Detecting and Surviving Intrusions: Exploring New Host-Based Intrusion Detection, Recovery, and Response Approaches

Ronny Chevalier

To cite this version:

Ronny Chevalier. Detecting and Surviving Intrusions: Exploring New Host-Based Intrusion Detection, Recovery, and Response Approaches. Cryptography and Security [cs.CR]. CentraleSupélec, 2019. English. NNT : . tel-02417644

HAL Id: tel-02417644
https://inria.hal.science/tel-02417644

Submitted on 18 Dec 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT DE

CENTRALESUPELEC, RENNES
COMUE UNIVERSITÉ BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies
de l'Information et de la Communication
Spécialité : Informatique

Par
Ronny Chevalier

Detecting and Surviving Intrusions

Exploring New Host-Based Intrusion Detection, Recovery, and Response Approaches

Thèse présentée et soutenue à Rennes, le 17 décembre 2019
Unité de recherche : IRISA
Thèse N° : 2019CSUP0003

Rapporteurs avant soutenance :

Joaquin Garcia-Alfaro	Professeur, Telecom SudParis
Herbert Bos	Full Professor, Vrije Universiteit Amsterdam

Composition du Jury :

Examinateurs :	Joaquin Garcia-Alfaro	Professeur, Telecom SudParis
	Herbert Bos	Full Professor, Vrije Universiteit Amsterdam
	Karine Heydemann	Maître de conférences, Université Pierre et Marie Curie
	Guillaume Hiet	Maître de conférences, CentraleSupélec Rennes
	Laurence Pierre	Professeur, Université Grenoble Alpes
	David Plaquin	Senior Research Scientist, HP
Dir. de thèse :	Ludovic Mé	Advanced Research Position, Inria

ABSTRACT

Computing platforms, such as embedded systems or laptops, are built with layers of preventive security mechanisms to help reduce the likelihood of attackers successfully compromising them. Nevertheless, given time and despite decades of improvements in preventive security, intrusions still happen. Therefore, systems should expect intrusions to occur, thus they should be built to detect and to survive them.

Systems are monitored with intrusion detection solutions, but their ability to survive them is limited. State-of-the-art approaches from industry or academia either involve manual procedures, loss of availability, coarse-grained responses, or non-negligible performance overhead. Moreover, low-level components, such as the BIOS, are increasingly targeted by sophisticated attackers to implant stealthy and resilient malware. State-of-the-art solutions, however, mainly focus on boot time security, leaving the most privileged part of the BIOS—known as the System Management Mode (SMM)—a prime target.

The introduction of new solutions raises various challenges such as the security of the monitor, its ability to gather information about its target, the detection models, responding to intrusions and maintaining the availability of the system despite the presence of an adversary.

Our contribution is two-fold:

- At the OS-level, we introduce an intrusion survivability approach aimed at commodity OSs. We combine intrusion recovery and fine-grained cost-sensitive intrusion response to leverage a safe degraded mode when an intrusion is detected. Such a degraded mode prevents attackers to reinfect the system or to achieve their goals if they managed to reinfect it. It maintains the availability of core functions while waiting for patches to be deployed.

The initial idea behind this contribution was presented as a short-paper at RESSI'18 then the final work was published at ACSAC'19.

- At the BIOS level, we introduce an event-based and co-processor-based behavior monitoring approach to detect intrusions targeting the SMM on x86 platforms. We isolate the monitor using a co-processor to ensure its security and we bridge the semantic gap resulting from it by using a dedicated communication channel. This channel is used to send relevant information about the SMM code behavior that we compare with the model of its expected behavior—using invariants of its control-flow and relevant CPU registers.

This contribution has been published at ACSAC'17.

Keywords: Information Security, Intrusion Detection, Intrusion Response, Intrusion Recovery, Intrusion Survivability

PUBLICATIONS

This thesis is based on previously published papers written jointly with several collaborators:

INTERNATIONAL CONFERENCE PAPERS

- Ronny Chevalier, David Plaquin, Chris Dalton, and Guillaume Hiet. "Survivor: A Fine-Grained Intrusion Response and Recovery Approach for Commodity Operating Systems". In: *Proceedings of the 35th Annual Computer Security Applications Conference*. ACSAC'19. ACM, Dec. 2019. DOI: `10.1145/3359789.3359792`.
- Ronny Chevalier, Maugan Villatel, David Plaquin, and Guillaume Hiet. "Co-processor-based Behavior Monitoring: Application to the Detection of Attacks Against the System Management Mode". In: *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACSAC'17. ACM, Dec. 2017, pp. 399–411. DOI: `10.1145/3134600.3134622`.

NATIONAL CONFERENCE PAPERS

- Ronny Chevalier, David Plaquin, and Guillaume Hiet. "Intrusion Survivability for Commodity Operating Systems and Services: A Work in Progress". In: *Rendez-vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information*. RESSI'18. May 2018.

In addition, during the three years of this Ph.D. I was employed by HP, patent applications related to the work, ideas, or solutions presented in this document were filed:

PATENT APPLICATIONS

- Ronny Chevalier, David Plaquin, Guillaume Hiet, and Adrian Baldwin. "Mitigating Actions". Pat. req. Hewlett-Packard Development Company, L.P. May 2018.
- Ronny Chevalier, David Plaquin, Maugan Villatel, and Guillaume Hiet. "Intrusion Detection Systems". Pat. req. Hewlett-Packard Development Company, L.P. June