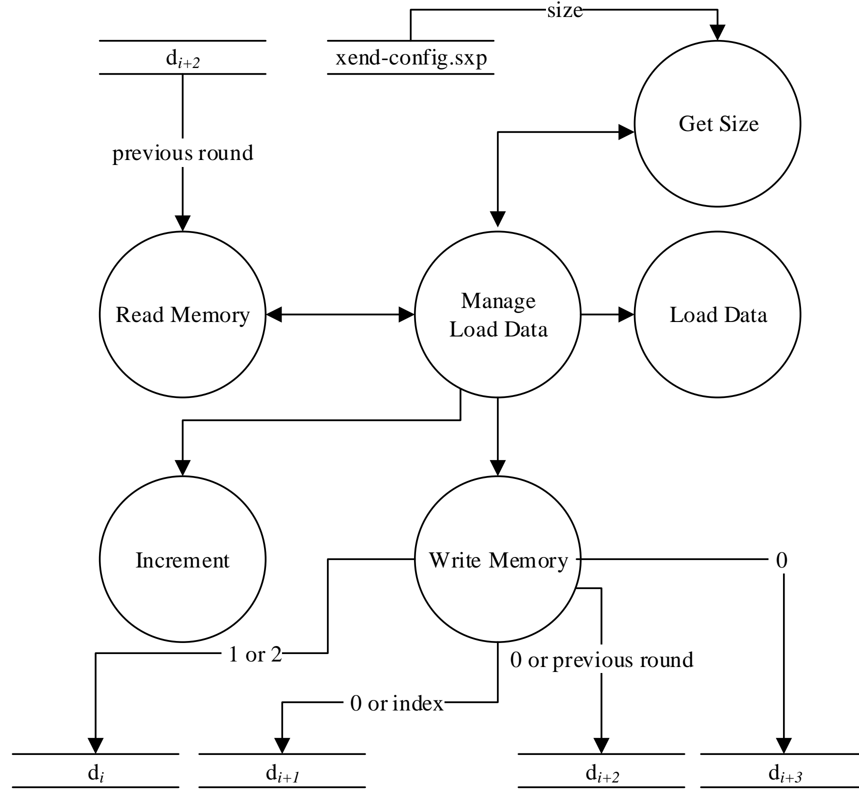

Figure 5.4: **Algorithm Manage Load Data.** This function prepares the environment to load data. Data is loaded by function LoadData( ). It reads from and writes to variables (d_{i+j}) located in $mem_{SMRAM}[i]$ to save the round context.

Function *ReadMemory( )* reads a value from a memory position. It has one parameter indicating the memory unit where the data will be written, with its respective index.

Function *GetSize(string file)* obtains the size of a file or table.

Function *WriteMemory(long place, data value)* writes a value to a memory position and has two parameters: first parameter is the memory unit where the data will be written. The address where the data will be written is indicated by the variable d_i. The $i + 1$ in the subscript of variable d indicates the address offset for the d variable. The second parameter is the value to be written. Note when implementing the $SBST$ that d_i must have enough space to store the type and amount of data required. For example, d_{i+4} must have enough space to store the whole *xend-config.sxp* file and d_{i+5} must have enough space to store all hashes computed.

Algorithm 5.3 describes the data loading process (Figure 5.5). If there is a previous address, it gets $previous_address$ from $mem_{SMRAM}[d_{i+3}]$ and increment it by the maximum data size by round to access the correct position to be read. It also gets the size of file *xend-config.sxp* and subtract that size from the size of the data

Algorithm 5.2: Manage to call function LoadData( ). It prepares the environment to call function LoadData( ). If no more rounds are needed, it saves the state and passes control to the next function. If rounds are required yet, it calls the LoadData( ) again.

Input : An integer $index$ indicating the next position in the $mem_{SMRAM}[i]$ to store data.
Output: Call function LoadData( ).

```
1  Function  ManageLoadData( )
2  Begin
3  previous_round ← ReadMemory(mem_SMRAM[d_{i+2}]);
4  if previous_round == NULL then
5  |  previous_round ← 0;
6  size ← GetSize(xend − config.sxp);
7  round ← ⌈size/DATA_BY_ROUND⌉ − previous_round;
8  if round == 0 then
9  |  WriteMemory(mem_SMRAM[d_{i+1}], 0);
10 |  WriteMemory(mem_SMRAM[d_{i+2}], 0);
11 |  WriteMemory(mem_SMRAM[d_{i+3}], 0);
12 |  if mem_SMRAM[d_{i+7}] == 0 then
13 |  |  WriteMemory(mem_SMRAM[d_i], 1);
14 |  else
15 |  |  WriteMemory(mem_SMRAM[d_i], 2);
16 else
17 |  LoadData( );
18 |  WriteMemory(mem_SMRAM[d_{i+1}], Increment(index));
19 |  WriteMemory(mem_SMRAM[d_{i+2}], Increment(previous_round));
20 End-ManageLoadData
```

already loaded in $mem_{SMRAM}[d_{i+4}]$, assigning the result to variable *size*.

If there is not a previous address, it means we are in the first round, so the algorithm obtains the file *xend-config.sxp* address in DRAM and assign it to variable *address*. It gets the size of file *xend-config.sxp*, assigning that to variable *size*. Then, a loop manages the loading process, which is controlled by $size_section <= DATA_BY_ROUND$, which means that the loop stops when the *size_section* reaches the maximum data size by round, and by $size_section <= size$, which means that the loop can stop if the file size is reached, even if the maximum data size by round has not been reached.

Inside the loop, it starts reading data from the file and store it in the variable *data* and accumulates in *size_section* the total loaded up to the point. The data read is concatenated in *total_data*. After the loop terminates, the read address is stored in d_{i+3}, which will become the $previous_address$ in the next round; and the *total_data* is stored into $mem_{SMRAM}[i]$ in the data structure $d_{i+4}[index]$, in the index indicated by $index$. Finally, the local variables created are erased.

Algorithm 5.3 has a new function called GetAddress( ), which obtain the memory address of a file or table.

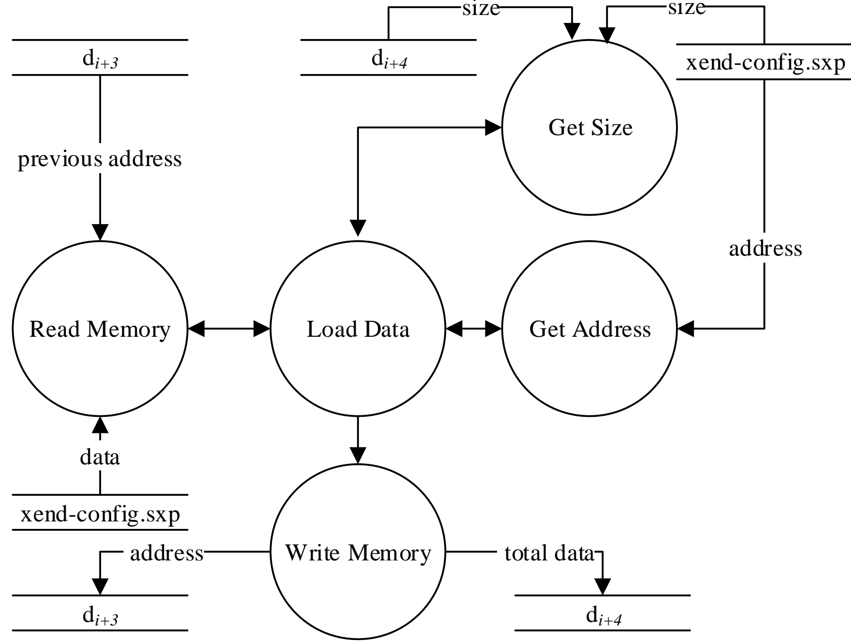

Figure 5.5: **Algorithm Load Data.** The function obtains the file address and from that, it reads the file, copying its content into $mem_{SMRAM}[d_{i+4}]$.

Algorithm 5.4 is similar to the algorithm 5.2. But now, it needs to get the size of d_{i+4} that is the data structure loaded from $mem_{DRAM}[i]$ into $mem_{SMRAM}[i]$ (figure 5.6). To control de number of rounds, we use the equation $round \leftarrow \lceil size/HASH_BY_ROUND \rceil - previous_round$, which gets the file size, divide it by the maximum data size that can be hashed by round ($HASH_BY_ROUND$), rounding (ceiling) the result, which is subtracted by the previous round.

When no more rounds are needed ($round == 0$), it passes control to the next function, VerifyHash( ), by writing 2 to the $monitor$ (d_i) and clears the index (d_{i+1}), the previous round (d_{i+2}), the previous address (d_{i+3}) and the total data loaded (d_{i+6}) by writing 0 to them. Here there is a difference: the ComputeHash( ) function stores the value of $total_data$, which will be hashed and represents the data get from $mem_{SMRAM}[d_i + 4]$. This value is stored in $mem_{SMRAM}[d_i + 6]$. If some round is required yet, it calls the ComputeHash( ), passing $index$ to the function; and incrementing $index$ and $previous_round$ and writing them into the $mem_{SMRAM}[i]$.

Algorithm 5.5 describes the process of get data from $mem_{SMRAM}[i]$, compute the digest and save that digest to the data structure $d_i + 5[index]$, using the $index$ received from the algorithm 5.4 (figure 5.7). If there is a previous address, it gets $previous_address$ from $mem_{SMRAM}[d_{i+3}]$ and increment it by the maximum data size that can be hashed by round to access the correct position to be read. It also gets the size of file stored in $mem_{SMRAM}[d_{i+4}]$ and subtract from the total of data

Algorithm 5.3: Load the data to be measured from DRAM to SMRAM. It loads data to be measured from DRAM to SMRAM, considering data previously loaded.

Input : An integer $index$ indicating the next position in the $mem_{SMRAM}[i]$ to store data.
Output: Data loaded.

1 $Function \quad LoadData(\quad)$
2 **Begin**
3 $previous_address \leftarrow ReadMemory(mem_{SMRAM}[d_{i+3}]);$
4 **if** $previous_address <> NULL$ **then**
5 $\quad address \leftarrow previous_address + DATA_BY_ROUND;$
6 $\quad size \leftarrow GetSize(xend - config.sxp) - GetSize(mem_{SMRAM}[d_{i+4}]);$
7 **else**
8 $\quad address \leftarrow GetAddress(xend - config.sxp);$
9 $\quad size \leftarrow GetSize(xend - config.sxp);$
10 $size_section \leftarrow 0;$
11 **while** $((size_section <= DATA_BY_ROUND)\&(size_section <= size))$ **do**
12 $\quad data \leftarrow ReadMemory(mem_{DRAM}[address]));$
13 $\quad size_section \leftarrow size_section + GetSize(data);$
14 $\quad total_data \leftarrow total_data + data;$
15 $WriteMemory(mem_{SMRAM}[d_{i+3}], address);$
16 $WriteMemory(mem_{SMRAM}[d_{i+4}[index]], total_data);$
17 **End-LoadData**

already hashed ($mem_{SMRAM}[d_{i+6}]$), assigning the result to variable *size* (global).

If there is not a previous address, it means we are in the first round of ComputeHash( ), so the algorithm gets $mem_{SMRAM}[d_{i+4}]$ address and assign it to variable *address*. It also gets the size of $mem_{SMRAM}[d_{i+4}]$, assigning that to variable *size*. Then, a loop manages the loading process, which is controlled by $size_section <= HASH_BY_ROUND$, which means that the loop stops when the *size_section* reaches the maximum data size that can be hashed by round, and by $size_section <= size$, which means that the loop can stop if the d_{i+4} size is reached, even if the maximum data size by round has not been reached yet.

Function *SHA-256(data)* is the SHA hash function [50], as discussed in section 2.5.

After the loop terminates, the data digest is computed by $SHA-256(total_data)$, the read address is stored in $mem_{SMRAM}[d_{i+3}]$, the digest is stored into $mem_{SMRAM}[d_{i+5}[index]]$ at the data structure $d_{i+5}[index]$, in the memory position indicated by $index$ and the total of data already hashed is stored in d_{i+6}.

Algorithm 5.6 is similar to the algorithm 5.4, but the data structure considered now is $mem_{SMRAM}[d_i + 5]$, where the hash digests are stored (figure 5.8). Another difference is when no more rounds are needed, it passes control to the function LoadData( ), by writing 0 to the $monitor$ (d_i), restarting the integrity check cycle. If it remains any round yet, it calls VerifyHash( ), passing $index$ to the function; after concluding the verify hash round, it increments $index$ and $previous_round$,

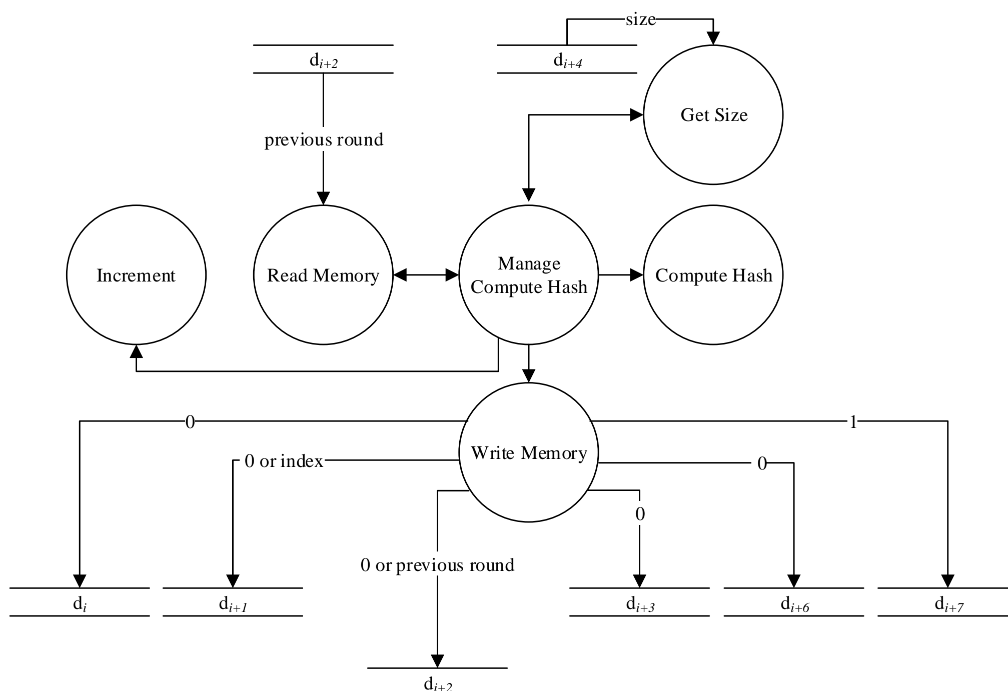

Figure 5.6: **Algorithm Manage Compute Hash.** This function prepares the environment to compute hash. Digest is computed by function ComputeHash( ). It reads from and writes to variables (d_{i+j} located in $mem_{SMRAM}[i]$) to save the round context.

writing them to $mem_{SMRAM}[i]$.

Algorithm 5.7 works similarly to algorithm 5.5. However, after the loop, it computes the hash of loaded data (in fact, the reloaded data), retrieves the respective stored digest by section and compare those two digests (figure 5.9). If they are equals, a message is displayed informing that it passes the integrity check for the data section indicated by $index$. If they are different, a message reports that the integrity of data section indicated by $index$ is compromised.

By executing continuously the above algorithms requirements $r2$, $r7$ and $r8$ are met.

5.4.3 Step 3: Persisting the $SBST$

This step aims to meet requirement $r3$ (persistent) specified in section 5.2.3. To persist $SBST$, it is embedded into the $mem_{BIOS}[i]$. Thereby, any time the target machine executes the bootup process, the *BIOS* uploads $SBST$ into $mem_{SMRAM}[i]$. An alternative to embed $SBST$ into $mem_{BIOS}[i]$ is to use the manager module in the target machine to copy the agent from $mem_{DRAM}[i]$ to $mem_{SMRAM}[i]$, where the follow situation described in section 3.2.3 occurs: most machines released about 2004 or before that have the bit D_LCK cleared and the bit D_OPEN is set after

Algorithm 5.4: Manage to call function ComputeHash( ). It prepares the environment to call function ComputeHash( ). If no more rounds are needed, it saves the state and passes control to the next function. If rounds are required yet, it calls the ComputeHash( ) again.

Input : An integer $index$ indicating the next position in the $mem_{SMRAM}[i]$ to store the hash digest.

Output: Call function ComputeHash( ).

1 $Function \quad ManageComputeHash(\ )$
2 **Begin**
3 $previous_round \leftarrow ReadMemory(mem_{SMRAM}[d_{i+2}]);$
4 **if** $previous_round == NULL$ **then**
5 | $previous_round \leftarrow 0;$
6 $size \leftarrow GetSize(mem_{SMRAM}[d_{i+4}]);$
7 $round \leftarrow \lceil size/HASH_BY_ROUND \rceil - previous_round;$
8 **if** *round == 0* **then**
9 | $WriteMemory(mem_{SMRAM}[d_i], 0);$
10 | $WriteMemory(mem_{SMRAM}[d_{i+1}], 0);$
11 | $WriteMemory(mem_{SMRAM}[d_{i+2}], 0);$
12 | $WriteMemory(mem_{SMRAM}[d_{i+3}], 0);$
13 | $WriteMemory(mem_{SMRAM}[d_{i+6}], 0);$
14 **else**
15 | $ComputeHash(\ );$
16 | $WriteMemory(mem_{SMRAM}[d_{i+1}], Increment(index));$
17 | $WriteMemory(mem_{SMRAM}[d_{i+2}], Increment(previous_round));$
18 | $WriteMemory(mem_{SMRAM}[d_{i+7}], 1);$
19 **End-ManageComputeHash**

the boot up process in the SMRAMC register [69, 70, 47] as presented in figure 3.5. As this is a matter of implementation, it is discussed in chapter 6. Although this alternative can be viable for testing the agent, we just recommend it for test purposes, not for production environment since $SBST$ would fail to meet requirements $r5$ (isolated) and $r10$ (complete). This step is implemented considering the whole $SBST$.

5.4.4 Step 4: Uploading $SBST$ into $mem_{SMRAM}[i]$

Step 4 aims to meet requirement $r5$ (isolated), according to section 5.2.5. Isolate $SBST$ means to upload the complete $SBST$ as monolithic software from $mem_{BIOS}[i]$ to $mem_{SMRAM}[i]$ during the bootup process. Alternatively, it can be uploaded by the manager module, as described in step 3 (Section 5.4.3), which has security and requirements implications. To meet requirements $r3$, $r4$ and $r8$ contribute to meet requirement $r5$. This step is implemented considering the whole $SBST$.

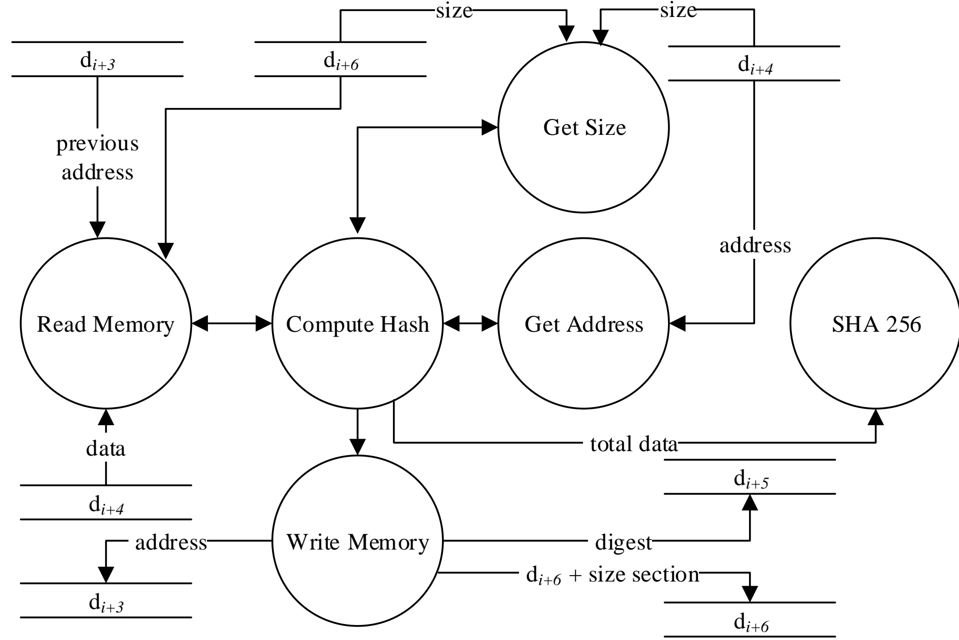

Figure 5.7: **Algorithm Compute Hash.** The compute hash function computes the data hash read from $mem_{SMRAM}[d_{i+4}]$, which was loaded in the previous round.

5.4.5 Step 5: $SBST$ Resistant

Step 5 aims to meet requirement $r6$. Algorithm 5.8 describes a basic approach to meet this requirement, by enforcing the use of SMRR interface and by setting up the SMRAM address range protected. It is done by setting the appropriated registers as described in section 3.2.3. This step is implemented in the bc, since it is a management function which should be performed whenever the $SBST$ starts.

Thus, algorithm 5.8 tests bit 11 in the IA32_MTRRCAP register to check the support to the SMRR interface. Intel recommends that such a bit is tested, before try to access SMRR registers, because in the case SMRR interface is not supported, a attempt to read or writes those registers will cause general-protection exceptions [87, 97]. If this bit is set, it means that the processor support SMRR interface.

Function *ReadRegister( )* receives two parameters: a register to be read and position indicating the bit or bits to be read.

So, it configures registers IA32_SMRR_PHYSBASE and IA32_SMRR_PHYSMASK to enforce the use of SMRR interface, thwart attacks as "cache poisoning" and to determine SMRAM address range protected. Such a range can be increased or decreased, according to the security design.

Algorithm 5.5: Compute data hash. It computes the hash code from the data load in by the previous function.

Input : Data to be computed the hash digest.
Output: Hash digest.

```
1  Function ComputeHash( )
2  Begin
3  previous_address ← ReadMemory(mem_SMRAM[d_{i+3}]);
4  if previous_address <> NULL then
5      address ← previous_address + HASH_BY_ROUND;
6      size ← GetSize(mem_SMRAM[d_{i+4}]) − GetSize(mem_SMRAM[d_{i+6}]);
7  else
8      address ← GetAddress(mem_SMRAM[d_{i+4}]);
9      size ← GetSize(mem_SMRAM[d_{i+4}]);
10 size_section ← 0;
11 while ((size_section <= HASH_BY_ROUND)&(size_section <= size)) do
12     data ← ReadMemory(mem_SMRAM[address]));
13     size_section ← size_section + GetSize(data);
14     total_data ← total_data + data;
15 digest ← SHA − 256(total_data);
16 WriteMemory(mem_SMRAM[d_{i+3}], address);
17 WriteMemory(mem_SMRAM[d_{i+5}[index]], digest);
18 WriteMemory(mem_SMRAM[d_{i+6}], (ReadMemory(mem_SMRAM[d_{i+6}]) +
     size_section));
19 End-ComputeHash
```

5.5 Discussion

In this chapter we specified and discussed the requirements, detailing how they can be built. We also proposed a general architecture, explaining where each requirement fit in the global view. Then, we designed the architecture to answer the proposed research questions adding the requirements step by step to the $SBST$ general architecture. Finally, we designed the algorithms to realize the requirements by means of a study case on a payload to measure the integrity of a simple table (file **xend-config.sxp**) of a hypervisor (**xen 4.0**).

The algorithms designed ignore most of the implementations issues and complexities, since they are addressed in the next chapters. However, those algorithms describe the steps necessary to meet the requirements, pointing out specifics components, as register, which must be address to implement an $SBST$.

Technological details as memory and CPU speed to read and write were not imperative during the design of the architecture presented in this chapter. However, we used results of obtained in our experiments with SMI handler, as decribed in section 3.2.4 and use approximated values in the other cases.

In section 4.5 we discussed on some desirable characteristics and capacities as the characteristic of being "unique" in the SMRAM and the capacity of "self-cleaning". So, here we expand a bit more that discussion. The characteristic of

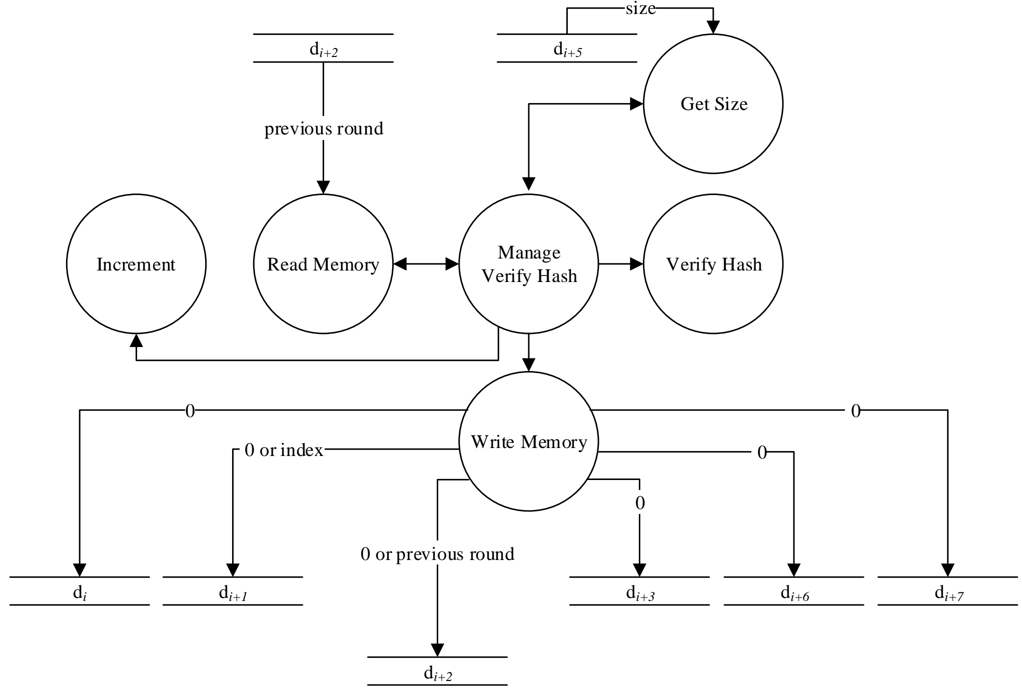

Figure 5.8: **Algorithm Manage Verify Hash.** This function prepares the environment to verify the data hash. The digest is verified by function VerifyHash( ). It reads from and writes to variables (d_{i+j}) located in $mem_{SMRAM}[i]$ to save the round context.

being "unique" can be achieved by setting appropriated registers in the set of SMM related registers to shrinking the SMRAM size after uploading the $SBST$ and allowing no more codes to be deployed into SMRAM. It can be specified as $mem_{SMRAM} = SBST \equiv mem_{SMRAM} = (SH)$, where mem_{SMRAM} is represented as an unitary set, since it must not be any other code sharing mem_{SMRAM} with $SBST$. Note that the state save map (sections 3.3.1 and 3.2.2) still in mem_{SMRAM}, stored in its own area (figures 3.2 and 3.3). Also, note that after meeting requirement $r4$ (cooperative), $SBST \equiv SH$. So, the available mechanisms to protect mem_{SMRAM} and SH, as that describe in requirement $r6$, can be used and enforced to $SBST$ too.

The capacity of being "self-cleaning" can be achieved by implement a function to issue GETSEC[SENTER] and GETSEC[SEXIT] instructions whenever necessary. It can be specified as $bc_{LateLaunch} : mem_{SMRAM_x} \rightarrow mem_{SMRAM_y}$, meaning that bc would have the capacity to start a late launch or another equivalent operation to get mem_{SMRAM} cleaned. The defined function $bc_{LateLaunch}$ take mem_{SMRAM} from a status x to a status y, where x is a potential tampered status and y is a pristine one. The algorithm 5.9 presents a potential design for this capacity.

Thus, algorithm 5.9 checks the status of the IA32_MCn_STATUS register and triggers a late launch operation to clean the mem_{SMRAM}. The algorithm receives

Algorithm 5.6: Manage to call function VerifyHash( ). It prepares the environment to call function VerifyHash( ). If no more rounds are needed, it saves the state and passes control to the next function. If rounds are required yet, it calls the VerifyHash( ) again.

Input : An integer $index$ indicating the next position in the $mem_{SMRAM}[i]$ to read the hash digest.
Output: Call function VerifyHash().

```
1  Function  ManageVerifyHash( )
2  Begin
3  previous_round ← ReadMemory(mem_SMRAM[d_{i+2}]);
4  if previous_round == NULL then
5  |   previous_round ← 0;
6  size ← GetSize(mem_SMRAM[d_{i+5})];
7  round ← ⌈size/HASH_BY_ROUND⌉ − previous_round;
8  if round == 0 then
9  |   WriteMemory(mem_SMRAM[d_i], 0);
10 |   WriteMemory(mem_SMRAM[d_{i+1}], 0);
11 |   WriteMemory(mem_SMRAM[d_{i+2}], 0);
12 |   WriteMemory(mem_SMRAM[d_{i+3}], 0);
13 |   WriteMemory(mem_SMRAM[d_{i+6}], 0);
14 else
15 |   VerifyHash();
16 |   WriteMemory(mem_SMRAM[d_{i+1}], Increment(index));
17 |   WriteMemory(mem_SMRAM[d_{i+2}], Increment(previous_round));
18 |   WriteMemory(mem_SMRAM[d_{i+7}], 0);
19 End-ManageComputeHash
```

two integers: $number_of_cores$, indicating the number of processor cores presents in the target machine, and an integer $number_of_registers$, indicating the number of IA32_MCn_STATUS registers present in the processor, since those registers are unique by core (section 3.2.3). Then, it checks the status of IA32_MCn_STATUS registers by probing bit 31, which indicates when an error has happened, and the bit 2, which indicates when the error was a violation of the ranges configured in the SMRR interface. In the case of a SMRR interface violation, a GETSEC[SEXIT]-like instruction is triggered to flush mem_{SMRAM} load a pristine $SBST$ into mem_{SMRAM}.

Those characteristics and capacities are discuss but not implemented in our proof of concept.

5.6 Summary

The requirements identified in this research are classified as functional requirements ($r1$, $r2$, $r3$, $r4$ and $r7$) and security requirements ($r5$, $r6$ and $r8$). Functional requirements are related to functionality of the $SBST$ (as $r3$ - persistent) or related to the way it needs to work (as fast). Security requirements are needed to improve the $SBTS$ security (as $r5$ - isolate and $r6$ - resistance). Some requirements can be

Algorithm 5.7: Verify data hash. It computes the hash code from the data load in by the previous function, retrieve the hash code stored in $mem_{SMRAM}[i]$ and compare those two hash code. Then, the algorithm exhibit a message with the result of verification.

Input : Hash digest.
Output: The result of integrity check.

1 $Function \quad VerifyHash(\quad)$
2 **Begin**
3 $previous_address \leftarrow ReadMemory(mem_{SMRAM}[d_{i+3}]);$
4 **if** $previous_address <> NULL$ **then**
5 | $address \leftarrow previous_address + HASH_BY_ROUND;$
6 | $size \leftarrow GetSize(mem_{SMRAM}[d_{i+4}]) - GetSize(mem_{SMRAM}[d_{i+6}]);$
7 **else**
8 | $address \leftarrow GetAddress(mem_{SMRAM}[d_{i+4}]);$
9 | $size \leftarrow GetSize(mem_{SMRAM}[d_{i+4}]);$
10 $size_section \leftarrow 0;$
11 **while** $((size_section <= HASH_BY_ROUND)\&(size_section <= size))$ **do**
12 | $data \leftarrow ReadMemory(mem_{SMRAM}[address]));$
13 | $size_section \leftarrow size_section + GetSize(data);$
14 | $total_data \leftarrow total_data + data;$
15 $digest \leftarrow SHA - 256(total_data);$
16 $stored_digest \leftarrow ReadMemory(mem_{SMRAM}[d_{i+5}[index]]);$
17 **if** $digest == stored_digest$ **then**
18 | Print("The integrity of *xend-config.sxp* table IS PRESERVED in section %d", $index$);
19 **else**
20 | Print("The integrity of *xend-config.sxp* table IS COMPROMISED in section %d", $index$);
21 $WriteMemory(mem_{SMRAM}[d_{i+3}], address);$
22 $WriteMemory(mem_{SMRAM}[d_{i+6}], (ReadMemory(mem_{SMRAM}[d_{i+6}]) + size_section));$
23 **End-VerifyHash**

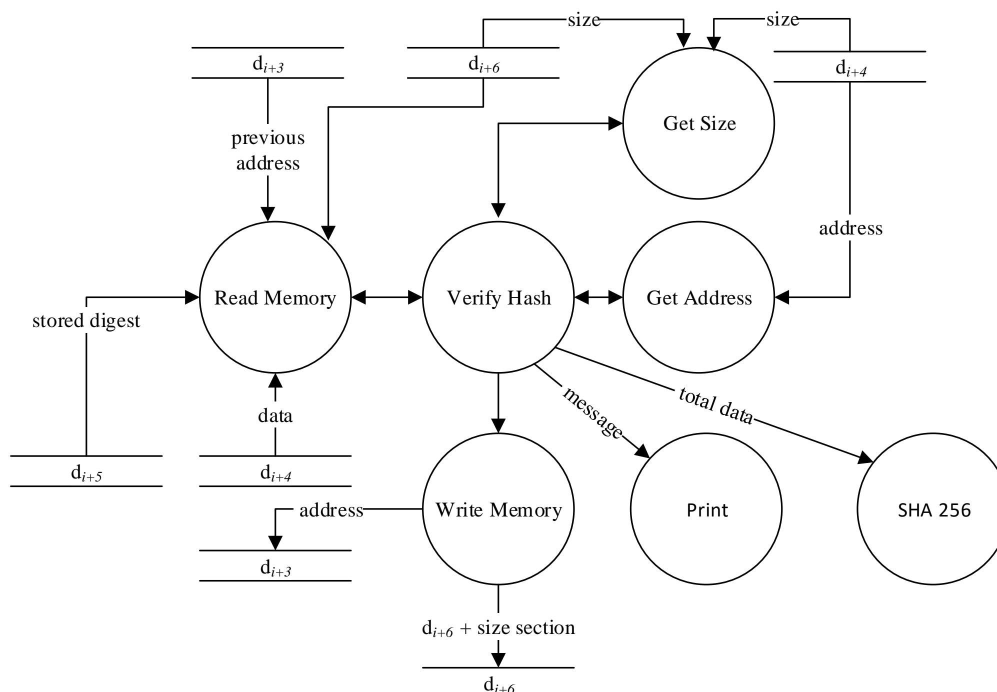

Figure 5.9: **Algorithm Verify Hash.** Verify hash function computes the data hash read from $mem_{SMRAM}[d_{i+4}]$, which was load in the previous round and compare with the hashes computed by function ComputeHash( ) stored in $mem_{SMRAM}[d_{i+5}]$, printing a message to inform the result.

Algorithm 5.8: Verify and set registers to reinforce SMM security. It verifies if the target machine supports SMRR interface. If yes, the algorithm sets registers IA32_SMRR_PHYSBASE and IA32_SMRR_PHYSMASK to enforce the use of SMRR interface and avoid attacks as "cache poisoning".

Output: Registers configured.

1 $Function \quad Main$;
2 **Begin**;
3 integer $smrr_support$;
4 $smrr_support \leftarrow ReadRegister(IA32_MTRRCAP, 11)$;
5 **if** $smrr_support == 1$ **then**
6 | Print("This processor support SMRR interface");
7 | $Configure(IA32_SMRR_PHYSBASE)$;
8 | $Configure(IA32_SMRR_PHYSMASK)$;
9 **else**
10 | Print("This processor does NOT support SMRR interface");
11 **End-Main**;

Algorithm 5.9: Verify SMMR interface integrity and trigger a late launch instruction. It probes IA32_MCn_STATUS registers to check if a violation of the addresses ranges configured in the SMRR interface happened. If yes, it triggers a GETSEC[SEXIT]-like instruction to flush mem_{SMRAM} and reload $SBST$ from mem_{BIOS}.

Input : An integer $number_of_cores$ indicating the number of processor cores and an integer $number_of_registers$ the number of IA32_MCn_STATUS registers present in the processor.
Output: Restart the system and reload $SBST$.

```
1  Function Main(integer number_of_cores, integer number_of_registers);
2  Begin;
3  integer counter_1, counter_2, smram_integrity;
4  counter_1 ← 0;
5  while counter_1 < number_of_cores do
6  |   counter_2 ← 0;
7  |   while counter_2 < number_of_registers do
8  |   |   smram_integrity ← ReadRegister(IA32_MC_counter_2_STATUS, 31);
9  |   |   if smram_integrity == 1 then
10 |   |   |   Print("An error reported in the register IA32_MC_STATUS = %d",
   |   |   |     IA32_MC_n+counter_2_STATUS);
11 |   |   |   smram_integrity ←
   |   |   |     ReadRegister(IA32_MC_n+counter_2_STATUS, 2);
12 |   |   |   if smram_integrity == 1 then
13 |   |   |   |   Print("SBST integrity violated! Starting a late launch
   |   |   |   |     operation...");
14 |   |   |   |   GETSEC[SEXIT];
15 |   |   |   else
16 |   |   |   |   Print("Error not related to SMM!");
17 |   |   else
18 |   |   |   Print("No error reported in the register IA32_MC_STATUS = %d",
   |   |   |     IA32_MC_n+counter_2_STATUS);
19 |   |   Increment(counter_2);
20 |   Increment(counter_1);
21 End-Main;
```

met by implementation (as $r10$ - complete) and other by following some operational procedure (as $r5$ - isolated).

According to the figure 5.1: requirements $r1$, $r2$ and $r4$ are met by coding them in the $SBST$ as specified in the last section; requirement $r3$ is met by inserting $SBST$ in the mem_{BIOS}; requirement $r5$ is met by uploading $SBST$ from mem_{BIOS} to mem_{SMRAM}; requirement $r6$ can also be codified in the $SBST$ since it is about to handle registers values; requirement $r7$ is met by coding it in the $SBST$ and by using another software artefact called "Manager", which contributes to discover the event that can trigger an SMI in the target platform. The manager resides in the DRAM, but it is not necessarily in the target machine. So, there is not isolation problem in this concept; and requirement $r8$ is met by designing $SBST$ to be a monolithic code, deployed into mem_{SMRAM}.

We developed a payload study case on measuring the integrity of hypervisor essential data. That data consisted of a simple table. So, our design have focused on making $r2$ (fast) dependent on the time elapsed when executing a subtask, in such a way that $SBST$ pre-empts before a subtask reach the time limit. We estimate the time needed to perform each subtask assuming experiments on time elapsed for our functions in the study case. However, in a real case, it is necessary to know the technological details about DRAM and CPU to calculate the execution time and pre-empt $SBST$.

This chapter has specified and discussed the requirements established in chapter 4, detailing how they can fit in the architecture. Then, we proposed a general architecture and design the algorithms to built that architecture. In the next chapter, we will tackle implementation issues, detailing the function in the manager module.

Chapter 6

Implementation and Evaluation - Manager Module and SBST

Ignota nulla curatio morbi (do not attempt to cure what you do not understand).

OLD PRINCIPLE OF MEDICINE , AS DESCRIBED BY ANDRZEJ M. ŁOBACZEWSKI IN HIS BOOK POLITICAL PONEROLOGY.

The proof of concept developed in this work was planned to be implemented in two parts: a manager module and an agent module. The agent (properly the $SBST$, as defined in 2.23) does not need the manager to operate. The planned actions for the manager are probe and research a target platform to understand and learn about the environment where an agent ($SBST$) will be deployed.

Some requirements are implemented in the manager module and others in the agent (figure 5.1). That issue will be made clear in this chapter.

6.1 Introduction

In the previous chapter, the requirements established were specified and discussed. Those requirements were mapped in a general architecture, where they were classified as functional or security requirements. Then, we designed the architecture and its respective algorithms to answer the proposed research questions adding the requirements step by step to general architecture. The algorithms were designed considering a payload study case to measure the integrity of a simple table (file xend-config.sxp) of a hypervisor (the Xen hypervisor 4.0).

Since it is impossible remove the SMM from x86 architecture [57] and since the reported SMM violations are platform specific or operating system specific. It is better to understand and learn about SMM.

In this chapter we consider the findings in chapter 2 and 3 to build the manager module targeting the machine 12 at table 3.1, considering that the actions and functions planned to the manager is probe and research a platform to understand and learn about the environment where the agent will be deployed. The manager module is useful to understand and learn about a target chipset, mainly about the status of some SMM related registers.

This chapter is organised in four content sections and a discussion and summary section. In the first section, we discuss the constraints and limits of the manager module. The second section discusses some of the implementations issues. Section three details the functions implemented in the manager. Then, section four presents some results from the execution of the manager module targeting a subset of the machines presented in table 3.1.

This chapter is organised in three content sections and a discussion and a summary section. In the first section, we discuss the limits and constraints of the tool. Second Section discusses some of the proof of concept itself. Section three details the functions implemented in the tool and other related issues and report the experiments with the tool.

6.1.1 Manager Module Implementation particularities

In this section, we briefly discuss some implementation issues from our computational experiments.

We implement the manage module in eight files (Figure 6.1), writing it in C language. That implementation was inspired in the Inteltools project [37] and [23]. The $manager.c$ file implements the main functions and it calls all the other files. The $manager.h$ file implements the header file used for all other files, including $manager.c$. The file $smramc.c$ implements functions to report information and manage controls related to the SMRAM and SMRAMC. The file $msr.c$ implements functions to report information about the Architectural Model-Specific Registers. The file $chipset.c$ implements functions to report information about the chipset specific registers, as detailed in section 3.2.3. File $info.c$ implements functions to reports information about the host platform. The file $smi.c$ implements functions to enable, disable and generate SMI. File $inject.c$ implements functions to inject the $SBST$ in to the $mem_{SMRAM}[i]$, alternatively to embed it into $mem_{BIOS}[i]$, but not meeting requirements $r5$ and $r10$, as discussed in section 5.4.3.

Functions implemented in **smramc.c** file:

```
void set_SMRAM_OFFSET(unsigned int offset);
void print_smramc(uint8_t smramc_status);
void lock_smramc(struct pci_dev *smramc, uint8_t smramc_status);
void unlock_smramc(struct pci_dev *smramc, uint8_t smramc_status);
void open_smram(struct pci_dev *smramc, uint8_t smramc_status);
void close_smram(struct pci_dev *smramc, uint8_t smramc_status);
uint16_t get_pmbase(void);
uint16_t get_smi_en_iop(void);
uint16_t get_smi_sts_iop(void);
uint16_t get_pm1_cnt_iop(void);
uint16_t get_gen_pmcon_1_iop(void);
uint32_t get_smi_en_content(uint16_t smi_en_iop);
uint32_t get_smi_sts_content(uint16_t smi_sts_iop);
void write_to_apm_cnt(void);
void print_gen_pmcon_1(uint16_t gen_pmcon_1_status);
void print_apm_cnt_sts(uint8_t apm_cnt_status, uint8_t apm_sts_status);
void print_gpi_rout(uint32_t gpi_rout_status);
void print_gpi_rout2(uint32_t gpi_rout2_status);
```

Functions implemented in **msr.c** file:

```
unsigned int cpuid(unsigned int op);
msr_t rdmsr(int addr);
int print_intel_core_msrs(void);
int verify_smrr_support(void);
```

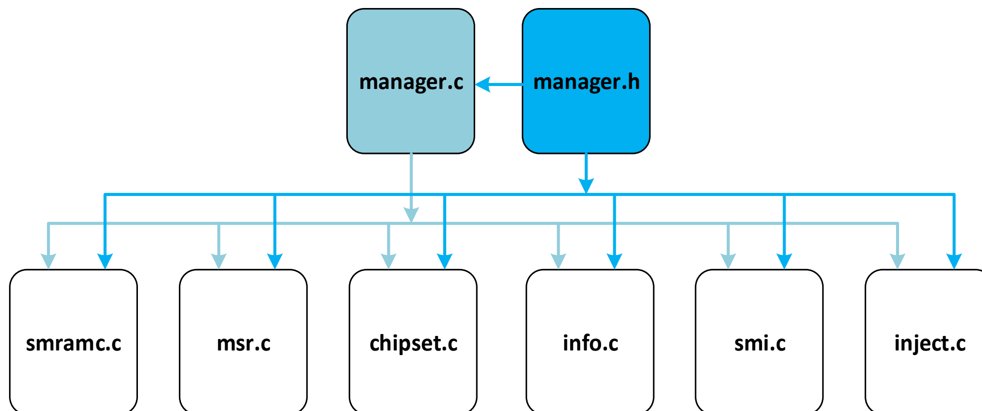

Figure 6.1: **Manager program files.** The manager program comprises of eight files written in C language with some code chunks in Assembly language. The *manager.c* file is the main file and *manager.h* is header file used by all the other programs in the manager module.

```
int verify_stm_support(void);
int verify_ia32_smbase_support(void);
int verify_ia32_smm_monitor_ctl_lock(void);
int print_mseg_revision_identifier(void);
int verify_stm_invocation(void);
int print_mseg_base(void);
int verify_freeze_support(void);
int verify_freeze_performance(void);
int check_smi_handler_violation(void);
```

Functions implemented in **chipset.c** file:

```
int print_pmbase(struct pci_dev *sb, struct pci_access *pacc);
void print_smi_en(uint32_t smi_en_status);
void print_smi_sts(uint32_t smi_sts_status);
void print_alt_gpi_smi_en(uint16_t alt_gpi_smi_en_status);
void print_alt_gpi_smi_en2(u16 alt_gpi_smi_en2_status);
void print_alt_gpi_smi_sts(u16 alt_gpi_smi_sts_status);
void print_alt_gpi_smi_sts2(u16 alt_gpi_smi_sts2_status);
void print_gpe0_en(uint32_t gpe0_en_status_high, uint32_t gpe0_en_status_low);
void print_gpe0_sts(uint32_t gpe0_sts_status_high, uint32_t gpe0_sts_status_low);
```

Functions implemented in **info.c** file:

```
void *map_physical(uint64_t phys_addr, size_t len);
void unmap_physical(void *virt_addr, size_t len);
int print_supported_chipsets( );
int print_platform( );
```

Functions implemented in **smi.c** file:

```
int enable_smi_gbl(u16 smi_en_iop);
int disable_smi_gbl(u16 smi_en_iop);
int enable_smi_on_apm(u16 smi_en_iop);
int disable_smi_on_apm(u16 smi_en_iop);
int enable_smi_xHCI(u16 smi_en_iop);
int enable_smi_Intel_ME(u16 smi_en_iop);
int enable_smi_GPIO_Unlock(u16 smi_en_iop);
int enable_smi_Intel_USB2_EN(u16 smi_en_iop);
```

```
int enable_smi_legacy_USB2_EN(u16 smi_en_iop);
int enable_smi_periodic(u16 smi_en_iop);
int enable_smi_TCO_logic(u16 smi_en_iop);
int enable_smi_MCSMI(u16 smi_en_iop);
int enable_smi_SWSMI_TMR_EN(u16 smi_en_iop);
int enable_smi_APM_CNT(u16 smi_en_iop);
int enable_smi_SLP_SMI_EN(u16 smi_en_iop);
int enable_smi_legacy_USB_EN(u16 smi_en_iop);
int enable_smi_BIOS_EN(u16 smi_en_iop);
int disable_smi_xHCI(u16 smi_en_iop);
int disable_smi_Intel_ME(u16 smi_en_iop);
int disable_smi_GPIO_Unlock(u16 smi_en_iop);
int disable_smi_Intel_USB2_EN(u16 smi_en_iop);
int disable_smi_legacy_USB2_EN(u16 smi_en_iop);
int disable_smi_periodic(u16 smi_en_iop);
int disable_smi_TCO_logic(u16 smi_en_iop);
int disable_smi_MCSMI(u16 smi_en_iop);
int disable_smi_SWSMI_TMR_EN(u16 smi_en_iop);
int disable_smi_APM_CNT(u16 smi_en_iop);
int disable_smi_SLP_SMI_EN(u16 smi_en_iop);
int disable_smi_legacy_USB_EN(u16 smi_en_iop);
int disable_smi_BIOS_EN(u16 smi_en_iop);
int generate_smi_periodic(u16 gen_pmcon_1_status_x, u16 address_gen_pmcon_1, int rate);
int generate_smi_SLP_SMI_EN(u16 smi_en_iop, u16 smi_sts_iop, u16 pm1_cnt_iop);
int generate_smi_BIOS_EN(u16 smi_en_iop, u16 smi_sts_iop, u16 pm1_cnt_iop);
```

The manager module is developed mainly in C language, with some code chunks in Assembly language. We use two machines for that : an Acer Intel pentium i5 8GB [80, 81] and a Compac Intel pentium III 1GB RAM [64, 65] (see table 3.1). The operating system upon it is developed is CentOS 5.10. To access registers, we use the "pciutils-3.2.1" library.

6.2 Functions in the Manager Module

As discussed in section 4.4, SMM is: chipset, platform, OEM and executive software specific. These specificities imply that to develop an $SBST$ is mandatory to know deeply the target machine. Thereby, in this section we present the functions that the manager module implements to: probe, identify components and obtain information from a target machine. It basically covers items related to SMM in platform x86, as described in figure 6.2:

6.2.1 Probe and Report the Target Machine Menu

This option offers a subset of functions to probe the platform to discover information related to SMM, as registers content and memory details. It also identifies information related to the target machine (the one hosting $SBST$). So, it can report: all SMM information, the host chipset, SMRAMC register, PMBASE and SMM related registers, CPU's SMM related registers, GEN_PMCON_1 register, APM registers, GPI_ROUT register, GPI_ROUT2 register, MSEG revision identifier, MSEG base and supported chipsets.

6.2.1.1 Report All SMM Information Function

This function consolidates and reports all the information provided by functions in the probe and report the target machine menu, except function "Report Supported

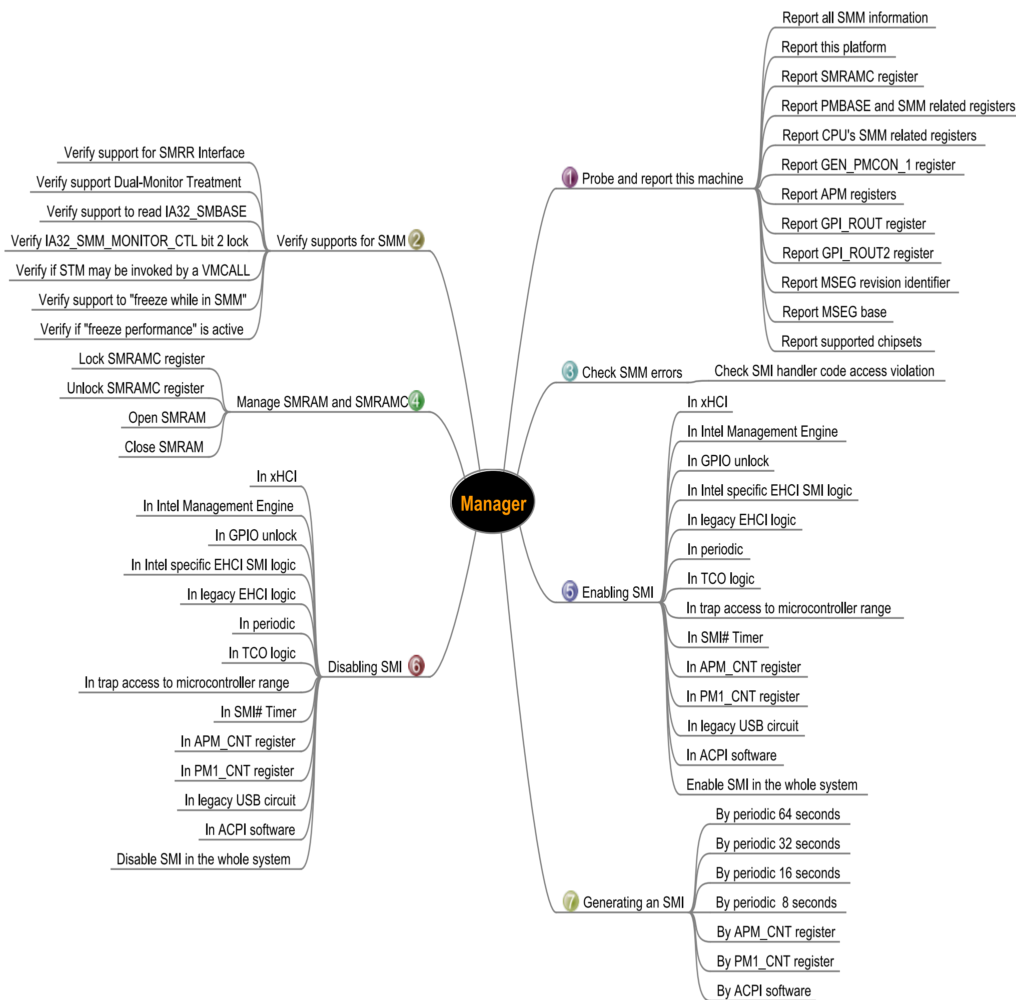

Figure 6.2: **Manager main menu.** The manager module has seven options offering functions to obtain information from the host machine, as: probe and report information, verify supports and check errors related to SMM, manage controls related to the SMRAM and SMRAMC and enable, disable and generating SMI.

Chipsets". It is implemented by function $print_all_SMM_information()$.

6.2.1.2 Report the Host Chipset Function

This option is implemented by function $print_platform()$ and reports data from CPU (id, Processor type, family, model and stepping), Northbridge (vendor and device id and name), Southbridge (vendor and device id and name) and the IGF (vendor and device id and name).

6.2.1.3 Report SMRAMC Register Function

This option is implemented by function *print_smramc(uint8_t smramc_status)* and reports the status of $mem_{SMRAM}[i]$ and the status of SMRAMC register.

6.2.1.4 Report PMBASE and SMM Related Registers Function

This option reports the status of the registers PMBASE, GPE0_EN, GPE0_STS, SMI_EN, SMI_STS, ALT_GPI_SMI_EN, ALT_GPI_SMI_STS, ALT_GPI_SMI_EN2, and ALT_GPI_SMI_STS2; and the information in the chipset related to them. It is implemented by function *int print_pmbase(struct pci_dev *sb, struct pci_access *pacc)*.

6.2.1.5 Report CPU's SMM Related Registers Function

This option reports the status of Model-Specific Registers shared with all cores in the processor, as the IA32_MCi_STATUS registers. It also report the status of registers IA32_SMM_MONITOR_CTL, IA32_MTRRCAP, IA32_VMX_BASIC, IA32_VMX_MISC, IA32_DEBUGCTL, IA32_PERF_CAPABILITIES, by core. It is implemented by function $print_intel_core_msrs()$.

6.2.1.6 Report GEN_PMCON_1 Register Function

This option reports the status of GEN_PMCON_1 Register and the information about the chipset this register provide. It is implemented by function $print_gen_pmcon_1(gen_pmcon_1_status)$.

6.2.1.7 Report APM Registers Function

This option reports the status of registers APM_CNT and APM_STS and it is implemented by function $print_apm_cnt_sts(apm_cnt_status, apm_sts_status)$.

6.2.1.8 Report GPI_ROUT Register Function

This option is implemented by function $print_gpi_rout(gpi_rout_status)$ and reports register GPI_ROUT status and reports information about the General Purpose I/O devices, from GPI0 to GPI15.

6.2.1.9 Report GPI_ROUT2 Register Function

This option is implemented by function $print_gpi_rout2(gpi_rout2_status)$ and reports register GPI_ROUT2 status and reports information about the General Purpose I/O devices: GPI17, GPI19, GPI21, GPI22, GPI43, GPI56, GPI57 and GPI60.

6.2.1.10 Report MSEG Revision Identifier Function

This option is implemented by function $int\ print_mseg_revision_identifier(void)$ and probes bit63 to bit32 of IA32_VMX_MISC register to report the 32-bit MSEG revision identifier used by the processor [87], by core.

6.2.1.11 Report MSEG Base Function

This option is implemented by function $int\ print_mseg_base(void)$ to report the MSEG base contained into bit31 to bit12 of the IA32_SMM_MONITOR_CTL register.

6.2.1.12 Report Supported Chipsets Function

This option is implemented by function $int\ print_supported_chipsets()$ and report all the chipsets supported by the manager module.

6.2.2 Verify Supports for SMM Menu

This option offers a subset of functions to verify different supports to SMM tasks, as support for SMRR interface and for dual-monitor treatment.

6.2.2.1 Verify support for SMRR Interface Function

This option is implemented by function $int\ verify_smrr_support(void)$ to verify bit11 of the IA32_MTRRCAP register and report per core if the processor in the host machine supports the SMRR interface.

6.2.2.2 Verify support Dual-Monitor Treatment Function

This option is implemented by function $int\ verify_stm_support(void)$ to verify bit17 of the IA32_VMX_BASIC register and report per core if the processor in the host machine supports the Dual-Monitor Treatment.

6.2.2.3 Verify support to read IA32_SMBASE Function

This option is implemented by function $int\ verify_ia32_smbase_support(void)$ to verify and report per core if the RDMSR instruction is allowed to read the IA32_SMBASE register when in SMM. VMXOFF instruction unblocks SMIs unless bit2 in IA32_SMM _MONITOR_CTL register is set (equals to 1).

6.2.2.4 Verify IA32_SMM_MONITOR_CTL bit 2 lock Function

This option is implemented by function $int\ verify_ia32_smm_monitor_ctl_lock(void)$ to verify and report per core if bit2 in IA32_SMM_MONITOR_CTL register can be set to 1. VMXOFF instruction unblocks SMIs unless bit2 in IA32_SMM_MONITOR_CTL register is set (equals to 1).

6.2.2.5 Verify if STM may be invoked by a VMCALL Function

This option is implemented by function $int\ verify_stm_invocation(void)$ to verify and report per core if STM can be invoked using a VMCALL instruction (section 3.3.4). It verifies bit0 in SMM_MONITOR_CTL register. The support is activated if bit0 is set.

6.2.2.6 Verify support to "freeze while in SMM" Function

This option is implemented by function $int\ verify_freeze_support(void)$ and verifies and reports per core if the processor supports the FREEZE_WHILE_SMM_EN feature. This supports is active when bit12 in IA32_PERF_CAPABILITIES register is set (equals to 1).

6.2.2.7 Verify if "freeze performance" is active Function

This option is implemented by function $int\ verify_freeze_performance(void)$ to verify and report per core if the "freeze performance" is activated in the processor. It is done by verifying the bit14 in the IA32_DEBUGCTL register.

6.2.3 Check SMM Errors Menu

It checks SMI handler code access violation defined in the SMRR interface (algorithm 5.9). In this manager module version it offers only the option "Check SMI handler code access violation", which is implemented by the function $int\ check_smi\ _handler_violation(void)$.

6.2.4 Manage SMRAM and SMRAMC Menu

It makes available functions to attempt to lock or unlock the SMRAMC register and to open or close the SMRAM.

6.2.4.1 Lock SMRAMC register Function

This option is implemented by function $void\ lock_smramc(struct\ pci_dev * smramc,$ $uint8_t\ smramc_status)$ to try locking SMRAMC register by setting bit D_LCK in SMRAMC register (section A.1 and figure 3.4).

6.2.4.2 Unlock SMRAMC register Function

This option is implemented by function $void\ unlock_smramc(struct\ pci_dev{*}smramc,$ $uint8_tsmramc_status)$ to try unlocking SMRAMC register by clearing bit D_LCK in SMRAMC register (section A.1 and figure 3.4).

6.2.4.3 Open SMRAM Function

This option is implemented by function $void\ open_smram(structpci_dev * smramc,$ $uint8_tsmramc_status)$ to try opening mem_{SMRAM} by setting bit D_OPEN in the SMRAMC register (section A.1 and figure 3.4).

6.2.4.4 Close SMRAM Function

This option is implemented by function $void\ close_smram(structpci_dev * smramc,$ $uint8_tsmramc_status)$ to try closing mem_{SMRAM} by clearing bit D_OPEN in the SMRAMC register (section A.1 and figure 3.4).

6.2.5 Enabling SMI Menu

As stated in section 3.3.1, There are many reasons to generate an SMI, depending on the chipset in use. Thereby, in this section we explain a set of functions offered to enable SMI to be generated in the chipset for a specific reason, by setting specifics bits in registers, as APM_CNT register and PM1_CNT register. There is also a function to enable SMI globally in the chipset. This last function is quite important to generate SMI in general. For example, to generate an SMI in the APM_CNT, for

example, SMI must be globally enable. In general, writing to those kind of registers requires high privileges. So, when using Linux OSs one can use the command line $iopl(3)$, this command changes the I/O privilege level of the calling process.

In the subsections described ahead, we use the verb "try", for example, "try to enable", "try to disable", "try to control", "try to generate", for the functions described by those subsections because even if the implemented function have the $iopl(3)$ privilege, it might not be always able to change the values in the registers. The reason for that needs to be investigated and will not be addressed in this research.

6.2.5.1 Enable SMI in xHCI function

This function try to enable an SMI to be generated by a xHCI device, by setting bit31 in the SMI_EN register. It is implemented by function $int\ enable_smi_xHCI(u16\ smi_en_iop)$.

6.2.5.2 Enable SMI in Intel Management Engine function

This function try to enable an SMI to be generated by Intel Management Engine, by setting bit30 in the SMI_EN register. It is implemented by function $int\ enable_smi_Intel_ME(u16\ smi_en_iop)$.

6.2.5.3 Enable SMI in GPIO unlock function

This function try to enable the GPIO registers lockdown logic to launch an SMI, by setting bit27 in the SMI_EN register. It is implemented by function $int\ enable_smi_GPIO_Unlock(u16\ smi_en_iop)$.

6.2.5.4 Enable SMI in Intel specific EHCI SMI logic function

This function try to enable Intel-Specific EHCI SMI logic to cause an SMI, by setting bit18 in the SMI_EN register. It is implemented by function $int\ enable_smi_Intel_USB2_EN(u16\ smi_en_iop)$.

6.2.5.5 Enable SMI in legacy EHCI logic function

This function try to enable legacy EHCI logic to cause an SMI, by setting bit17 in the SMI_EN register. It is implemented by function $int\ enable_smi_legacy_USB2_EN(u16\ smi_en_iop)$.

6.2.5.6 Enable SMI in periodic function

This function try to enable the chipset to generate an SMI periodically, by setting bit14 in the SMI_EN register. It is implemented by function $int\ enable_smi_periodic(u16\ smi_en_iop)$.

6.2.5.7 Enable SMI in TCO logic function

This function try to enable the TCO logic to generate an SMI, by setting bit13 in the SMI_EN register. It is implemented by function $int\ enable_smi_TCO_logic(u16\ smi_en_iop)$.

6.2.5.8 Enable SMI in trap access to microcontroller range function

This function try to enable the chipset to trap accesses to the microcontroller range and generate an SMI, by setting bit11 in the SMI_EN register. It is implemented by function $int\ enable_smi_MCSMI(u16\ smi_en_iop)$.

6.2.5.9 Enable SMI in SMI Timer function

This function try to start Software SMI Timer, by setting bit6 in the SMI_EN register. When the timer expires, an SMI is generated. It is implemented by function $int\ enable_smi_SWSMI_TMR_EN(u16\ smi_en_iop)$.

6.2.5.10 Enable SMI in APM_CNT register function

This function try to enable writing to the APM_CNT register to generate an SMI, by setting bit5 in the SMI_EN register. It is implemented by function $int\ enable_smi_APM_CNT(u16\ smi_en_iop)$.

6.2.5.11 Enable SMI in PM1_CNT register function

This function try to enable the PM1_CNT register to generate an SMI, by setting bit4 in the SMI_EN register. It is implemented by function $int\ enable_smi_SLP_SMI_EN(u16\ smi_en_iop)$.

6.2.5.12 Enable SMI in legacy USB circuit function

This function try to enable legacy USB circuit to cause an SMI, by setting bit3 in the SMI_EN register. It is implemented by function $int\ enable_smi_legacy_USB_EN(u16\ smi_en_iop)$.

6.2.5.13 Enable SMI in ACPI software function

This function try to enable the generation of an SMI when ACPI software writes a 1 to the GBL_RLS bit, by setting bit2 in the SMI_EN register. The GBL_RLS bit (Global Release) is the bit2 in the PM1_CNT register. It is implemented by function $int\ enable_smi_BIOS_EN(u16\ smi_en_iop)$.

6.2.5.14 Enable SMI in the whole system function

This function try to enable the generation of SMI in the system upon any enabled SMI event, by setting bit0 in the SMI_EN register. When the SMI_LOCK bit is set, this bit cannot be changed. The SMI_LOCK bit is the bit4 in the GEN_PMCON_1 register. It is implemented by function $int\ enable_smi_gbl(u16\ smi_en_iop)$.

6.2.6 Disabling SMI Menu

As in previous section, it is offered a set of functions to disable SMI to be generated in the chipset by clearing specifics bits in registers, as APM_CNT register and PM1_CNT register. Also as in previous section, it is offer a function to disable an SMI globally in the chipset. In general, writing to those register requires high privileges. So, when using Linux OSs one can use the command line $iopl(3)$, this command changes the I/O privilege level of the calling process.

6.2.6.1 Disable SMI in xHCI function

This function try to disable an SMI to be generated by a xHCI device, by clearing bit31 in the SMI_EN register. It is implemented by function $int\ disable_smi_xHCI$ $(u16\ smi_en_iop)$.

6.2.6.2 Disable SMI in Intel Management Engine function

This function try to disable an SMI to be generated by Intel Management Engine, by clearing bit30 in the SMI_EN register. It is implemented by function $int\ disable_smi_Intel_ME(u16\ smi_en_iop)$.

6.2.6.3 Disable SMI in GPIO unlock function

This function try to disable the GPIO registers lockdown logic to launch an SMI, by clearing bit27 in the SMI_EN register. It is implemented by function $int\ disable_smi_GPIO_Unlock(u16\ smi_en_iop)$.

6.2.6.4 Disable SMI in Intel specific EHCI SMI logic function

This function try to disable Intel-Specific EHCI SMI logic to cause an SMI, by clearing bit18 in the SMI_EN register. It is implemented by function $int\ disable_smi_Intel_USB2_EN(u16\ smi_en_iop)$.

6.2.6.5 Disable SMI in legacy EHCI logic function

This function try to disable legacy EHCI logic to cause an SMI, by clearing bit17 in the SMI_EN register. It is implemented by function $int\ disable_smi_legacy_USB2_EN(u16\ smi_en_iop)$.

6.2.6.6 Disable SMI in periodic function

This function try to disable the chipset to generate an SMI periodically, by clearing bit14 in the SMI_EN register. It is implemented by function $int\ disable_smi_periodic$ $(u16\ smi_en_iop)$.

6.2.6.7 Disable SMI in TCO logic function

This function try to disable the TCO logic to generate an SMI, by clearing bit13 in the SMI_EN register. It is implemented by function $int\ disable_smi_TCO_logic$ $(u16\ smi_en_iop)$.

6.2.6.8 Disable SMI in trap access to micro-controller range function

This function try to disable the chipset to trap accesses to the micro-controller range and generate an SMI, by clearing bit11 in the SMI_EN register. It is implemented by function $int\ disable_smi_MCSMI(u16\ smi_en_iop)$.

6.2.6.9 Disable SMI in SMI Timer function

This function try to starts Software SMI Timer, by clearing bit6 in the SMI_EN register. When the timer expires, an SMI is generated. It is implemented by function $int\ disable_smi_SWSMI_TMR_EN(u16\ smi_en_iop)$.

6.2.6.10 Disable SMI in APM_CNT register function

This function try to disable writing to the APM_CNT register to generate an SMI, by clearing bit5 in the SMI_EN register. It is implemented by function $int\ disable_smi_APM_CNT(u16\ smi_en_iop)$.

6.2.6.11 Disable SMI in PM1_CNT register function

This function try to disable the PM1_CNT register to generate an SMI, by clearing bit4 in the SMI_EN register. It is implemented by function $int\ disable_smi_SLP_SMI_EN(u16\ smi_en_iop)$.

6.2.6.12 Disable SMI in legacy USB circuit function

This function try to disable legacy USB circuit to cause an SMI, by clearing bit3 in the SMI_EN register. It is implemented by function $int\ disable_smi_legacy_USB_EN(u16\ smi_en_iop)$.

6.2.6.13 Disable SMI in ACPI software function

This function try to disable the generation of an SMI when ACPI software writes a 1 to the GBL_RLS bit, by clearing bit2 in the SMI_EN register. The GBL_RLS bit (Global Release) is the bit2 in the PM1_CNT register. It is implemented by function $int\ disable_smi_BIOS_EN(u16\ smi_en_iop)$.

6.2.6.14 Disable SMI in the whole system function

This function try to disable the generation of SMI in the system upon any enabled SMI event, by clearing bit0 in the SMI_EN register. When the SMI_LOCK bit is set, bit0 in the SMI_EN register cannot be changed. The SMI_LOCK bit is the bit4 in the GEN_PMCON_1 register. It is implemented by function $int\ disable_smi_gbl(u16\ smi_en_iop)$.

6.2.7 Generating an SMI Menu

A set of functions to generate SMI in the chipset immediately (it means without waiting for a system management event) are offered. For example, an SMI can be programmed to be generated at every 64 seconds. Again, in general, writing to

those register requires high privileges. So, when using Linux OSs one can use the command line $iopl(3)$, this command changes the I/O privilege level of the calling process.

6.2.7.1 Generating an SMI by Periodic Seconds Function

This function try to control the rate at which periodic SMI is generated, by handling bit1 and bit0 in GEN_PMCON_1 register, following the schema: bit1 = 0 and bit0 = 0, then rate = 64 seconds; bit1 = 0 and bit0 = 1, then rate = 32 seconds; bit1 = 1 and bit0 = 0, then rate = 16 seconds; and bit1 = 1 and bit0 = 1, then rate = 8 seconds. It is implemented by function $int\ generate_smi_periodic(u16\ gen_pmcon_1_status_x,$ $u16 address_gen_pmcon_1, int\ rate)$. Before executing that function, it is a good practice to verify the GBL_SMI_EN bit (bit0 in the SMI_EN register) to check if SMIs are globally enabled.

6.2.7.2 Generating an SMI by APM_CNT Register Function

This function try to generate an SMI by writing any value to the APM_CNT register. Write any value to this register generates an SMI when the APMC_EN bit (bit5 in SMI_EN register) is set. It can be accomplished by a command line as $outb(0x00, 0xb2)$, where the first parameter is the value to be written and the second one is the address of the register. The first parameter can be any 8-bit value. Before executing that command, it is important to verify if the APMC_EN bit is set.

6.2.7.3 Generating an SMI by PM1_CNT register Function

This function try to generate an SMI by setting the SLP_EN bit, bit13 in the PM1_ CNT register. Set SLP_EN bit generates an SMI when the SLP_SMI_EN bit (bit4 in SMI_EN register) is set. It can be verified if an SMI was generated by checking if the SLP_SMI_STS bit (bit4 in the SMI_STS register) is set. It is implemented by function $int\ generate_smi_SLP_SMI_EN(u16\ smi_en_iop, u16\ smi_sts_iop, u16$ $pm1_cnt_iop)$. Before executing that command, it is important to verify if the SLP _SMI_EN bit is set.

6.2.7.4 Generating an SMI by ACPI software Function

This function try to generate an SMI by setting the GBL_RLS bit, bit2 in the PM1_ CNT register. Set GBL_RLS bit generates an SMI when the BIOS_EN bit (bit2 in SMI_EN register) is set. It can be verified if an SMI was generated by checking if the BIOS_STS bit (bit2 in the SMI_STS register) is set. It is implemented by function $int\ generate_smi_BIOS_EN\ (u16\ smi_en_\ iop, u16\ smi_sts_iop, u16\ pm1_cnt_iop)$. Before executing that command, it is important to verify if the GBL_RLS bit is set.

6.3 Manager Module Computational Experiments

In this section we perform experiments in different machines and chipsets to test the manager module ability to execute its functions. We test some of the registers, which are of primary interest when considering to develop an $SBST$. We consider

in this experiment the different chipsets listed in table 3.1. All figures and reports are presented as they are presented by the manager module.

6.3.1 Reporting SMRAMC Register Status Machine 12

The 8-bit SMRAMC register is potentially the most important register related to SMM, since it acts as an access control to the isolated SMM memory space the mem_{SMRAM}. Then, in this section, we follow the path "Probe and report this machine" and "Report SMRAMC register" in the manager module menu to invoke the function to report that register.

Figure 6.3 reports the SMRAMC status in the very moment of function *print_smramc(uint8_t smramc_status)* was launched in the target machine.

```
=================================
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
=================================
|RES|OPN|CLS|LCK|SMR|C_BASE_SEG |
=================================
|RES| 0 | 0 | 1 | 1 | 0 | 1 | 0 |
=================================
```

Figure 6.3: SMRAMC status reported by the manager module for machine 12. Offset 88H.

Below, we can see the manager automatic interpretation of the SMRAMC status. That report indicates that we cannot access the mem_{SMRAM} to inject any code and we cannot modify the SMRAMC status. This is consistent with section A.1, where is stated that in old machines (released up to 2004), there is the situation where D_LCK is cleared and the D_OPEN is set after the boot up process. Since this chipset was released in 2012, this situation is corrected. We say "corrected" because the previous situation opened a venue for attacks, as described in section 3.5.

- SMRAM is closed.
- SMRAM is accessible for code and data references.
- Writing access to SMRAMC register is blocked.
- The use of bits D_OPEN, D_CLS and D_LCK is enabled.
- Bit 7 is reserved.
- Bit 2, 1 and 0 are hardwired too 010B.

6.3.2 Reporting SMRAMC Register Status Machines 1 to 5 and 13

The 8-bit SMRAMC register status for machines 1 to 5 an 13 is reported in figure 6.4. As can be seen, all of allowing access to SMRAM. Those machines have the linux CentOS i386 version 5.11 installed.

```
==================================
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
==================================
|RES|OPN|CLS|LCK|SMR|C_BASE_SEG |
==================================
|RES| 0 | 0 | 0 | 1 | 0 | 1 | 0 |
==================================
```

Figure 6.4: SMRAMC status reported by the manager module for machines 1 to 5 and 13. Offset 90H for machines 1 to 5 and 9dH for machine 13.

ahead, it is presented the manager automatic interpretation of the SMRAMC status. The report indicates that we can access the mem_{SMRAM} to move code and data to there and we can modify the SMRAMC status. This is consistent with section A.1, where is stated that in old machines (released up to 2004), there is the situation where D_LCK is cleared and the D_OPEN is set after the boot up process.

- SMRAM is closed.
- SMRAM is accessible for code and data references.
- Writing access to SMRAMC register is opened.
- The use of bits D_OPEN, D_CLS and D_LCK is enabled.
- Bit 7 is reserved.
- Bit 2, 1 and 0 are hardwired too 010B.

Since D_LCK is cleared we can easily change its value by software. For example, the small set of instruction below can be used to lock the register SMRAMC, using the **libpci**. For the manager works full-fledged it needs to be executed with root rights.

```
unsigned int  D_LCK = 0x01 << 4;
struct pci_dev *smramc;
uint8_t smramc_status;
unsigned int SMRAM_OFFSET = 0x90;
smramc_status =  pci_read_byte(smramc, SMRAM_OFFSET);
pci_write_byte(smramc, SMRAM_OFFSET, (smramc_status & ~D_LCK));
```

6.3.3 Reporting PMBASE and SMM Related Registers Status Machine 12

This section reports the registers PMBASE, SMI_EN, SMI_STS, GPE0_EN, GPE0_STS, ALT_GPI_SMI_EN, ALT_GPI_SMI_STS, ALT_GPI_SMI_ENS and ALT_GPI_SMI_STS2. Those registers are quite important to the SMM operation, being related to devices and events whose generate SMI. We follow the path “Probe and report this machine” and “Report PMBASE and SMM related registers” in the manager module menu to invoke the function *int print_pmbase(struct pci_dev *sb, struct pci_access *pacc)* to report those register. Figure 6.5 brings all registers status with values in hexadecimal. The next eight sections report and detail each register.

REGISTER	ADDRESS	CONTENT
PMBASE	PCI 0x1f+0x40	0x00000400
GPE0_STS	0x0420	0x0000000006000000
GPE0_EN	0x0428	0x0000000040000046
SMI_EN	0x0430	0x80020023
SMI_STS	0x0434	0x00004100
ALT_GPI_SMI_EN	0x0438	0x0000
ALT_GPI_SMI_STS	0x043a	0x0600
ALT_GPI_SMI_EN2	0x045c	0x0000
ALT_GPI_SMI_STS2	0x045e	0x0000

Figure 6.5: All PMBASE SMM Related Registers Status.

6.3.4 Reporting SMI_EN Register Status Machine 12

In this section, we describe the manager report of the 32-bit SMI_EN register. Figure 6.6 presents the SMI_EN register status, which is equivalent to the value 80020023H presented in figure 6.5. After, there is the manager interpretation of the register status.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1

Figure 6.6: SMI_EN Register Status.

- Bit xHCI_SMI_EN [bit 31] is set: xHCI is able to generate an SMI.
- Bit ME_SMI_EN [bit 30] is cleared: ME cannot generate an SMI.
- Bit GPIO_UNLOCK_SMI_EN [bit 27] is cleared: PCH will NOT generate an SMI when bit GPIO_UNLOCK_SMI_STS [bit 27] at SMI_STS register is set.
- Bit INTEL_USB2_EN [bit 18] is cleared: Intel-Specific EHCI SMI logic cannot to cause an SMI.

- Bit LEGACY_USB2_EN [bit 17] is set: Legacy EHCI SMI logic is able to cause an SMI.
- Bit PERIODIC_EN [bit 14] is cleared: PCH will NOT generate an SMI when bit PERIODIC_STS [bit 14] at SMI_STS register is set.
- Bit TCO_EN [bit 13] is cleared: TCO logic cannot generate an SMI [Except if bit NMI2SMI_EN [bit 9] in TCO1_CNT register is set].
- Bit MCSMI_EN [bit 11] is cleared: PCH cannot trap accesses to the microcontroller range (62H or 66H) and generate an SMI.
- Bit SWSMI_TMR_EN [bit 6] is cleared: Software SMI timer is disabled or reset.
- Bit APMC_EN [bit 5] is set: Writes to the APM_CNT register will cause an SMI.
- Bit SLP_SMI_EN [bit 4] is cleared: Generation of SMI on SLP_EN [bit 13] in PM1_CNT register is disabled.
- Bit LEGACY_USB_EN [bit 3] is cleared: Legacy USB circuit cannot generate an SMI.
- Bit BIOS_EN [bit 2] is cleared: An SMI cannot be generated when ACPI software set the bit GBL_RLS [bit 2] in PM1_CNT register.
- Bit EOS [bit 1] is set: SMI signal will be de-asserted for 4 PCI clocks before its assertion.
- Bit GBL_SMI_ES [bit 0] is set: The generation of SMI in the system upon any enabled SMI event is enabled. When SMI_LOCK bit is set, this bit cannot be changed!

6.3.5 Reporting SMI_STS Register Status Machine 12

In this section, we detail the manager report of the 32-bit SMI_STS register. Figure 6.7 presents the SMI_STS register status, which is equivalent to the value 00004100H presented in figure 6.5. After, there is the manager interpretation of the register status.

```
=================================================================
|31 |30 |29 |28 |27 |26 |25 |24 |23 |22 |21 |20 |19 |18 |17 |16 |
=================================================================
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
=================================================================
|15 |14 |13 |12 |11 |10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
=================================================================
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
=================================================================
```

Figure 6.7: SMI_STS Register Status.

- Bit GPIO_UNLOCK_SMI_STS [bit 27] is cleared: GPIO registers lockdown logic is NOT requesting an SMI [Writing a 1 to this bit clears it to 0].
- Bit SPI_STS [bit 26] is cleared: SPI logic is NOT generating an SMI.
- Bit MONITOR_STS [bit 21] is cleared: The trap/SMI logic has NOT caused the SMI.
- Bit PCI_EXP_SMI_STS [bit 20] is cleared: No PCI Express SMI event has occurred.
- Bit INTEL_USB2_STS [bit 18] is cleared: The INTEL_USB2_EN [bit 18] in SMI_EN register is cleared and Intel-Specific EHCI SMI logic is NOT able to cause an SMI.
- Bit LEGACY_USB2_STS [bit 17] is cleared: The LEGACY_USB2_EN [bit 17] in SMI_EN register is cleared and Legacy EHCI SMI logic is NOT able to cause an SMI.
- Bit SMBUS_SMI_STS [bit 16] is cleared: The SMI was NOT caused by a SMBUS related event [Writing a 1 to this bit clears it to 0].
- Bit SERIRQ_SMI_STS [bit 15] is cleared: The SMI was NOT caused by the SERIRQ decoder.
- Bit PERIODIC_STS [bit 14] is set: PCH will generate an SMI when bit PERIODIC_EN [bit 14] at SMI_EN register is set. This bit is set a the rate determined by the PER_SMI_SEL bits [Writing a 1 to this bit clears it to 0].
- Bit TCO_STS [bit 13] is cleared: The SMI was NOT caused by TCO logic.
- Bit DEVMON_STS [bit 12] is cleared: SMI was not caused by Device Monitor.
- Bit MCSMI_STS [bit 11] is cleared: There has been NO access to the power management microcontroller range (62H or 66H) [Writing a 1 to this bit clears it to 0].
- Bit GPE1_STS [bit 10] is cleared: SMI was NOT generated by a GPI assertion.
- Bit GPE0_STS [bit 9] is cleared: SMI was NOT generated by a GPE event.
- Bit PM1_STS_REG [bit 8] is set: SMI was generated by a PM1_STS event.
- Bit SWSMI_TMR_STS [bit 6] is cleared: Software SMI timer has NOT expired.
- Bit APMC_STS [bit 5] is cleared: No SMI was generated by a write access to the APM_CNT register with bit APMCH_EN bit set [Writing a 1 to this bit clears it to 0].
- Bit SLP_SMI_STS [bit 4] is cleared: No SMI was generated by a write of 1 to SLP_EN bit when SLP_SMI_EN bit is also set [Writing a 1 to this bit clears it to 0].

- Bit LEGACY_USB_STS [bit 3] is cleared: SMI was NO generated by USB Legacy event.
- Bit BIOS_STS [bit 2] is cleared: No SMI generated due to ACPI software requesting attention [Writing a 1 to this bit clears it to 0].

6.3.6 Reporting GPE0_EN Register Status Machine 12

In this section, we detail the manager report of the 64-bit GPE0_EN register. Figure 6.8 presents the GPE0_EN register status, which is equivalent to the value 0000000040000046H presented in figure 6.5. After, there is the manager interpretation of the register status.

```
=================================================================
|63 |62 |61 |60 |59 |58 |57 |56 |55 |54 |53 |52 |51 |50 |49 |48 |
=================================================================
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
=================================================================
|47 |46 |45 |44 |43 |42 |41 |40 |39 |38 |37 |36 |35 |34 |33 |32 |
=================================================================
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
=================================================================
|31 |30 |29 |28 |27 |26 |25 |24 |23 |22 |21 |20 |19 |18 |17 |16 |
=================================================================
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
=================================================================
|15 |14 |13 |12 |11 |10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
=================================================================
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
=================================================================
```

Figure 6.8: GPE0_EN Register Status.

- Bit WADT_EN [bit 38] is cleared: Bit WADT_STS [bit 38] in GPE0_STS register is NOT enabled to be set to cause an SMI [WADT stands for Wake Alarm Device Timer].
- Bit GPI27_EN [bit 35] is set: Bit GPI27_STS [bit 35] in GPE0_STS register is NOT enabled to be set to cause an SMI.
- Bit PME_B0_EN [bit 13] is set: Bit PME_B0_STS [bit 13] in GPE0_STS register is NOT enabled to be set to cause an SMI.
- Bit PME_EN [bit 11] is set: Bit PME_STS [bit 11] in GPE0_STS register is NOT enabled to be set to cause an SMI.
- Bit SWGPE_EN [bit 2] is set: Bit SWGPE_STS [bit 2] in GPE0_STS register is enabled to be set to cause an SMI.

6.3.7 Reporting GPE0_STS Register Status Machine 12

In this section, we detail the manager report of the 64-bit GPE0_STS register. Figure 6.9 presents the GPE0_STS register status, which is equivalent to the value 0000000006000000H presented in figure 6.5. After, there is the manager interpretation of the register status.

```
=================================================================
|63 |62 |61 |60 |59 |58 |57 |56 |55 |54 |53 |52 |51 |50 |49 |48 |
=================================================================
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
=================================================================
|47 |46 |45 |44 |43 |42 |41 |40 |39 |38 |37 |36 |35 |34 |33 |32 |
=================================================================
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
=================================================================
|31 |30 |29 |28 |27 |26 |25 |24 |23 |22 |21 |20 |19 |18 |17 |16 |
=================================================================
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
=================================================================
|15 |14 |13 |12 |11 |10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
=================================================================
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
=================================================================
```

Figure 6.9: GPE0_STS Registers Status.

- Bit WADT_STS [bit 38] is cleared: No WADT signal. No SMI was generated [WADT stands for Wake Alarm Device Timer].
- Bit GPI27_STS [bit 35] is cleared: An SMI was NOT generated.
- Bit PME_B0_STS [bit 13] is cleared: An SMI was not generated.
- Bit PME_STS [bit 11] is cleared: An SMI was not generated.
- Bit SMB_WAK_STS [bit 7] is cleared: A wake event was NOT caused by the PCH's SMBus logic.
- Bit SWGPE_STS [bit 2] is is cleared: An SMI was not generated.

6.3.8 Reporting ALT_GPI_SMI_EN Register Status Machine 12

In this section, we detail the manager report of the 16-bit ALT_GPI_SMI_EN register. Figure 6.10 presents the ALT_GPI_SMI_EN register status, which is equivalent to the value 0000H presented in figure 6.5. After, there is the manager interpretation of the register status.

- GPI15 is NOT able to cause an SMI.
- GPI14 is NOT able to cause an SMI.

```
=================================================================
|15  |14  |13  |12  |11  |10  | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
=================================================================
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
=================================================================
```

Figure 6.10: ALT_GPI_SMI_EN Register Status.

- GPI13 is NOT able to cause an SMI.
- GPI12 is NOT able to cause an SMI.
- GPI11 is NOT able to cause an SMI.
- GPI10 is NOT able to cause an SMI.
- GPI9 is NOT able to cause an SMI.
- GPI8 is NOT able to cause an SMI.
- GPI7 is NOT able to cause an SMI.
- GPI6 is NOT able to cause an SMI.
- GPI5 is NOT able to cause an SMI.
- GPI4 is NOT able to cause an SMI.
- GPI3 is NOT able to cause an SMI.
- GPI2 is NOT able to cause an SMI.
- GPI1 is NOT able to cause an SMI.
- GPI0 is NOT able to cause an SMI.

6.3.9 Reporting ALT_GPI_SMI_STS Register Status Machine 12

In this section, we detail the manager report of the 16-bit ALT_GPI_SMI_STS register. Figure 6.11 presents the ALT_GPI_SMI_STS register status, which is equivalent to the value 0600H presented in figure 6.5. After, there is the manager interpretation of the register status.

- GPI15 is inactive!
- GPI14 is inactive!
- GPI13 is inactive!
- GPI12 is inactive!
- GPI11 is inactive!

```
=================================================================
|15 |14 |13 |12 |11 |10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
=================================================================
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
=================================================================
```

Figure 6.11: ALT_GPI_SMI_STS Register Status.

- GPI10 is active!
- GPI9 is active!
- GPI8 is inactive!
- GPI7 is inactive!
- GPI6 is inactive!
- GPI5 is inactive!
- GPI4 is inactive!
- GPI3 is inactive!
- GPI2 is inactive!
- GPI1 is inactive!
- GPI0 is inactive!

6.3.10 Reporting ALT_GPI_SMI_EN2 Register Status Machine 12

In this section, we detail the manager report of the 16-bit ALT_GPI_SMI_EN2 register. Figure 6.12 presents the ALT_GPI_SMI_EN2 register status, which is equivalent to the value 0000H presented in figure 6.5. After, there is the manager interpretation of the register status.

```
=================================================================
|15 |14 |13 |12 |11 |10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
=================================================================
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
=================================================================
```

Figure 6.12: ALT_GPI_SMI_EN2 Register Status.

- Bits 15 - 8 are reserved.
- GPI60 is NOT able to cause an SMI.

- GPI57 is NOT able to cause an SMI.
- GPI56 is NOT able to cause an SMI.
- GPI43 is NOT able to cause an SMI.
- GPI22 is NOT able to cause an SMI.
- GPI21 is NOT able to cause an SMI.
- GPI19 is NOT able to cause an SMI.
- GPI17 is NOT able to cause an SMI.

6.3.11 Reporting ALT_GPI_SMI_STS2 Register Status Machine 12

In this section, we detail the manager report of the 16-bit ALT_GPI_SMI_STS2 register. Figure 6.13 presents the ALT_GPI_SMI_STS2 register status, which is equivalent to the value 0000H presented in figure 6.5. After, there is the manager interpretation of the register status.

```
=================================================================
|15 |14 |13 |12 |11 |10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
=================================================================
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
=================================================================
```

Figure 6.13: ALT_GPI_SMI_STS2 Register Status.

- Bits 15 - 8 are reserved.
- GPI60 is inactive!
- GPI57 is inactive!
- GPI56 is inactive!
- GPI43 is inactive!
- GPI22 is inactive!
- GPI21 is inactive!
- GPI19 is inactive!
- GPI17 is inactive!

6.3.12 Reporting GEN_PMCON_1 Register Status Machine 12

This 16-bit register is important because it controls the access to bits in some of the previously discussed register, for example, it lock or unlock the access to the bit GBL_SMI_EN [Bit 0] in SMI_EN register. That bit is used to enable or disable SMI in this system.

Figure 6.14 presents the GEN_PMCON_1 register status, which is equivalent to the value 0E08H presented in figure 6.5. After, there is the manager interpretation of the register status.

```
=================================================================
|15 |14 |13 |12 |11 |10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
=================================================================
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
=================================================================
```

Figure 6.14: GEN_PMCON_1 Register Status.

- Bit SMI_LOCK_GP22 [bit 6] is cleared: Bit ALT_GPI22_SMI_EN [Bit 3] in ALT_GPI_SMI_EN2 registers unlocked.
- Bit SMI_LOCK_GP6 [bit 5] is cleared: Bit GPI6 [Bit 6] in ALT_GPI_SMI_EN register is unlocked.
- Bit SMI_LOCK [bit 4] is cleared: Bit GBL_SMI_EN [Bit 0] at SMI_EN register is unlocked. SMI CAN be enabled or disabled in this system.
- The rate at which periodic SMI is generated: 64 seconds [That control is set by software].

6.3.13 Verifying Support for SMRR Interface Machine 12

In this section we verify the support for the SMRR Interface, which contributes to enhance the security of SMRAM, mitigating attacks as “cache poisoning” described in section 3.5. We report below the manager module analysis after probing the IA32_MTRRCAP register. Note that this support is verified by core and our target machine is a quad-core processor. We follow the path “Verify supports for SMM” and “Verify support for SMRR Interface” in the manager module menu to invoke the function *int verify_smrr_support(void)* to report that support.

- SMRR Interface is supported in core 0.
- SMRR Interface is supported in core 1.
- SMRR Interface is supported in core 2.
- SMRR Interface is supported in core 3.

6.3.14 Verifying Support for Dual-Monitor Treatment Machine 12

In this section, we verify the support for the Dual-Monitor Treatment related to SMM, as described in section 3.3.4. Below, we can see the report provided by the manager module, after probing IA32_VMX_BASIC register to verify if that processor supports the Dual-Monitor Treatment. Note that this support is verified by core and our target machine is a quad-core processor. We follow the path "Verify supports for SMM" and "Verify support Dual-Monitor Treatment" in the manager module menu to invoke the function *int verify_stm_support(void)* to report the support.

- This processor supports the Dual-Monitor Treatment in core 0.
- This processor supports the Dual-Monitor Treatment in core 1.
- This processor supports the Dual-Monitor Treatment in core 2.
- This processor supports the Dual-Monitor Treatment in core 3.

6.3.15 Enabling and Disabling SMI Globally in the System Machine 12

In this section, we try to enable and disable SMI globally in the target machine by setting (to enable) or clearing (to disable) the GBL_SMI_ES [bit 0] in SMI_EN register. We try the follow steps to test the globally enable and disable functions: enable, disable, disable and enable. The objective is verify the initial status of the SMI globally and test if the manager can detect the globally enabled status and leave it as it is. After, if SMI is globally enabled, we try to disable. If successful, we try to disable again to verify it the manager can leave the status as it is. Finally, we try to globally enable SMI again.

Thus, we first try to globally enable SMI in the target machine, by following the path "Enabling SMI" and "ENABLE SMI IN THE WHOLE SYSTEM" in the manager module menu. The result of execution is presented below:

```
SMI_EN initial value: 0x80020023
SMI is already globally enabled!
Double-checking...
Global SMI is enabled at SMI_EN!
SMI_EN final value: 0x80020023
```

The manager starts probing the SMI_EN register and then getting the initial register value of 80020023H, which is consistent with figure 6.5. Since SMI is already globally enabled, no changes are made.

After, we try to globally disable SMI in the target machine, by following the path "Disabling SMI" and "DISABLE SMI IN THE WHOLE SYSTEM" in the manager module menu. The result of execution is presented below:

```
SMI_EN initial value: 0x80020023
Trying to disable SMI globally...
SMI is now globally disabled!
Double-checking...
Global SMI is disabled at SMI_EN!
SMI_EN final value: 0x80020022
```

The manager probes the SMI_EN register and get the initial register value of 80020023H, which is consistent with the previous execution. Then, it tries to globally disable SMI, by clearing the GBL_SMI_ES [bit 0]. It is successful, so the new register value is 80020022H (3H = 0011B = Enabled, 2H = 0010B = Disabled).

Now, we try to globally disable again SMI in the target machine, by following the path "Disabling SMI" and "DISABLE SMI IN THE WHOLE SYSTEM" in the manager module menu. The result of execution is presented below:

```
SMI_EN initial value: 0x80020022
SMI is already globally disabled!
Double-checking...
Global SMI is disabled at SMI_EN!
SMI_EN final value: 0x80020022
```

The manager probes the SMI_EN register and get the initial register value of 80020022H. Then, it tries to globally disable SMI, by clearing the GBL_SMI_ES [bit 0]. Since SMI is already globally enabled, no changes are made.

Finally, we try to globally enable back SMI in the target machine, by following the path "Enabling SMI" and "ENABLE SMI IN THE WHOLE SYSTEM" in the manager module menu. The result of execution is presented below:

```
SMI_EN initial value: 0x80020022
Trying to enable SMI globally...
SMI is now globally enabled!
Double-checking...
Global SMI is enabled at SMI_EN!
SMI_EN final value: 0x80020023
```

The manager verifies the SMI_EN register and obtain the value of 80020022H and then it tries to globally enable SMI, by setting the GBL_SMI_ES [bit 0]. It is successful, so the new register value is 80020023H (3H = 0011B = Enabled, 2H = 0010B = Disabled).

6.4 SBST Implementation and Evaluation

In the previous sections, we presented the manager module and its implementation details. That module is useful to understand and learn about a target chipset, mainly about the status of some SMM related registers.

From this section on, we consider the findings in chapter 2 and 3, what we have built in chapter 5 and what we learned in the previous sections to build our proof of concept.

6.4.1 Proof of Concept

Hypervisors are high privileged entities and play a main role in virtualised environments, enabling virtualisation in many hardware platforms. A great deal of research has been done to improve functionality, performance and usability of hypervisors, which gives rise to different hypervisor architectures proposals.

In recent years the focus of hypervisor's research has changed. Because of its importance in virtualised environments, hypervisors are a prime target for attacks.

In this way, different kinds of attacks have been arising, exploiting flaws in the hypervisor design and implementation and in the environment hosting it. Successful attacks have been performed against hypervisors, both commercial [102] and open source [145].

Then, research efforts on hypervisors have been concentrating in mitigating or eliminating threats, which hypervisors are prone to. However, it led attackers to look for other venues of attack, trying to tamper or circumvent the new protections, for example, by exploiting system components with high execution privileges and launching their attacks from there. One of those components is the System Management Mode (SMM), which is one of the most powerful and resourceful system component in x86 platforms.

In this work we aimed to build a proof of concept to the generic architecture described in chapter 5 and to meet the requirements proposed in chapter 4. Then we built all the basic code as defined in 2.4 and a payload as defined in 2.9 to measure the integrity of a Xen hypervisor configuration file. We have chose that file as we could chose any other one, it is just to test our proof-of-concept. In developing that proof-of-concept, we follow the algorithms designed in chapter 5.

The Measurement target file: xend-config.sxp file is one the configuration files of Xen and many important configurations are done there. For instance, configure an IP address, enable migration, enable scripts to run, change memory configurations and so forth. It is located in /etc/xen [137].

6.4.2 Implementation particularities and experiments

In this section, we briefly discuss some implementation particularities from our computing experiments.

In our proposed architecture, we recommended the use of payloads to make the *sbst* code smaller and to deal with the maximum latency time issue. About the implementation, we recommend that the payload should be written partially in Assembly language. It may help meeting the requirements about size and maximum latency [105]. Since portability is a difficult issue when using Assembly [125], the payload strategy would have an advantage to make each payload suitable for a given machine model. For the basic code part, the C language is suitable.

We implemented our proof-of-concept in C language and compile it using gcc in the linux Centos i386 5.11. Our source code is 12723 bytes and the compiled file for 32-bits platform is 13780 bytes, considering the discussion in section 5.4.1. Then, we met the requirement *r1* - Small. As described in algorithm 5.5 we are using the SHA-256 to measure the integrity. We use the implementation of the SHA-512 available in [4]. Since we will call our tool from DRAM, we need to simulate an SMI in the system. We did it by creating a launcher program to call our tool 5.000 times with a chance of 25% of executing the tool each time. In our basic code we implemented a function called **map_memory** to map our persistent data to the SMRAM space area. We did it the same way as [23] and [45] by using the *"/dev/mem"* resources.

We use the functions in the *librt.so* "Real Time" shared library to get the execution time of the functions. We did these experiments for machines 1 to 3 and 13 at table 3.1, since they have the SMRAM opened and all of them are 32-bits machines, so we use the figure 3.2 memory scheme.

The main function in basic code is **map mamory** which map the memory which will be used by our persistent data acording to figure 3.2. We have 32768 bytes after the state save map area to use for persisting our data So, we map the memory from address *00040000H* to address *0x00048000H*, follow the scheme below.

```
variable monitor        = SMBASE(30000) + 8000H + 8000H to 40000H (1 byte)
variable index          = SMBASE(30000) + 8000H + 8001H to 40001H (1 byte)
variable previous_round = SMBASE(30000) + 8000H + 8002H to 40002H (1 byte)
variable hashed         = SMBASE(30000) + 8000H + 8003H to 40003H (1 byte)
variable address        = SMBASE(30000) + 8000H + 8004H to 40007H (4 bytes)
variable sized_section  = SMBASE(30000) + 8000H + 8008H to 40009H (2 bytes)
variable total_data_load= SMBASE(30000) + 8000H + 800aH to 4000bH (2 bytes)
variable digest         = SMBASE(30000) + 8000H + 800cH to 4008bH (1 byte * 128)
data_loaded             = SMBASE(30000) + 8000H + 808cH to 48000H (1 byte * 32628)
```

The main function in the payload are: **prepare for hashing.** Execute tasks to prepare data to be loaded from the file to the SMRAM for being hashed. **Compute SHA-256.** Compute the 256-bit digest. **Move digest.** Move the computed digest from SMRAM to DRAM and from DRAM to SMRAM. This is associated to the function **prepare for hashing**. **Verify hash.** Compute the new digest from the fresh data got from the file and compare it with the digest stored in the SMRAM. **Move data from file to SMRAM.** This function get the data in the target file and store it in the designed area at SMRAM.

Table 6.1 shows the time execution for the experiments with machines 1, 2, 3 and 13, according to table 3.1. Any time the tool verify the integrity of the Xen file, a message is showed to the user so it can decide which action take.

Table 6.1: **Execution time.** In this experiment, we are considering the basic code execution plus payload execution plus the maximum latency for each machine considered reported by the BITS TOOL at table 3.1. Values are tabulated in microseconds.

OEM	**bc**	**payload**	Adding max latency
1 - Gigabyte	510	118	1481
2 - Gigabyte	408	130	993
3 - Gigabyte	472	140	930
13- Compaq	560	175	759

6.5 SBST Limits and Constraints

The tool was implemented assuming that it is embedded in the SMI handler code and then deployed in the BIOS set of programs and finally laid out in the SMRAM space during the bootup process.

The idea was to insert our proof of concept code in the *smihandler.c* from coreboot 4.4 and then compile it, in such a way that after finishing its tasks the **smi handler** code would call our tool as specified in chapter 5. Since this was not possible, we assume that our tool is inside of the SMI handler, so we can kept meeting our requirement *r4* - Cooperative. Since we did not find a way to embed our proof of concept in the BIOS in such a way that when the bootup process occurs our tool would be embedded in the SMRAM space we need another assumption that

it is done so we can test our proof-of-concept into DRAM without jeopardise our requirement *r5* - Isolated. But, to make our experiments more *real*, we consider the time the **smi handler** requires to execute at each platform in our experiments, considering the measurements reported in table 3.1.

6.6 Manager Limits and Constraints

The manager module is chipset specific. To work properly, it is required to codify the specificities of the target chipset in the manager, as the offset address of base address registers. the **PMBASE** is an example of register, whose content is a base address for many others registers, as **SMI_EN** and **SMI_STS**. Moreover, as we saw in chapter 3 and appendix A each chipset has its own set of register. So it is necessary to check the target chipset documentation to implement functionalities to probe the registers related to SMM in the machine considered. The manager implementation reported in this text was codified to target machine 12 at Table 3.1. That machine has the linux CentOS amd64 version 5.11 installed and for the manager works full-fledged it needs to be executed with root rights. However, we adapted and experimented the manage module with Machines 1 to 5 and 13. Anyway, the manager module is able to work in many different chipsets and different processors families and generations with slight adaptation.

6.7 Discussion

In this chapter we discussed the manager module implementation and the functions designed in this module to probe and research a platform to understand and learn about the environment where the tool will be deployed. It was implemented to target the machine 12 at table 3.1. That machine has the linux CentOS amd64 version 5.11 installed and for the manager works full-fledged it needs to be executed with root rights. it was also adapted and experimented with Machines 1 to 5 and 13.

We also discussed the SBST implementation. We develop the tool considering the characteristics of SMM. Two new assumptions were necessary in this point: we assume that our tool is inside of the smi handler since we could not solve compilations problems with the coreboot version we had at hands and we assumed that when the bootup process occur our tool would be embedded in the SMRAM space so we can test our tool, since we did not find a way to embed our proof of concept in the BIOS in such a way that when the bootup process occur our tool would be embedded in the SMRAM space. We can observe total time (table 6.1) considering the basic code execution plus payload execution plus the maximum latency for each machine considered reported by the BITS TOOL at table 3.1 is quite above the maximum 150 μs recommend by intel.

6.8 Summary

In this chapter we have considered the implementation of the manager module and the SBST and discussed particularities about those implementations. Although all

machines in the experiments are reported quite above the maximum 150 μs recommend by intel we can observe in table 3.1 that all those machines, but 13 had failed in the maximum latency test. So we believe it is not a big problem at all. Another thing is in a more realistic environment our tool would execute with more processor resources than in our experiments, since some other operating systems processes were competing for resources with our tools. Such a situation would not happen in a more realistic experiment. Moreover, by design, we may regulate or calibrate the amount of data our tool will load and verify at each round, as described in chapter 5. So those time can decrease at our will.

Observing the experiments, we can see that the manager is useful to learn about the target chipset, for example the status of SMM related registers.

As discussed in chapter 3 the impact of the SMM latency is not quite strict. If it was, we believe the machines at table 3.1 should pass the BITS TEST. Anyway, an developer can decide to implement the tool or not observing the information reported in the table.

This chapter discussed details about the manager module and how it is uselful in the context of this work and about implementation particularities of the manager module and the $SBST$ implementation, detailing the function in the $SBST$ and presenting experiments. Here we discussed the limits and constraints of the tool. We also presented our proof of concept, reporting the experiments conduct with it. In the next chapter we make a conclusion of this work and indicate future works in the same area as the one reported in this thesis.

Chapter 7

Conclusion

In this work we investigated the Sistem Managment Mode (SMM), presenting a detailed review and description of its resources and components. The Attacks against the SMM, misuses and attacks using SMM as a platform to lunch stronger attacks were analysed and presented together with a discussion about their feasibility nowadays and ways to thwart them. We also analyse works using SMM for security purposes presenting opportunities to improve them.

The chipset is a specially complex component in this scenario. Different versions of them are around and some part of the chipsets, as firmwares and device drivers, are OEM specific, which tangling even more the relation among components.

We identify that although there are many SMM-based security tools, SMM and its resources were designed by Intel and are implemented to be use by system firmware (as OEM and BIOS manufacturer codes), not by "general-purpose" systems software. So, an $SBST$ is a use of SMM for general-purpose. Such a recommendation aims to preserve the management SMM functions and avoid that such a "General-purpose" software violates the limits and constraints of SMM resources and interfere in the correct functioning of SMM components. So, this research established a set comprise of eight requirements, which a security tool must met to overcome the SMM limits and constraints and allow the use of SMM for "Security-purpose", without damaging the designed features of the SMM.

We pointed out the SMI handler is potentially the more powerful software artefact in Intel architecture. In fact, any security tool (or any software) capitalising on SMM must emulate the SMI handler in some way.

The set of requirements to use SMM for security purposes were established after analysing the SMM features, attacks, misuses, the security implementations, capitalising on SMM and the threat model (10 threats identified) built from the analysis of the current SMM-based security tools architecture and model. From those requirements, a generic architecture for SMM-Based Security Tools was proposed. The implementation of this architecture was done in a proof of concept designed to have two modules: a manager and an agent. The manager module is used for learning about and researching a target machine, as for probing the registers related to SMM. The manager can be used in a machine endowed with the same chipset of the target machine. So, it does not need to be deployed in the SMM memory or even in the target machine. The agent basically comprises of two parts: a basic code embodying management functions and a payload. The payload implements the security function intended for the $SBST$ and can be changed for another payload with a different security task to give more flexibility to our $SBST$.

We reported the proof-of-concept and the experiments conduct with it and with

the manager module, presenting the constraint, limits and results of them.

We concludes that eight requirements must be met to build a SMM-based tool for security purposes, which can be developed according to the architecture proposed:

- ***r1* - Small.** There are 32512 bytes available for the SMI handler code, data, heap and stack. Thus, a SMM-based security application should be small enough to fit in the minimum size available.
- ***r2* - Fast.** Intel specifies that the SMI latency must be less than 150 μs to minimise the risk of system executive software time-outs. So we designed algorithms based in payloads to deal with that requirement.
- ***r3* - Persistent.** The SMRAM is volatile. A reboot or system restart will clean the whole SMRAM content. So, the SMM related code and data need to be loaded again. Thus, the design of a SMM-based security tool needs to consider that the tool must be embedded in the BIOS or equivalent entity.
- ***r4* - Cooperative.** The SMI handler functions need to be preserved since they have important tasks to perform. Any $SBST$ must preserve the original SMI handler functions and code, by adding its own code to the SMI handler and not overwriting any part of it. Since when entering SMM the processor looks for the first instruction to be executed at the address SMBASE (register) + 8000H (by default 38000H) in SMRAM, where the SMI handler is located, which implies that any SMM-based security tool must be a modified version of the SMI handler.
- ***r5* - Isolated.** SMI handler, and consequently any $SBST$ needs to be protected by isolation and its code and data, even temporary, should be kept in the SMRAM, since the main reason for using SMM is to benefit from its powerful resources, such as isolation and transparency. Moreover code and data outside of SMRAM can be tampered.
- ***r6* - Resistant.** The SMRR Interface, if available in the target chipset, should be used to protect the related MTRRs registers and thwart the "cache poisoning" attack and any other attacks.
- ***r7* - SMI-independent.** To start any SMM-based security tool, an SMI needs to be generated. To avoid an attacker tries to deny an SMI to be triggered by well-know ways as by writing to the Programmed I/O Port 0xB2H, an SMM-based security tool should take advantage from any SMI generated to start executing its job. Anyway, any SMI will make the processor execute the $SBST$ even if the tool is not designed to act that way, according to the implication of requirement *r4*.
- ***r8* - Complete.** As discussed before Some tools need to keep part of their code in the system executive software. For example, HyperSentry uses an agent deployed in the hypervisor code base. While that model of architecture overcomes some SMM limitations, they lose the main benefit from using SMM: isolation and transparency. Then, the SMM-based security tool must have all

functionalities to execute its tasks and all needed data completely deployed in the SMRAM.

The requirements above were established to deal with the SMM limits and constraints and to mitigate the threats identified in our threat model. We classify the requirements $r1$, $r2$, $r3$, $r4$ and $r7$ (small, fast, persistent, cooperative and SMI-independent, respectively) as functional requirements, since they are related to the functioning of $SBST$ and must be met to overcome the limitations and constraints of SMM. Requirements $r5$, $r6$ and $r8$ (isolated, resistant and complete, respectively) are classified as security requirements, since they must be met to mitigate the threats $SBST$s are prone to.

7.1 Directions for Future work

We consider that an $SBST$ can be used in real world scenarios, as long as, some issues as execution time and its impact in the system can be further investigated and addressed. So, we provide directions for future work in three sub-areas of this work, which considering the limitation of time and resources of our work could not be addressed, as:

7.2 Investigate the interaction of an $SBST$ with technologies in the chipset

Considering that Intel processors were the focus of this work, we presented the main technologies which might be present in the chipset of a target machine and which might interact with an $SBST$: Intel TXT, Intel VMX, Intel SGX, TPM, UEFI. Note that not all chipsets are endowed or offer support for all those technology, which means that not always we will find all of them working together in a chipset and that SGX technology is not yet commercially available.

Although we did not identify any issue related those technologies that can have any impact in our proposal, we judge that is noteworthy investigate our proposal when interacting with those technologies. For example: our proposal is based on BIOS. Then since the use of UEFI has been increasing, it worthwhile to investigate and adapt our proposal to be used in a UEFI scenario. Another example is how might TPM be used to improve the proposal in this work, as discussed in Hyper-Check [141]. A third example is Intel SGX which might be a good Intel answer for low-level security issues since it takes security to the processor level. It is not available yet commercially but the documentation available can help to identify in which extent our proposal may work cooperatively with SGX (see section 3.3.5).

7.3 Investigate the Impact of SMI Latency

Intel specifies a strict time limit for the SMI latency: it must be less than 150 μs to minimise the risk of system executive software time-outs [90]. However, as we can see from our experiments in section 3.2.4.1 described in table 3.1 only machines from 9 to 13 were able to keep below 150 μs. Thus, if the time limit for the SMI

latency is so important, why did not OEMs from machines 1 to 8 design their SMI handlers to stick to the time limit? Then investigating that issue is an important task to unveil, learn and use the SMM resources.

7.4 Optimize the Proof of Concept Execution Time

As we can see in table 6.1, the experiments with our proof-of-concept, considering the addition of time-elapsed of the execution of the basic code and the payload were close to 1 millisecond, which is too high considering the maximum latency. In the table we can see also the maximum latency time which is added to the final result but about it we can do nothing in terms of optimisation since it is the time designed by the OEM (the SMI handler execution time). So, optimisation on the basic code and reduce the workload of the payload should be the answer to make the $SBST$ faster, but further investigation is necessary.

7.5 Embed the Tool in a BIOS to Test It in a More Realistic Scenario

Since the resources limitations we had to make an assumption to met our *r3* - Persistent, since we had no means to embed our proof of concept in a real BIOS. As in HyperGuard [146], after perform the optimisations and code adjust we need to find some kind of cooperation or use coreboot with other tools to test the capacity of the tool to be embedded in a BIOS (or equivalent entity) and to be distributed in large scale.

Appendix A

Specific SMM Registers

In this appendix, we discuss other specifics SMM registers related to the chipsets considered in this work according to section 1.2 and table 3.1.

A.1 Chipset 1 Specific Registers

The chipset 1 specific registers related to SMM are: PMBASE, SMI_EN, SMI_STS, GEN_PMCON_1, ALT_GP_SMI_EN, GPI_ROUT, ALT_GP_SMI_STS, GPE0_EN, GPE0 _STS, APM_CNT and APM_STS. Those registers can vary their location, depending on the chipset considered. For example, in chipset 1, the register SMRAMC is located at PCI device 0, address offset 88H [92, 93]. But, in chipset 2, it is located at PCI device 0, address offset 90H [65]. In other chipset, as in Intel 845 chipset, the register SMRAMC is located at PCI device 0, address offset 9DH [69, 70].

PMBASE is the 32-bit Advanced Configuration and Power Interface (ACPI) base address register. It can be accessed at device 31, function 0, offset 40H-43H and sets the base address for ACPI I/O registers and other ones, including the **SMI_EN** and **SMI_STS**. Those registers can be accessed by PMBASE + offset; for instance, the SMI_EN at PMBASE + 30H and the SMI_STS at PMBASE + 34H [80, 81].

SMI_EN is the 32-bit SMI Control and Enable register. Its rightmost bit (GBL_ SMI_EN) indicates that SMI is enabled in the system (GBL_SMI_EN = 1) or not (GBL_SMI_EN = 0). A PCI reset event can reset this bit. This register is symmetrical to the SMI_STS, so they work together to generate and provide information about SMI. The other bits of this register enable which devices can trigger an SMI and other SMI related functions. Below, we describe the bits of interest for this research [80, 81]:

- **Bit xHCI_SMI_EN [bit 31].** When set this bit enables the extensible Host Controller Interface (xHCI) to cause an SMI. The xHCI is a host controller that supports up to four USB 3.0 ports.
- **Bit GPIO_UNLOCK_SMI_EN [bit 27].** Set this bit causes an SMI when the GPIO_UNLOCK_SMI_STS bit at SMI_STS register is set too.
- **Bit INTEL_USB2_EN [bit 18].** Set this bit enables Intel-Specific Enhanced Host Controller Interface (EHCI) SMI logic to cause SMI. The EHCI is a host controller that support USB high-speed USB 2.0.
- **Bit LEGACY_USB2_EN [bit 17].** Set this bit enables legacy EHCI logic to cause an SMI.

- **Bit PERIODIC_EN [bit 14].** Set this bit causes an SMI when the PERIODIC_STS bit at SMI_STS register is set too.
- **Bit TCO_EN [bit 13].** Set this bit enables the Total Cost of Ownership (TCO) logic to generate SMI. There are a huge set of register related to TCO.
- **Bit MCSMI_EN [bit 11].** Set this bit enables PCH to trap accesses to the microcontroller range (62H or 66H) and then generate an SMI.
- **Bit SWSMI_TMR_EN [bit 6].** Set this bit starts Software SMI Timer, such that when the SWSMI timer expires, the SWSMI_TMR_STS bit at SMI_STS is set and an SMI is generated.
- **Bit APMC_EN [bit 5].** When this bit is set any write to the APM_CNT register will cause an SMI.
- **Bit SLP_SMI_EN [bit 4].** When this bit is set, this enable that set the SLP_EN bit (bit 13 in PM1_CNT register) cause an SMI.
- **Bit LEGACY_USB_EN [bit 3].** Set this bit enables legacy USB circuit to cause an SMI.
- **Bit BIOS_EN [bit 2].** Set this bit enables the generation of SMI when ACPI software writes a 1 to the GBL_RLS bit (bit 2 in PM1_CNT register).
- **Bit EOS [bit 1].** Set this bit cause an SMI signal to be deasserted for 4 PCI clocks before its assertion.
- **Bit GBL_SMI_EN [bit 0].** As explained before, set this bit enable SMI globally in the system.

SMI_STS is the 32-bit SMI Status register and indicates the device that have caused an SMI. As a general rule, when a bit is set in this register, whenever the correspondent bit is set in the SMI_EN register an SMI is generated, since those registers are symmetrical. However, in some chipsets not all bits in SMI_EN have correspondent in SMI_STS register that is the case of our target chipset 1: the SMI_STS register uses bit 26 (SPI_STS), bit 21 (MONITOR_STS), bit 20 (PCI_EXP_SMI_STS), bit 16 (SMBUS_SMI_STS), bit 15 (SERIRQ_SMI_STS), bit 12 (DEVMON_STS), bit 10 (GPE1_STS), bit 9 (GPE0_STS) and bit 8 (PM1_STS_REG); but the SMI_EN register does not. Conversely, the SMI_EN register uses bit 1 (EOS) and 0 (GBL_SMI_ES), but SMI_STS does not. Also, the bits activated or valid in those registers differ from one chipset to another. Below, we describe the bits of interest for this research [80, 81]:

- **Bit xHCI_SMI_STS [bit 31].** This bit is set when the extensible Host Controller Interface (xHCI) is requesting an SMI.
- **Bit GPIO_UNLOCK_SMI_STS [bit 27].** This bit is set when some of GPIO registers lockdown logic is requesting an SMI.
- **Bit SPI_STS [bit 26].** This bit is set whenever the Serial Peripheral Interface (SPI) logic is generating an SMI.

- **Bit MONITOR_STS [bit 21].** This bit will be set if the Trap/SMI logic has caused the SMI.
- **Bit PCI_EXP_SMI_STS [bit 20].** This bit will be set whenever a PCI Express SMI event occurred.
- **Bit INTEL_USB2_STS [bit 18].** If this bit is set, the INTEL_USB2_EN [bit 18] in SMI_EN register is set and the Intel-Specific EHCI SMI logic has caused an SMI. This bit refers to all integrated EHCIs.
- **Bit LEGACY_USB2_STS [bit 17].** If this bit is set, the LEGACY_USB2_EN [bit 17] in SMI_EN register is set and the Legacy EHCI SMI logic has caused an SMI. This bit refers to all legacy EHCIs.
- **Bit SMBUS_SMI_STS [bit 16].** If this bit is set, an SMI was caused by an SMBUS related event.
- **Bit SERIRQ_SMI_STS [bit 15].** This bit is set when an SMI was caused by the Serial Interrupt Request (SERIRQ) decoder.
- **Bit PERIODIC_STS [bit 14].** This bit is set at the rate indicated by the PER _SMI_SEL (bits 1 and 0 in GEN_PMCON_1 register). Then, if the PERIODIC _EN bit (bit 14 in SMI_EN register) is also set, an SMI is generated.
- **Bit TCO_STS [bit 13].** If this bit is set, an SMI was caused by the Total Cost of Ownership (TCO) logic.
- **Bit DEVMON_STS [bit 12].** This bit is set when an SMI was caused by a Device Monitor.
- **Bit MCSMI_STS [bit 11].** If this bit is set, there was an access to the power management microcontroller range (62H or 66H).
- **Bit GPE1_STS [bit 10].** This bit is set when an SMI was generated by a General Purpose Input/Output (GPI) assertion.
- **Bit GPE0_STS [bit 9].** This bit is set when an SMI was generated by an SMI was generated by a General Purpose Event (GPE0) event.
- **Bit PM1_STS_REG [bit 8].** This bit is set when an SMI was generated by a PM1_STS event.
- **Bit SWSMI_TMR_STS [bit 6].** This bit is set by hardware whenever the Software SMI Timer expires.
- **Bit APMC_STS [bit 5].** If this bit is set, this indicates that an SMI was generated due to a write access to the APM_CNT register when the APMC_EN bit (bit 5 in SMI_EN register) was set.
- **Bit SLP_SMI_STS [bit 4].** When this bit is set, this indicates an SMI was caused due to set the SLP_EN bit (bit 13 in PM1_CNT register) when SLP_SMI_EN bit (bit 5 in SMI_EN) is also set.

- **Bit LEGACY_USB_STS [bit 3].** If this bit is set, an SMI was caused by a USB Legacy event.

- **Bit BIOS_STS [bit 2].** This bit is set by hardware when the GBL_RLS bit (bit 2 in PM1_CNT register) is set. In this case, if the BIOS_EN bit (bit 2 in SMI_EN register) is also set, an SMI is generated.

GEN_PMCON_1 is the 16-bit General Power Management Configuration 1 register. It can be accessed at device 31, function 0, offset A0H. This register has two main functions related to SMI. First, as said before the GBL_SMI_EN bit (bit0 in SMI_EN register) enable SMI in the system upon any enabled SMI event [81]. However, this can be blocked if the SMI_LOCK bit (bit4) is set (SMI_LOCK = 1). When SMI_LOCK is set, it locks itself and just can be cleared by asserting the PLTRST# pin. This pin is located on the Power Management block in the PCH [80, 81]. Second, the PER_SMI_SEL bits (bit1 and bit0) controls the rate at which periodic SMI is generated, according to the following scheme: bit1 = 0 and bit0 = 0 equals 64 seconds, bit1 = 0 and bit0 = 1 equals 32 seconds, bit1 = 1 and bit0 = 0 equals 16 seconds and bit1 = 1 and bit0 = 1 equals 8 seconds. These bits are related to the PERIODIC_EN bit (bit14 in SMI_EN register) and to the PERIODIC_STS bit (bit14 in SMI_STS register).

ALT_GP_SMI_EN is the 16-bit Alternate GPI SMI Enable register, where each bit indicates enable the corresponding GPIO (General Purpose I/O) to cause an SMI. For example, set bit0 enables the GPI0 to cause an SMI, set bit1 enables GPI1 to cause an SMI and so forth. GPI stands for General Purpose Input. However, to generate an SMI some conditions must be true, as the corresponding GPI must be routed in the **GPI_ROUT** (GPI Routing Control Register) register. **GPI_ROUT** is a 32-bit register and routes the GPI to generate an SMI using pair of bits, where the first bit must be 1 and the second bit must be 0 (0:1). For example, bit0 and bit1 route GPI0 to generated an SMI with bit0 = 1 (first bit) and bit1 = 0 (second bit), bit2 and bit3 route GPI0 to generated an SMI with bit2 = 1 (first bit) and bit3 = 0 (second bit) and so forth. The **ALT_GP_SMI_STS** (16-bit), Alternate GPI SMI Status register, reports the corresponding GPIOs status. When a bit is set (equals to 1) the GPIO is active. For example, bit0 = 1, GPI0 is active; bit1 = 0, GPI1 is inactive, and so forth.

In more recent chipsets [91], those registers have a slightly different name: **ALT_GP_SMI_EN** and **ALT_GP_SMI_STS**. They also have three complementary registers: **ALT_GPI_SMI_EN2** (16-bit), **GPI_ROUT2** (32-bit) and **ALT_GPI_SMI_STS2** (16-bit) to extend the range of GPI (GPIs: 17, 21, 22, 43, 56, 57, 60) to cause SMI [91].

GPE0_EN is the 64-bit General Purpose Event 0 Enable register and **GPE0_STS** is the 64-bit General Purpose Event 0 Status register. Those registers are symmetrical to each other and they are used to manage wake events in the system. For some functions, an SMI will be generated. For example, when the BATLOW_EN bit (bit10 in GPI0_EN register) is set, it enables the BATLOW# signal to cause an SMI when the battery goes low. In this case, the hardware set the BATLOW_STS bit (bit10 in GPI0_STS register) as soon as the BATLOW# signal is asserted [80, 81].

APM_CNT is the 8-bit Advanced Power Management Control Port register and it is used to pass an APM command between the OS and the SMI handler. Writes to this port not only store data in the APMC register, but also generates an SMI

when the **APMC_EN** bit (bit5) in the SMI_EN register is set. **APM_STS** it the 8-bit Advanced Power Management Status Port register and it is used to pass data between the OS and the SMI handler [80, 81].

A.1.1 Chipset 1 Architectural Model-Specific Registers

Architectural Model-Specific Registers (MSR) are registers carrying over through IA-32 processors generations to Intel 64 processors and which will remain the same in future generations of processors. The name of those registers starts with "IA-32" (from Pentium 4 processors on) [87, 97].

Below, it is presented the Architectural Model-Specific Registers related to SMM in chipset 1 and the family or model of processor where they can be found. As we can notice, most of them are specific to a family/model, except the IA32_SMBASE register which is implemented in the same way in all platform that support it.

- **IA32_SMBASE:** All.
- **IA32_SMRR_PHYSBASE:** Silvermont Microarchitecture, Nehalem Microarchitecture, Sandy Bridge Microarchitecture and Xeon Phi Processors.
- **IA32_SMRR_PHYSMASK:** Silvermont Microarchitecture, Nehalem Microarchitecture, Sandy Bridge Microarchitecture and Xeon Phi Processors.
- **IA32_MTRRCAP:** Core Microarchitecture, Atom Processor Family, Silvermont Microarchitecture, Nehalem Microarchitecture, Sandy Bridge Microarchitecture, Skylake Microarchitecture, Xeon Phi Processors, Pentium 4 and Xeon Processors and Core Solo, Intel Core Duo Processors, and Dual-Core Intel Xeon Processor LV.
- **IA32_VMX_BASIC:** Core Microarchitecture, Atom Processor Family, Silvermont Microarchitecture, Nehalem Microarchitecture, Sandy Bridge Microarchitecture, Xeon Phi Processors, Pentium 4 and Intel Xeon Processors and Core Solo, Intel Core Duo Processors and Dual-Core Intel Xeon Processor LV.
- **IA32_VMX_MISC:** Core Microarchitecture, Atom Processor Family, Silvermont Microarchitecture, Nehalem Microarchitecture, Sandy Bridge Microarchitecture, Xeon Phi Processors, Pentium 4 and Intel Xeon Processors and Core Solo, Intel Core Duo Processors and Dual-Core Intel Xeon Processor LV.
- **IA32_SMM_MONITOR_CTL:** Pentium 4 and Intel Xeon Processors.
- **IA32_DEBUGCTL:** Core Microarchitecture, Atom Processor Family, Silvermont Microarchitecture, Nehalem Microarchitecture, Sandy Bridge Microarchitecture, Haswell or Haswell-E microarchitectures, Xeon Phi Processors and Core Solo, Intel Core Duo Processors and Dual-Core Intel Xeon Processor LV.
- **IA32_PERF_CAPABILITIES:** Core Microarchitecture, Atom Processor Family, Silvermont Microarchitecture, Nehalem Microarchitecture, Sandy Bridge Microarchitecture and Xeon Phi Processors.

- **IA32_MCi_STATUS:** Core Microarchitecture, Atom Processor Family, Silvermont Microarchitecture, Nehalem Microarchitecture, Sandy Bridge Microarchitecture, Xeon Phi Processors, Pentium 4 and Intel Xeon Processors, Core Solo, Intel Core Duo Processors, and Dual-Core Intel Xeon Processor LV and Pentium M Processors.

IA32_SMBASE is the 32-bit register which contains the Base address of the logical processors SMRAM image. To check if a processor supports this register, we need to verify the bit15 in the IA32_VMX_MISC register is set. If it is set, so the IA32_SMBASE is supported. It is accessible just when the processor is in SMM. Any attempt to read or write those registers will trigger a general-protection fault (#GP(0)).

Some new processors support the System-Management Range Register (SMRR) Interface. It allows restricting access to the memory address range in the SMRAM, which is used by the SMI handler code and data. Also, cacheable addresses references to SMI handler will be limited. SMRR interface comprise of two Model Specific Registers (MSR): The **IA32_SMRR_PHYSBASE** (64-bit) defines the base address for the SMRAM and the memory type used to access it; and the **IA32_SMRR_PHYSMASK** (64-bit) determines the SMRAM address range protected. They just can be written when in SMM and an attempt to write to them outside of SMM will cause a general-protection exception [87, 97].

To check if a processor supports the SMRR interface, it is necessary to verify the Memory Type Range Register (MTRR) **IA32_MTRRCAP**, specifically if its bit 11 is set. MTTRs are Model Specific Registers (MSR), which provide a mechanism for associating the memory types with physical-address ranges in system memory. Those registers are architecturally dependent on the microarchitecture in use, being more complex in multi-core processors, where might exist one or more of the same registers to be used by core. It can be generally referred as **scope**. For example, in processors based on the Silvermont microarchitecture, the multi-core processor (physical package) scope can be: **Package**, all cores share the same MSR or bit interface; **Shared**, a MSR or bit field is shared by one or more cores; **Core**, each core has a separate MSR or a bit field not shared with another core [87, 97].

The follow memory types can be encoded in MTRR registers, as **IA32 _ MTRR_PHYSBASEi** registers (with $i = 1, 2, ...n$) (section 11.11.2.3 in [97]), **IA32_VMX_BASIC**, **IA32_ SMRR_PHYSBASE** and other: Uncacheable (UC), Write Combining (WC), Write-through (WT), Write-protected (WP) and Writeback (WB) [86, 87, 97, 96].

IA32_VMX_BASIC is a 64-bit Model-Specific Register (MSR) used to report basic VMX capabilities (section 3.3.4). It should be consulted to check if the SMM Dual-Monitor Treatment is supported by a processor or not. If bit 49 is set, it means that the processor supports the dual-monitor treatment of system-management interrupts and SMM. The Dual-Monitor Treatment is discussed in section 3.3.4 [87, 97].

IA32_VMX_MISC is a 64-bit MSR used to report miscellaneous VMX capabilities. This work is concerned with the SMM related functions. So, the bit15, when set, allow the RDMSR instruction to read the **IA32_SMBASE** register when in SMM. When bit28 is set, it means that bit2 of the **IA32_SMM_MONITOR_CTL** can be set (to 1). This affect the VMXOFF instruction, which can be executed only with the

default treatment (section 3.3.4) and only outside SMM. VMXOFF can unblock SMI if bit2 of **IA32_SMM_MONITOR_CTL** is cleared. Finally, bit63 to bit32 contains the 32-bit MSEG revision identifier used by the processor [87, 97].

IA32_DEBUGCTL is the 32-bit Debug Control register. It controls many features related to debug tasks. What concern to the SMM is the bit14 (FREEZE_WHILE_SMM _EN). If this bit is set, the processor will freeze performance counters during all time the processor is in SMM [87, 97].

IA32_PERF_CAPABILITIES is the 64-bit performance capabilities register and indicates in bit12 (SMM_FREEZE) that the processor supports FREEZE_WHILE_SMM_EN (SMM_FREEZE = 1) or not (SMM_ FREEZE = 0) [87, 97].

Processors have a set of 64-bit MSR registers denominated **IA32_MCi_STATUS**, with $i = 1, 2, ...n$, containing information related to machine-check error. The relevant bits to SMM are, a priori: bit63 (VAL flag), which is set when the processor find an error after a machine-check and indicates that the information in that register is valid; and the MCA (machine-check architecture) error code field (bits from 0 to 15) (section 15.3.2.2 in [87, 97]). When MCA code is equals to 0006H, it configures an "SMM Handler Code Access Violation", which means that "an attempt was made by the SMM Handler to execute outside the ranges specified by SMRR" (section 15.9 and 16 in [87, 97]).

IA32_SMM_MONITOR_CTL is the 64-bit SMM Monitor Configuration. It can be read any time by a RDMSR instruction, but just can be written when in SMM. Bit0 is the valid bit. When the valid bit is set the STM may be invoked using VMCALL. Then, the dual-monitor treatment is activated only if the valid bit is set. As discussed before, it controls SMI unblocking by VMXOFF in its bit2. Its bit31 to bit12 contains the MSEG base. The other bits are reserved [87, 97]. We cite this register here for illustration purposes since it is related to SMM, but it is not supported by our chipset 1, just in Pentium 4 and Intel Xeon Processors.

A.1.2 Chipset 1 SMM Model-Specific Registers

This section describes the Model-Specific Registers related to the SMM, whose are not categorized as architectural MSR. Below is listed those MSR and the family/model of processor which supports them. As we can see, our chipset 1 is not supported. This section is based on [87, 97].

- **MSR_SMM_FEATURE_CONTROL:** Haswell microarchitecture.
- **MSR_SMM_MCA_CAP:** Haswell microarchitecture, Xeon Processor E5 v3 Family, Xeon Processor D and Broadwell Microarchitecture.
- **MSR_SMM_DELAYED:** Haswell microarchitecture.
- **MSR_SMM_BLOCKED:** Haswell microarchitecture.

Those registers are accessible just when the processor is in SMM. Any attempt to read or write those registers will trigger a general-protection fault (#GP(0)).

The 64-bit **MSR_SMM_FEATURE_CONTROL** is the Enhanced SMM Feature Control register and restricts SMI handler address range. The 64-bit **MSR_SMM**

_MCA_CAP** it the Enhanced SMM Capabilities register and offers additional write protection to that latter register.

MSR_SMM_DELAYED is the 64-bit SMM Delayed register and reports the interruptible state of all logical processors in the physical package (and package, as described before, comprises of the processor and its cores and other internal components).

MSR_SMM_BLOCKED is the 64-bit SMM Blocked registers and reports the blocked state of all logical processors in the physical package. Those logical processor reported in the MSR_SMM_BLOCKED are the ones blocked from servicing interrupts, including SMI [87, 97].

A.1.3 Other Registers Related to SMM in Chipset 1

There are plenty of registers related to SMM in chipset 1. It is neither possible nor convenient to address all of them in this research. So, we list them below with a short description, their size and the bits in those registers related to SMM to map them for future works. Some bits related to SMM in those registers are reported in the format "number to number", as in the TRSR register, whose bits are reported as 3 to 0. It means that a specific function related to SMM in this register requires all those bits together to be defined [81].

- **Register:** TRSR. **Description:** Trap Status Register. **Size:** 32-bit. **Bits related:** 3 to 0.
- **Register:** TRCR. **Description:** Trapped Cycle Register. **Size:** 64-bit. **Bits related:** 3 to 0.
- **Register:** IOTRi. **Description:** I/O Trap Register (register from 0 to 3). **Size:** 64-bit. **Bit related:** 0.
- **Register:** GCS. **Description:** General Control and Status Register. **Size:** 32-bit. **Bit related:** 5.
- **Register:** PCICMD. **Description:** PCI Command Register. **Size:** 16-bit. **Bit related:** 8.
- **Register:** GC. **Description:** GPIO Control Register. **Size:** 8-bit. **Bit related:** 0.
- **Register:** ULKMC. **Description:** USB Legacy Keyboard / Mouse Control Register. **Size:** 32-bit. **Bits related:** 15, 11, 10, 9, 8, 7, 5, 4, 3, 2, 1 and 0.
- **Register:** BIOS_CNTL. **Description:** BIOS Control Register. **Size:** 8-bit. **Bits related:** 5, 1 and 0.
- **Register:** PM1_STS. **Description:** Power Management 1 Status Register. **Size:** 16-bit. **Bits related:** 8, 4 and 0.
- **Register:** PM1_EN. **Description:** Power Management 1 Enable Register. **Size:** 16-bit. **Bits related:** 10, 8 and 0.
- **Register:** PM1_CNT. **Description:** Power Management 1 Control Register. **Size:** 32-bit. **Bit related:** 0.

- **Register:** DEVACT_STS. **Description:** Device Activity Status Register. **Size:** 16-bit. **Bit related:** 12, 9, 8, 7 and 6.
- **Register:** TCO_DAT_IN. **Description:** TCO Data In Register. **Size:** 8-bit. **Bits related:** 7 to 0
- **Register:** TCO_DAT_OUT. **Description:** TCO Data Out Register. **Size:** 8-bit. **Bits related:** 7 to 0.
- **Register:** TCO1_STS. **Description:** TCO1 Status Register. **Size:** 16-bit. **Bits related:** 10, 8, 7, 3, 2, 1 and 0.
- **Register:** TCO2_STS. **Description:** TCO2 Status Register. **Size:** 16-bit. **Bits related:** 4 and 0.
- **Register:** TCO1_CNT. **Description:** TCO1 Control Register. **Size:** 16-bit. **Bits related:** 9 and 8.
- **Register:** TCO2_CN. **Description:** TCO2 Control Register. **Size:** 16-bit. **Bits related:** 2 to 1.
- **Register:** GPI_INV. **Description:** GPIO Signal Invert Register. **Size:** 32-bit. **Bits related:** 15 to 0.
- **Register:** ATC. **Description:** APM Trapping Control Register. **Size:** 8-bit. **Bits related:** 3, 2, 1 and 0.
- **Register:** LEG_EXT_CS. **Description:** USB EHCI Legacy Support Extended Control / Status Register. **Size:** 32-bit. **Bits related:** 31, 30, 29, 21, 20, 19, 18, 17, 16, 15, 14, 13, 5, 4, 3, 2, 1 and 0.
- **Register:** SPECIAL_SMI. **Description:** Intel Specific USB 2.0 SMI Register. **Size:** 32-bit. **Bits related:** 24 to 22, 21, 20, 19, 18, 17, 16, 13 to 6, 5, 4, 3, 2, 1 and 0.
- **Register:** HOSTC. **Description:** Host Configuration Register. **Size:** 8-bit. **Bits related:** 1 and 0.
- **Register:** HST_STS. **Description:** Host Status Register. **Size:** 8-bit. **Bits related:** 5, 4, 3, 2 and 1.
- **Register:** HST_CNT. **Description:** Host Control Register. **Size:** 8-bit. **Bits related:** 4 to 2, 1 and 0.
- **Register:** HOST_BLOCK_DB. **Description:** Host Block Data Byte Register. **Size:** 8-bit. **Bits related:** 7 to 0.
- **Register:** SLV_STS. **Description:** Slave Status Register. **Size:** 8-bit. **Bit related:** 0.
- **Register:** SLV_CMD. **Description:** Slave Command Register. **Size:** 8-bit. **Bits related:** 2 and 0.

- **Register:** MPC. **Description:** Miscellaneous Port Configuration Register. **Size:** 32-bit. **Bits related:** 1 and 0.
- **Register:** SMSCS. **Description:** SMI/SCI Status Register. **Size:** 32-bit. **Bits related:** 4, 1 and 0.
- **Register:** HSFS. **Description:** Hardware Sequencing Flash Status Register (SPI Memory Mapped Configuration Registers). **Size:** 16-bit. **Bit related:** 0.
- **Register:** HSFC. **Description:** Hardware Sequencing Flash Control Register (SPI Memory Mapped Configuration Registers). **Size:** 16-bit. **Bit related:** 15.
- **Register:** SSFS. **Description:** Software Sequencing Flash Status Register (SPI Memory Mapped Configuration Registers). **Size:** 8-bit. **Bit related:** 2.
- **Register:** SSFC. **Description:** Software Sequencing Flash Control Register (SPI Memory Mapped Configuration Registers). **Size:** 24-bit. **Bit related:** 15.
- **Register:** HSFS. **Description:** Hardware Sequencing Flash Status Register (GbE LAN Memory Mapped Configuration Registers). **Size:** 16-bit. **Bit related:** 0.
- **Register:** SSFS. **Description:** Software Sequencing Flash Status Register (GbE LAN Memory Mapped Configuration Registers). **Size:** 8-bit. **Bit related:** 2.
- **Register:** TSPC. **Description:** Thermal Sensor Policy Control Register. **Size:** 8-bit. **Bits related:** 3, 2, 1 and 0.
- **Register:** HIDM. **Description:** MEI Interrupt Delivery Mode Register. **Size:** 8-bit. **Bits related:** 1 and 0.
- **Register:** HIDM. **Description:** Intel MEI Interrupt Delivery Mode Register. **Size:** 8-bit. **Bits related:** 1 and 0.

A.2 Chipset 2 System Management RAM Control register

Like in chipset 1, this is the 8-bit System Management RAM Control register (SMRAMC), the access control mechanism to the SMRAM (figure 3.4). In our target chipset 2, it is located at PCI device 0, address offset 90H [65]. According to [70], the SMRAMC in chipset 2 does not differ from the SMRAMC in chipset 1, as described in section 3.2.3.2. However, the default value of this register is 02H, which means that SMRAM is open after the bootup process. This chipset has an extra control register, the Extended System Management RAM Control Register (ESMRAMC), that is described in section A.2.1. In chipset 2, this register also controls access to the E_SMRAM register (section A.2.1)

A.2.1 Chipset 2 Extended System Management RAM Control register

This register appears in older chipsets, as 830 [65] and 845 [69, 70] chipset families. The 8-bit Extended System Management RAM Control (ESMRAMC) register controls the configuration of Extended SMRAM (E_SMRAM) space. The E_SMRAM memory provides a write-back cacheable SMRAM memory space that is above 1 MB. it is located at PCI device 0, address offset 91H and the default value of this register is 02H [65].

Bit7 (H_SMRAME) controls the SMM memory space location, so that when this bit is set and the G_SMRAME bit (bit3 in SMRAMC register) is also set, the high SMRAM memory space is enabled. In this case, the SMRAM accesses from FEDA0000H to FEDBFFFFH are remapped to SDRAM address A0000H to BFFFFH. E_SMERR bit (Bit6) is set when CPU accesses the Extended SMRAM while not in SMM and with the D_OPEN bit cleared. Bit5 (SM_CACHE), bit4 (SM_L1) and bit3 (SM_L2) are set by the chipset (hardwired). The bit2 is reserved. bit1 (T_SZ) indicates the size of the TSEG memory block if TSEG is enabled. When T_SZ is cleared (TOM-512K) to TOM 1 (TOM-1M) to TOM. When bit0 (T_EN) is set and bit G_SMRAME (bit3 in SMRAMC register) is also set, it enables TSEG is enabled to appear in the appropriate physical address space.

A.2.2 Chipset 2 Specific Registers

The chipset 2 specific registers related to SMM are: PMBASE, SMI_EN, SMI_STS, GEN_PMCON_1, GPI_ROUT, GPE0_EN, GPE0_STS, APM_CNT and APM_STS [64].

PMBASE is the 32-bit Advanced Configuration and Power Interface (ACPI) base address register. It can be accessed at device 31, function 0, offset 40H-43H and sets the base address for ACPI I/O registers and other ones, including the **SMI_EN** and **SMI_STS**. Those registers can be accessed by PMBASE + offset; for instance, the SMI_EN at PMBASE + 30H and the SMI_STS at PMBASE + 34H [64].

SMI_EN is the 32-bit SMI Control and Enable register. This register is symmetrical to the SMI_STS, so they work together to generate and provide information about SMI. The bits of this register enable which devices can trigger an SMI and other SMI related functions. Below, we describe the bits of interest for this research [64]:

- **Bit PERIODIC_EN [bit 14].** Set this bit causes an SMI when the PERIODIC_STS bit at SMI_STS register is set too.
- **Bit TCO_EN [bit 13].** Set this bit enables the Total Cost of Ownership (TCO) logic to generate SMI. There are a huge set of register related to TCO.
- **Bit MCSMI_EN [bit 11].** Set this bit enables PCH to trap accesses to the microcontroller range (62H or 66H) and then generate an SMI.
- **Bit SWSMI_TMR_EN [bit 6].** Set this bit starts Software SMI Timer, such that when the SWSMI timer expires, the SWSMI_TMR_STS bit at SMI_STS is set and an SMI is generated.
- **Bit APMC_EN [bit 5].** When this bit is set any write to the APM_CNT register will cause an SMI.

- **Bit SLP_SMI_EN [bit 4].** When this bit is set, this enable that set the SLP_EN bit (bit 13 in PM1_CNT register) cause an SMI.

- **Bit LEGACY_USB_EN [bit 3].** Set this bit enables legacy USB circuit to cause an SMI.

- **Bit BIOS_EN [bit 2].** Set this bit enables the generation of SMI when ACPI software writes a 1 to the GBL_RLS bit (bit 2 in PM1_CNT register).

- **Bit EOS [bit 1].** Set this bit cause an SMI signal to be deasserted for 4 PCI clocks before its assertion.

- **Bit GBL_SMI_EN [bit 0].** Set this bit enable SMI globally in the system.

SMI_STS is the 32-bit SMI Status register and indicates the device that have caused an SMI. As a general rule, when a bit is set in this register, whenever the correspondent bit is set in the SMI_EN register an SMI is generated, since those registers are symmetrical. However, in some chipsets not all bits in SMI_EN have correspondent in SMI_STS register. Below, we describe the bits of interest for this research [64]:

- **Bit SMBUS_SMI_STS [bit 16].** If this bit is set, an SMI was caused by a SMBUS related event.

- **Bit SERIRQ_SMI_STS [bit 15].** This bit is set when an SMI was caused by the Serial Interrupt Request (SERIRQ) decoder.

- **Bit PERIODIC_STS [bit 14].** This bit is set at the rate indicated by the PER_SMI_SEL (bits 1 and 0 in GEN_PMCON_1 register). Then, if the PERIODIC_EN bit (bit 14 in SMI_EN register) is also set, an SMI is generated.

- **Bit TCO_STS [bit 13].** If this bit is set, an SMI was caused by the Total Cost of Ownership (TCO) logic.

- **Bit DEVMON_STS [bit 12].** This bit is set when an SMI was caused by a Device Monitor.

- **Bit MCSMI_STS [bit 11].** If this bit is set, there was an access to the power management microcontroller range (62H or 66H).

- **Bit GPE1_STS [bit 10].** This bit is set when an SMI was generated by a General Purpose Input/Output (GPI) assertion.

- **Bit GPE0_STS [bit 9].** This bit is set when an SMI was generated by an SMI was generated by a General Purpose Event (GPE0) event.

- **Bit PM1_STS_REG [bit 8].** This bit is set when an SMI was generated by a PM1_STS event.

- **Bit SWSMI_TMR_STS [bit 6].** This bit is set by hardware whenever the Software SMI Timer expires.

- **Bit APM_STS [bit 5].** If this bit is set, this indicates that an SMI was generated due to a write access to the APM_CNT register when the APMC_EN bit (bit 5 in SMI_EN register) was set.
- **Bit SLP_SMI_STS [bit 4].** When this bit is set, this indicates an SMI was caused due to set the SLP_EN bit (bit 13 in PM1_CNT register) when SLP_SMI_EN bit (bit 5 in SMI_EN) is also set.
- **Bit LEGACY_USB_STS [bit 3].** If this bit is set, an SMI was caused by a USB Legacy event.
- **Bit BIOS_STS [bit 2].** This bit is set by hardware when the GBL_RLS bit (bit 2 in PM1_CNT register) is set. In this case, if the BIOS_EN bit (bit 2 in SMI_EN register) is also set, an SMI is generated.

GEN_PMCON_1 is the 16-bit General Power Management Configuration 1 register. It can be accessed at device 31, function 0, offset A0H. In this chipset, GEN_PMCON_1 register has two main functions related to SMI [64], one different and another similar to those one in chipset 1 [81]. First, the SWSMI_RATE_SEL bit (bit10) sets up the SWSMI Timer timeout to 64 ms $\pm$ 4 ms, which is the default value when it is cleared. When that bit is set, it sets up the SWSMI Timer timeout to 1.5 ms $\pm$ 0.5 ms. That bit is related to the SWSMI_TMR_EN bit (bit6 in SMI_EN register) and to the SWSMI_TMR_STS bit (bit6 in SMI_STS register). Second, the PER_SMI_SEL bits (bit1 and bit0) controls the rate at which periodic SMI is generated, according to the following scheme: bit1 = 0 and bit0 = 0 equals 1 minute, bit1 = 0 and bit0 = 1 equals 32 seconds, bit1 = 1 and bit0 = 0 equals 16 seconds and bit1 = 1 and bit0 = 1 equals 8 seconds. These bits are related to the PERIODIC_EN bit (bit6 in SMI_EN register) and to the PERIODIC_STS bit (bit6 in SMI_STS register).

GPE0_EN is the 64-bit General Purpose Event 0 Enables register and **GPE0_STS** is the 64-bit General Purpose Event 0 Status register. Those registers are symmetrical to each other and they are used to manage wake events in the system. For some functions, an SMI will be generated. For example, when the BATLOW_EN bit (bit10 in GPI0_EN register) is set, it enables the BATLOW# signal to cause an SMI when the battery goes low. In this case, the hardware set the BATLOW_STS bit (bit10 in GPI0_STS register) as soon as the BATLOW# signal is asserted [64].

APM_CNT is the 8-bit Advanced Power Management Control Port register and it is used to pass an APM command between the OS and the SMI handler. Writes to this port not only store data in the APMC register, but also generates an SMI when the **APMC_EN** bit (bit5) in the SMI_EN register is set. **APM_STS** it the 8-bit Advanced Power Management Status Port register and it is used to pass data between the OS and the SMI handler [80, 81].

A.2.3 Chipset 2 Architectural Model-Specific Registers

In our target chipset 2 there is no Architectural Model-Specific Registers (MSR) related to SMM. The follow registers are present in chipset 1, but not in chipset 2: IA32_SMRR_PHYSBASE, IA32_SMRR_PHYSMASK, IA32_VMX_BASIC, IA32_VMX_MISC and IA32_PERF_CAPABILITIES. The IA32_MTRRCAP, IA32_MCi_STATUS and IA32_ DEBUGCTL registers are present in chipset 2, but they have no function

related to SMM in chipset 2. It is because, in newer chipset, those registers embody modern features as the VMX technology (section 3.3.4 and SMRR interface (section A.1.1 and sections 11.11.2.4 and 34.4.2.1 in [87, 97]), which were not developed when chipset 2 was released.

A.2.4 Chipset 2 SMM Model-Specific Registers

This section describes the Model-Specific Registers related to the SMM, whose are not categorized as architectural MSR. Below is listed those MSR and the family/ model which supports them. As we can see, our chipset 1 is not supported. This section is based on [87, 97].

The follow registers are present in chipset 1, but not in chipset 2: MSR_SMM_FEATURE_CONTROL, MSR_SMM_MCA_CAP, MSR_SMM_ DELAYED and MSR _SMM_BLOCKED.

A.2.5 Other Registers Related to SMM in Chipset 2

As in chipset 1, in chipset 2 there are many registers related to SMM too. So, we list them below with a short description, their size and the bits in those registers related to SMM to map them for future works. Again, some bits related to SMM in those registers are reported in the format "number to number", as in the GPE1_STS register, whose bits are reported as 15 to 0. It means that a specific function related to SMM in this register requires all those bits together to be defined [64, 65].

- **Register:** PCISTS. **Description:** PCI Status Register. **Size:** 16-bit. **Bit related:** 14.
- **Register:** RRBAR. **Description:** Register Range Base Address Register. **Size:** 32-bit. **Bit related:** 31 to 18.
- **Register:** CMD. **Description:** Command Register. **Size:** 16-bit. **Bit related:** 8.
- **Register:** PD_STS. **Description:** Primary Device Status Register. **Size:** 16-bit. **Bits related:** 14 and 12.
- **Register:** BRIDGE_CNT. **Description:** Bridge Control Register. **Size:** 16-bit. **Bit related:** 1.
- **Register:** ERR_CMD. **Description:** Error Command Register. **Size:** 8-bit. **Bits related:** 2 and 1.
- **Register:** PCISTA. **Description:** PCI Device Status Register. **Size:** 16-bit. **Bit related:** 14.
- **Register:** BIOS_CNTL. **Description:** BIOS Control Register. **Size:** 16-bit. **Bits related:** 1 and 0.
- **Register:** TRP_FWD_EN. **Description:** I/O Monitor Trap Forwarding Enable Register. **Size:** 8-bit. **Bits related:** 7, 6, 5, and 4.
- **Register:** MON[n]_TRP_RNG. **Description:** I/O Monitor [4:7] Trap Range Register for Devices 47. **Size:** 16-bit. **Bits related:** 15 to 0.

- **Register:** PM1_STS. **Description:** Power Management 1 Status Register. **Size:** 16-bit. **Bits related:** 8 and 0.
- **Register:** PM1_EN. **Description:** Power Management 1 Enable Register. **Size:** 16-bit. **Bits related:** 10, 8 and 0.
- **Register:** PM1_CNT. **Description:** Power Management 1 Control Register. **Size:** 32-bit. **Bit related:** 0.
- **Register:** PROC_CNT. **Description:** Processor Control Register. **Size:** 32-bit. **Bit related:** 8.
- **Register:** GPE1_STS. **Description:** General Purpose Event 1 Status Register. **Size:** 16-bit. **Bits related:** 15 to 0.
- **Register:** GPE1_EN. **Description:** General Purpose Event 1 Enable Register. **Size:** 16-bit. **Bits related:** 15 to 0.
- **Register:** MON_SMI. **Description:** Device Monitor SMI Status and Enable Register. **Size:** 16-bit. **Bits related:** 15 to 12 and 11 to 8.
- **Register:** DEVACT_STS. **Description:** Device Activity Status Register. **Size:** 16-bit. **Bits related:** 13 to 5 and 3 to 0.
- **Register:** DEVTRAP_EN. **Description:** Device Trap Enable Register. **Size:** 16-bit.**Bits related:** 13 to 10, 5 and 3 to 0.
- **Register:** BUS_ADDR_TRACK. **Description:** Bus Address Tracker Register. **Size:** 16-bit. **Bits related:** 15 to 0.
- **Register:** BUS_CYC_TRACK. **Description:** Bus Cycle Tracker Register. **Size:** 8-bit. **Bits related:** 7 to 4 and 3 to 0.
- **Register:** TCO1_DAT_IN. **Description:** TCO Data In Register. **Size:** 8-bit. **Bits related:** 7 to 0.
- **Register:** TCO1_DAT_OUT. **Description:** TCO Data Out Register. **Size:** 8-bit. **Bits related:** 7 to 0.
- **Register:** TCO1_STS. **Description:** TCO1 Status Register. **Size:** 16-bit. **Bits related:** 12, 10, 8, 7, 3, 2, 1 and 0.
- **Register:** TCO2_STS. **Description:** TCO2 Status Register. **Size:** 16-bit. **Bit related:** 4.
- **Register:** TCO1_CNT. **Description:** TCO1 Control Register. **Size:** 16-bit. **Bits related:** 11, 9 and 8.
- **Register:** TCO2_CNT. **Description:** TCO2 Control Register. **Size:** 16-bit. **Bits related:** 2 to 1.
- **Register:** GPI_INV. **Description:** GPIO Signal Invert Register. **Size:** 32-bit. **Bits related:** 13 to 11, 8, 7 to 0.

- **Register:** USB_LEGKEY. **Description:** USB Legacy Keyboard/Mouse Control Register. **Size:** 16-bit. **Bits related:** 15, 13, 12, 11, 10, 9, 8, 7, 5, 4, 3, 2, 1 and 0.
- **Register:** HOSTC. **Description:** Host Configuration Register. **Size:** 8-bit. **Bits related:** 1 and 0.
- **Register:** HST_STS. **Description:** Host Status Register. **Size:** 8-bit. **Bits related:** 5, 4, 3, 2 and 1.
- **Register:** HST_CNT. **Description:** Host Control Register. **Size:** 8-bit. **Bits related:** 4 to 2, 1 and 0.
- **Register:** BLOCK_DB. **Description:** Block Data Byte Register. **Size:** 8-bit. **Bits related:** 7 to 0.
- **Register:** SLV_STS. **Description:** Slave Status Register. **Size:** 8-bit. **Bit related:** 0.
- **Register:** SLV_CMD. **Description:** Slave Command Register. **Size:** 8-bit. **Bits related:** 2 and 0.
- **Register:** ERRSTS. **Description:** Error Status Register. **Size:** 16-bit. **Bit related:** 12.
- **Register:** ERRCMD. **Description:** Error Command Register. **Size:** 16-bit. **Bit related:** 12.

Bibliography

[1] Aws case study: 6 waves limited. Site, 2011. Accessed in 17/04/2016. 21

[2] *Unified Extensible Firmware Interface Specification*, 2.4, errata b ed., April 2014. Accessed on 14/05/2014. 8, 36

[3] Unified extensible firmware interface (uefi) forum. Site, March 2014. Accessed in 13/03/2014. 36

[4] *Implementation of SHA-256 in C*, 2015. 84, 131

[5] Cloud standards. Site, February 2016. Accessed in 18/04/2016. 22

[6] Platform-as-a-service: Develop and deliver apps faster with paas. Site, 2016. Accessed in 17/04/2016. 21

[7] Security-as-a-service: Mcafee saas web protection. Site, 2016. Accessed in 17/04/2016. 20

[8] A. Whitaker, M. S., and Gribble, S. D. Scale and performance in the denali isolation kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, USA, December 2002), vol. Winter 2002 Special Issue of *ACM Operating Systems Review*, ACM, p. 195210. 30

[9] Acer. *Acer Aspire X1935 Service Guide*. Acer Corporation, 2012. 4

[10] Anati, I., Gueron, S., Johnson, S. P., and Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, 2013), HASP '13, ACM, pp. 10:1–10:1. doi:10.1145/2487726.2488368. 2, 58

[11] Anderson, D., and Shanley, T. *Pentium processor system architecture*, 2nd ed. MindShare, Reading, MA, 1995. 55

[12] Anderson, D., and Shanley, T. *PCI system architecture*, 4th ed. MindShare, Reading, MA, 1999. 63

[13] Azab, A. M., Ning, P., Sezer, E. C., and Zhang, X. Hima: A hypervisor-based integrity measurement agent. In *Proceedings of the 2009 Annual Computer Security Applications Conference* (Washington, DC, USA, 2009), ACSAC '09, IEEE Computer Society, pp. 461–470. doi:10.1109/ACSAC.2009.50. 41

[14] AZAB, A. M., NING, P., WANG, Z., JIANG, X., ZHANG, X., AND SKALSKY, N. C. Hypersentry: Enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 38–49. doi:10.1145/1866307.1866313. 2, 6, 7, 9, 10, 59, 60, 61, 70, 73, 74

[15] AZAB, A. M., NING, P., AND ZHANG, X. Sice: A hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 375–388. doi:10.1145/2046707.2046752. 2, 7, 60

[16] BACLIT, R., SICAM, C., MEMBREY, P., AND NEWBIGIN, J. *Foundations of CentOS Linux: Enterprise Linux On the Cheap*. The experts's voice in Linux. Apress, New York, NY, USA, 2009. 28

[17] BAIARDI, F., AND SGANDURRA, D. Building trustworthy intrusion detection through vm in-strospection. In *In Proceedings of the Third International Symposion on Information Assurance and Security* (August 2007), vol. IAS 2007, IEEE, pp. 209 – 214. 42

[18] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 164–177. doi:10.1145/945445.945462. 17, 23, 30, 31, 57, 80

[19] BARRETO, S. L. M. P., AND RIJMEN, V. *The Whirlpool Hash Function*, May 2003. 39

[20] BELL, D. E., AND LAPADULA, L. J. Secure computer systems: Mathematical foundations. Tech. Rep. 2547, MITRE, March 1973. 37

[21] BERGER, S., CÁCERES, R., GOLDMAN, K. A., PEREZ, R., SAILER, R., AND VAN DOORN, L. vtpm: Virtualizing the trusted platform module. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15* (Berkeley, CA, USA, 2006), USENIX-SS'06, USENIX Association. 41

[22] BING, S. Bios boot hijacking by using intel ichx "top-block swap" mode. In *Proceedings of XFocus Information Security Conference* (2007), XFocus. 65

[23] BRANCO, R. R. System management mode hack: Using smm for "other purposes", November 2008. 2, 7, 12, 48, 63, 65, 74, 106, 131

[24] BREY, B. *The Intel microprocessors 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, Pentium Pro processor, Pentium II, Pentium III, Pentium 4, and Core2 with 64-bit extensions: architecture, programming, and interfacing*, 8th ed. Pearson, Upper Saddle River, OH, 2008. 35, 48, 55

[25] BUTTERWORTH, J., KALLENBERG, C., KOVAH, X., AND HERZOG, A. Bios chronomancy: Fixing the core root of trust for measurement. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 25–36. doi:10.1145/2508859.2516714. 7, 60

[26] C. LI, A. R., AND JHA, N. K. Secure virtual machine execution under an untrusted management os. In *In Proceedings of the Conference on Cloud Computing (CLOUD)* (July 2010), IEEE, pp. 172 – 179. doi:10.1109/CLOUD.2010.29. 42

[27] CHEN, P. M., AND NOBLE, B. D. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems* (Washington, DC, USA, 2001), HOTOS '01, IEEE Computer Society, pp. 133–. 41

[28] CHISNAL, D. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall, Upper Saddle River, NJ, USA, 2008. 17, 30, 31

[29] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2001), SOSP '01, ACM, pp. 73–88. doi:10.1145/502034.502042. 42

[30] CHRISTODORESCU, M., SAILER, R., SCHALES, D. L., SGANDURRA, D., AND ZAMBONI, D. Cloud security is not (just) virtualization security: a short paper. In *ACM workshop on Cloud computing security* (2009), ACM, pp. 97–102. 19, 22

[31] CITRIX. Cve-2007-5497: Vulneravility in xenserver could result in privilege escalation and arbitrary code execution. Website, November 2008. Accessed in 27/01/2017. 71

[32] COMPAQ INFORMATION TECHNOLOGIES GROUP. *Hardware Guide: Compaq Evo Notebook N410c Series*, document part number: 274039-001 ed., July 2002. 4

[33] COMPAQ, MICROSOFT, NATIONAL SEMICONDUCTOR. *OpenHCI - Open Host Controller Interface Specification for USB*, revision 1.0a ed., September 1996. 64

[34] COPELAND, M., SOH, J., PUCA, A., MANNING, M., AND GOLLOB, D. *Microsoft Azure: Planning, Deploying, and Managing Your Data Center in the Cloud*. Apress, New York, NY, USA, 2015. 21

[35] CORBATO, F. J., DAGGETT, M. M., AND DALEY, R. C. An experimental timesharing system. In *Spring Joint Computer Conference* (San Francisco, California, May 1962), AIEE-IRE '62, pp. 335–344. 25

[36] COREBOOT. *Coreboot: fast and flexible Open Source firmware*, 2014. 4, 7, 36, 61, 65, 83

[37] COREBOOT. *Inteltool*, November 2014. 106

[38] DE SOUZA, W. A. R., AND TOMLINSON, A. Understanding threats in a cloud infrastructure with no hypervisor. In *Proceedings of the 2013 World Congress on Internet Security (WorldCIS),* (London, UK, December 2013), IEEE, pp. 128 – 133. doi:10.1109/WorldCIS.2013.6751032. 19, 42, 43

[39] DE SOUZA, W. A. R., AND TOMLINSON, A. Virtualisation without a hypervisor in cloud infra-structures: An initial analysis. In *Proceedings of the 14th Annual Postgraduate Sympo-sium on the Convergence of Telecommunications, Networking & Broadcasting* (Liverpool, UK, June 2013), Liverpool John Moores University, Liverpool John Moores University. 42

[40] DE SOUZA, W. A. R., AND TOMLINSON, A. Smm-based hypervisor integrity measurement. In *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing* (New York, NY, USA, November 2015), CSCloud, IEEE, pp. 362 – 367. doi:10.1109/CSCloud.2015.57. 22

[41] DELGADO, B., AND KARAVANIC. Performance implications of system management mode. In *Proceedings of the IEEE International Symposium on Workload Characteriztion* (September 2013), IISWC, pp. 163–173. 52

[42] DEPARTMENT OF DEFENSE. *Trusted Computer System Evaluation Criteria (Orange Book)*, volume dod 5200.28-std ed., December 1985. 33, 37, 38

[43] DOMAS, C. The memory sinkhole. In *Black Hat Conference* (Las Vegas, NV, August 2015). 76

[44] DORAN, M., ZIMMER, V. J., AND ROTHMAN, M. A. Beyond bios: exploring the many dimensions of the unified extensible firmware interface. *In Intel Technology Journal - UEFI Today: Bootstrapping the Continuum 15*, 1 (October 2011), 8–21. ISBN 978-1-934053-43-0, ISSN 1535-864X. 36

[45] DUFLOT, L., ETIEMBLE, D., AND GRUMELARD. Using cpu system management mode to circumvent operating system security functions. In *Proceedings of 7th CanSecWest Security Conference* (2006). 2, 7, 16, 29, 41, 48, 50, 51, 63, 74, 131

[46] DUFLOT, L., LEVILLAIN, O., MORIN, B., AND GRUMELARD, O. Getting into the smram: Smm reloaded. In *Proceedings of 12th CanSecWest Security Conference* (2009). 2, 29, 63, 64, 70, 74

[47] EMBLETON, S., SPARKS, S., AND ZOU, C. Smm rootkits: A new breed of os independent malware. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks* (New York, NY, USA, 2008), SecureComm '08, ACM, pp. 11:1–11:12. doi:10.1145/1460877.1460892. 2, 12, 41, 48, 63, 64, 65, 95

[48] ERL, T., MAHMOOD, Z., AND PUTTINI, R. *Cloud computing: concepts, technology and architecture*. Prentice Hall, Upper Saddle River, NJ, 2013. 19

[49] (FIPS), F. I. P. S. *Secure Hash Standard*, publication 180 ed. National Institute of Standards and Technology, USA, 1993. 39

[50] (FIPS), F. I. P. S. *Secure Hash Standard*, publication 180-2 ed. National Institute of Standards and Technology, USA, 2002. 39, 84, 93

[51] GALUS, D. Migration to new display technologies on intel embedded platforms. Tech. rep., Intel Corporation, 2012. 33

[52] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 193–206. doi:10.1145/945445.945464. 41

[53] GARFINKEL, T., AND ROSENBLUM. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed System Security Symposium* (San Diego, CA, February 2003), The Internet Society, pp. 191–206. 42, 60

[54] GEBHARDT, C. *Towards Trustworthy Virtualisation: Improving the Trusted Virtual Infrastructure*. Phd thesis, Royal Holloway, University of London, Egham Hill, Egham, UK, October 2010. vii, 26, 28, 38, 39

[55] GENDRON, M. S. *Business intelligence and the cloud: strategic implementation guide*. Wiley, Hoboken, NJ, USA, 2014. 20, 21

[56] GOLDBERG, R. P. A survey of virtual machine research. In *Computer* (June 1974), vol. 7, Honeywell Information Systems and Harvard University, IEEE Computer Society, pp. 34–43. 26, 29

[57] GRAWROCK, D. *Dynamics of a trusted platform: a building block approach*, 2nd ed. Intel Press, Hillsboro, OR, 2008. vii, 2, 10, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 46, 57, 59, 65, 73, 76, 105

[58] GREENE, J. Intel trusted execution technology: Hardware-based technology for enhancing server platform security. Tech. rep., Intel Corporation, 2012. 2, 10, 36, 37

[59] HAGEN, W. V. *Professional Xen Virtualization*. Wiley, Indianapolis, IN, USA, 2008. 17, 27, 31

[60] HOEKSTRA, M., LAL, R., PAPPACHAN, P., PHEGADE, V., AND DEL CUVILLO, J. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, 2013), HASP '13, ACM, pp. 11:1–11:1. doi:10.1145/2487726.2488370. 2, 58

[61] HOULIHAN, R., AND DU, X. An effective auditing scheme for cloud computing. In *Proceeginds of Global Communications Conference* (2012), GLOBECOM, IEEE, pp. 1599 – 1604. doi:10.1109/GLOCOM.2012.6503342. 6, 7, 9, 60, 61, 70, 73, 74

[62] HUGOS, M. H., AND HULITZKY, D. *Business in the Cloud: What Every Business Needs to Know about Cloud Computing*. Wiley, Hoboken, NJ, USA, 2011. 20

[63] IBRAHIM, A., HARRIS, J., AND GRUNDY, J. Emerging security challenges of cloud virtual infrastructure. In *Cloud Workshop* (Sidney, Australia, 2010), APSEC 2010. 19, 22

[64] INTEL. *Intel 82801CAM I/O Controller Hub 3 (ICH3-M)*, datasheet ed. Intel Corporation, July 2001. 2, 51, 53, 108, 149, 150, 151, 152

[65] INTEL. *Intel 830 Chipset Family: 82830 Graphics and Memory Controller Hub (GMCH-M)*, datasheet ed. Intel Corporation, October 2001. 2, 53, 108, 139, 148, 149, 152

[66] INTEL. *Intel IA-32 Architecture Software Developer's Manual. Volume 3: System Programming Guide*, order number 245472 ed. Intel Corporation, Denver, CO, USA, 2001. 2, 46, 48, 50, 52, 72

[67] INTEL. *Enhanced Host Controller Interface Specification for Universal Serial Bus*, revision 1.0 ed. Intel Corporation, March 2002. 65

[68] INTEL. *Intel 82801CA I/O Controller Hub 3-S (ICH3-S)*, datasheet document number 290733-002 ed. Intel Corporation, March 2002. 53

[69] INTEL. *Intel 845 Chipset: 82845 Memory Controller Hub (MCH) for DDR*, datasheet ed. Intel Corporation, January 2002. 48, 50, 51, 95, 139, 149

[70] INTEL. *Intel 845 Chipset: 82845 Memory Controller Hub (MCH) for SDR*, datasheet ed. Intel Corporation, January 2002. 48, 50, 51, 95, 139, 148, 149

[71] INTEL. *Intel IA-32 Architecture Software Developer's Manual. Volume 3: System Programming Guide*, order number 245472-012 ed. Intel Corporation, Denver, CO, USA, 2003. 2, 46, 48, 50, 52, 72

[72] INTEL. *Mobile Intel Pentium 4 Processor-M*, datasheet order number: 250686-007 ed. Intel Corporation, June 2003. 53

[73] INTEL. *Mobile Intel Pentium 4 Processor Supporting Hyper-Threading Technology on 90-nm Process Technology*, datasheet document number: 302424-003 ed. Intel Corporation, January 2005. 53

[74] INTEL. *Intel CoreTM 2 Duo E6400, E4300, and Intel Pentium Dual-Core E2160 Processor Thermal Design Guide*, order number 315279-003us ed. Intel Corporation, October 2007. 53

[75] INTEL. *Intel Pentium Dual-Core Desktop Processor E2000 Series*, datasheet document number 316981-005 ed. Intel corporation, March 2008. 53

[76] INTEL. *Intel Core 2 Duo Processor E8000 and E7000 Series*, datasheet document number 318732-006 ed. Intel Corporation, June 2009. 53

[77] INTEL. *Mobile Intel 945GSE Express Chipset for Embedded Computing (2010)*, product brief 320217-002us ed. Intel Corporation, January 2010. 53

[78] INTEL. *Intel Core i5-600, i3-500 Desktop Processor Series, Intel Pentium Desktop Processor 6000 Series (Volume 1)*, datasheet document number 322909-006 ed. Intel Corporation, January 2011. 53

[79] INTEL. *Intel Core i5-600, i3-500 Desktop Processor Series, Intel Pentium Desktop Processor 6000 Series (Volume 2)*, datasheet document number 322910-003 ed. Intel Corporation, January 2011. 53

[80] INTEL. *Intel 5 Series Chipset and Intel 3400 Series Chipset*, datasheet ed. Intel Corporation, January 2012. 2, 53, 108, 139, 140, 142, 143, 151

[81] INTEL. *Intel 7 Series C216 Chipset Family Platform Controller Hub (PCH)*, datasheet ed. Intel Corporation, June 2012. 2, 33, 34, 51, 53, 108, 139, 140, 142, 143, 146, 151

[82] INTEL. *Intel Platform Controller Hub EG20T*, datasheet ed. Intel Corporation, July 2012. 33

[83] INTEL. *2nd Generation Intel Core Processor Family Desktop, Intel Pentium Processor Family Desktop, and Intel Celeron Processor Family Desktop (Volume 1)*, datasheet document number 324641-008 ed. Intel Corporation, June 2013. 53

[84] INTEL. *2nd Generation Intel Core Processor Family Desktop, Intel Pentium Processor Family Desktop, and Intel Celeron Processor Family Desktop (Volume 2)*, datasheet document number 324642-003 ed. Intel Corporation, June 2013. 53

[85] INTEL. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture.*, order number 253665-050us ed. Intel Corporation, February 2014. 2, 31, 34, 35

[86] INTEL. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z*, order number 325462-050us ed. Intel Corporation, February 2014. 2, 34, 56, 144

[87] INTEL. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3 (3A, 3B & 3C): System Programming Guide*, order number 325462-050us ed. Intel Corporation, February 2014. 2, 6, 10, 11, 12, 45, 46, 48, 49, 55, 56, 57, 59, 65, 66, 72, 73, 76, 96, 110, 143, 144, 145, 146, 152

[88] INTEL. *Intel Atom Processor N270 Series (update)*, document number 320047-009us ed. Intel Corporation, July 2014. Revision 009. 53

[89] INTEL. *Intel Automated Relational Knowledgebase*. Intel Corporation, 2014. 33, 34

[90] INTEL. *The Intel BIOS Implementation Test Suite (BITS)*, version 1090 ed. Intel Corporation, September 2014. 6, 11, 52, 62, 66, 73, 137

[91] INTEL. *Intel C610 Series Chipset and Intel X99 Chipset Platform Controller Hub (PCH)*, datasheet ed. Intel Corporation, September 2014. 57, 72, 142

[92] INTEL. *Intel Core M Processor Family*, datasheet ed. Intel Corporation, September 2014. Volume 1 of 2. 50, 51, 72, 139

[93] INTEL. *Intel Core M Processor Family*, datasheet ed. Intel Corporation, September 2014. Volume 2 of 2. 50, 51, 57, 72, 139

[94] INTEL. *Intel Software Guard Extensions Programming Reference*, order number 329298-002us ed. Intel Corporation, October 2014. 67

[95] INTEL. *Trusted Boot (tboot)*, version 1.8.2 ed. Intel Corporation, November 2014. 37

[96] INTEL. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z*, order number 325383-057us ed. Intel Corporation, December 2015. 144

[97] INTEL. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*, order number 325384-056us ed. Intel Corporation, September 2015. 2, 46, 48, 49, 52, 72, 76, 96, 143, 144, 145, 146, 152

[98] INTEL. *The Intel BIOS Implementation Test Suite (BITS)*, version 2073 ed. Intel Corporation, January 2016. 52, 53, 54

[99] JAEGER, T., SAILER, R., AND SHANKAR, U. Prima: Policy-reduced integrity measurement architecture. In *Proceedings of the Eleventh ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2006), SACMAT '06, ACM, pp. 19–28. doi:10.1145/1133058.1133063. 40

[100] KELLER, E., SZEFER, J., REXFORD, J., AND LEE, R. B. Nohype: Virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2010), ISCA '10, ACM, pp. 350–361. doi:10.1145/1815961.1816010. 42

[101] KING, S. T., CHEN, P. M., WANG, Y.-M., VERBOWSKI, C., WANG, H. J., AND LORCH, J. R. Subvirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2006), SP '06, IEEE Computer Society, pp. 314–327. doi:10.1109/SP.2006.38. 41

[102] KORTCHINSKY, K. Hacking 3d (and breaking out of vmware). In *Black Hat Conference* (Las Vegas, NV, July 2009). 43, 70, 131

[103] KOURAI, K., AND CHIBA, S. Hyperspector: Virtual distributed monitoring environments for secure intrusion detection. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments* (New York, NY, USA, 2005), VEE '05, ACM, pp. 197–207. doi:10.1145/1064979.1065006. 42

[104] KOVAH, X., KALLENBERG, C., WEATHERS, C., HERZOG, A., ALBIN, M., AND BUTTERWORTH, J. New results for timing-based attestation. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2012), SP '12, IEEE Computer Society, pp. 239–253. doi:10.1109/SP.2012.45. 60

[105] KUSSWURM, D. *Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX*. Apress, New York, 2014. 131

[106] THE LINUX KERNEL ARCHIVES. *LibPCI for Linux*, 2014. 63

[107] LITTY, L., LAGAR-CAVILLA, H. A., AND LIE, D. Hypervisor support for identifying covertly executing binaries. In *Proceedings of the 17th Conference on Security Symposium* (Berkeley, CA, USA, 2008), SS'08, USENIX Association, pp. 243–258. 41

[108] MATHER, T., KUMARASWAMY, S., AND LATIF, S. *Cloud Security and Privacy*. OReilly Media, Sebastopol, CA, USA, 2009. 19, 22

[109] MCCUNE, J. M., PARNO, B. J., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (New York, NY, USA, 2008), Eurosys '08, ACM, pp. 315–328. doi:10.1145/1352592.1352625. 42

[110] MCEVOY, T. R., AND WOLTHUSEN, S. D. Host-based security sensor integrity in multiprocessing environments. In *Proceedings of the 6th International Conference on Information Security Practice and Experience* (Berlin, Heidelberg, 2010), ISPEC'10, Springer-Verlag, pp. 138–152. doi:10.1007/978-3-642-12827-1_11. 43

[111] MCKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C. V., SHAFI, H., SHANBHOGUE, V., AND SAVAGAONKAR, U. R. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, 2013), HASP '13, ACM, pp. 10:1–10:1. doi:10.1145/2487726.2488368. 2, 10, 58

[112] MELL, P., AND GRANCE, T. The nist definition of cloud computing. Special Publication 800-145, National Institute of Standards and Technology, September 2011. 20, 21

[113] MENEZES, A., OORSCHOT, P. C., AND VANSTONE, S. A. *Handbook of applied cryptography*. CRC Press, Boca Raton, FL, USA, 1996. 39

[114] PETTEY, C., AND GOASDUFF, L. Gartner highlights five attributes of cloud computing. Site, June 2009. Accessed in 16/04/2016. 20

[115] POPEK, G. J., AND GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. In *Communications of the ACM* (July 1974), vol. 17, ACM, pp. 412–421. 25, 26, 28, 29

[116] PORTNOY, M. *Virtualization essentials*. Wiley, Hoboken, NJ, 2012. 23, 25, 30

[117] RIVEST, R. *The MD5 message-digest algorithm*, rfc 1321, 37 ed., April 1992. 39

[118] ROBIN, J. S., AND IRVINE, C. E. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *9th conference on USENIX Security Symposium* (Berkeley, CA, USA, August 2000), SSYM '00, USENIX Association. 28, 30

[119] RUTKOWSKA, J., AND WOJTCZUK, R. Detecting and preventing the xen hypervisor subervisions. In *BlackHat Conference* (Las Vegas, NV, 2008). 64

[120] SAILER, R., VALDEZ, E., JAEGER, T., PEREZ, R., DOORN, L. V., GRIFFIN, J. L., BERGER, S., SAILER, R., VALDEZ, E., LINWOOD, J., AND BERGER, G. S. shype: Secure hypervisor approach to trusted virtualized systems. Technical Report RC23511, IBM Research, 2005. 42

[121] SAILER, R., ZHANG, X., JAEGER, T., AND VAN DOORN, L. Design and implementation of a tcg-based integrity measurement architecture. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13* (Berkeley, CA, USA, 2004), SSYM'04, USENIX Association, pp. 16–16. 40

[122] SCHIFFMAN, J., AND KAPLAN, D. The smm rootkit revisited: Fun with usb. In *Proceedings of the 2014 Ninth International Conference on Availability, Reliability and Security* (Washington, DC, USA, 2014), ARES '14, IEEE Computer Society, pp. 279–286. doi:10.1109/ARES.2014.44. 64

[123] SCHNEIER, B. *Applied cryptography: protocols, algorithms and source code in C*, 2nd ed. Addison-Wesley Publishing Company Reading, Massashussets, USA, 1996. 39

[124] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 335–350. doi:10.1145/1294261.1294294. 42

[125] SEYFARTH, R. *Introduction to 64 bit Assembly Programming for Linux and OS X*. Hattiesburg, MS, USA, 2013. 131

[126] SHACKLEFORD, D. *Virtualization Security: Protecting Virtualized Environments*. Sybex, 2013. 19, 22

[127] SHANLEY, T. *Pentium Pro and Pentium II system architecture*, 2 ed. MindShare, Reading, MA, USA, 1998. 55

[128] SHINAGAWA, T., EIRAKU, H., TANIMOTO, K., OMOTE, K., HASEGAWA, S., HORIE, T., HIRANO, M., KOURAI, K., OYAMA, Y., KAWAI, E., KONO, K., CHIBA, S., SHINJO, Y., AND KATO, K. Bitvisor: A thin hypervisor for enforcing i/o device security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2009), VEE '09, ACM, pp. 121–130. doi:10.1145/1508293.1508311. 42

[129] SILBERSCHATZ, A., GALVIN, P., AND GAGNE, G. *Operating System Concepts*, 9 ed. Wiley, Hoboken, NJ, USA, 2013. 22, 23, 28

[130] SKOUDIS, E., AND ZELTSER, L. *Malware: fighting malicious code*. Prentice Hall, Upper Saddle River, NJ, USA, 2004. 41

[131] SOSINSKY, B. *Cloud Computing Bible*. Wiley, Hoboken, NJ, USA, 2011. vii, 19, 20, 21, 22

[132] STALLINGS, W. *Cryptography and network security: principles and practice*, 5th ed. Prantice Hall, Upper Saddle River, USA, 2010. 39

[133] STEINBERG, U., AND KAUER, B. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems* (New York, NY, USA, 2010), EuroSys '10, ACM, pp. 209–222. doi:10.1145/1755913.1755935. 42

[134] STINSON, R. D. *Cryptography: theory and practice*, 3rd ed. Chapman and Hall/CRC, Boca Raton, FL, USA, 2006. 39

[135] SZEFER, J., KELLER, E., LEE, R. B., AND REXFORD, J. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 401–412. doi:10.1145/2046707.2046754. 23, 42, 43, 71

[136] SZEFER, J., AND LEE, R. B. Architectural support for hypervisor-secure virtualization. *SIGPLAN Not. 47*, 4 (Mar. 2012), 437–450. doi:10.1145/2248487.2151022. 42

[137] TAKEMURA, C., AND CRAWFORD, L. S. *The Book of Xen : a Practical Guide for the System Administrator*. No Starch Press, San Francisco, CA, USA, 2010. 17, 27, 31, 84, 131

[138] TRUSTED COMPUTING GROUP. *Trusted platform*. 32, 33, 37, 38

[139] UNIFIED EXTENSIBLE FIRMWARE INTERFACE (UEFI). *Platform Initialization Specification, Pre-EFI Initialization Core Interface*, version 1.3 ed., May 2014. Accessed on 14/05/2014. 36

[140] VMWARE. Understanding full virtualization, paravirtualization, and hardware assist, 2007. Accessed in 20/03/2013. vii, 28, 29, 30

[141] WANG, J., STAVROU, A., AND GHOSH, A. Hypercheck: A hardware-assisted integrity monitor. In *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection* (Berlin, Heidelberg, 2010), RAID'10, Springer-Verlag, pp. 158–177. 6, 7, 8, 9, 10, 11, 41, 58, 61, 70, 73, 74, 137

[142] WANG, J., SUN, K., AND STAVROU, A. Hardware-assisted application integrity monitor. In *Proceedings of 45th Hawaii International Conference on System Science* (January 2012), HICSS, IEEE Computer Society, pp. 5375 – 5383. doi:10.1109/HICSS.2012.299. 2, 6, 7, 9, 11, 60, 61, 70, 73, 74

[143] WECHEROWSKI, F. A real smm rootkit: reversing and hooking bios smi handlers, November 2009. 2009. 51, 52, 64, 72

[144] WILKINS, R., AND RICHARDSON, B. Uefi secure boot in modern computer security solutions. White Paper (website Unified Extensible Firmware Interface Forum), September 2013. 36

[145] WOJTCZUK, R. Subverting the xen hypervisor. In *Black Hat Conference* (Las Vegas, NV, August 2008). 12, 29, 43, 70, 131

[146] WOJTCZUK, R., AND RUTKOWSKA. Xen 0wning trilogy. In *Black Hat Conference* (Las Vegas, NV, August 2008). 7, 58, 138

[147] WOJTCZUK, R., AND RUTKOWSKA, J. Attacking intel trusted execution technology. In *Black Hat Conference* (Las Vegas, NV, July 2009). 2, 41, 64

[148] WOJTCZUK, R., AND RUTKOWSKA, J. Attacking smm memory via intel cpu cache poisoning. Article on Internet, 2009. 2, 41, 47, 64, 65, 70, 74

[149] ZHANG, F., LEACH, K., SUN, K., AND STRAVOU, A. Spectre: a dependable instrospection framework via system management mode. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (2013), DSN. 6, 7, 9, 11, 60, 61, 70, 73, 74

[150] ZHANG, F., WANG, H. LEACH, K., AND STAVROU, A. A framework to secure peripherals at runtime. In *Proceedings of the 19th European Symposium on Research in Computer Security* (Wroclaw, Poland, September 2014), vol. 8712 of *Lecture Notes in Computer Science*, LNCS, pp. 219–238. 6, 7, 9, 11, 61, 70, 73, 74

[151] ZHANG, N., LI, M., LOU, W., AND HOU, Y. Mushi: Toward multiple level security cloud with strong hardware level isolation. In *Proceedings of Military Communications Conference* (October 2012), MILCOM 2012, IEEE, pp. 1 – 6. doi:10.1109/MILCOM.2012.6415698. 6, 7, 60, 61, 70, 73, 74

[152] ZHANG, Y., PAN, W., WANG, Q., BAI, K., AND YU, M. Hypebios: Enforcing vm isolation with minimized and decomposed cloud tcb. Technical report, Virginia Commonwealth University, 2012. 7, 60

[153] ZIMMER, V., ROTHMAN, M., AND MARISETTY, S. *Beyond BIOS: developing with the Unified Extensible Firmware Interface*, 2 ed. Intel Press, Hillsboro, OR, USA, 2010. vii, 37, 72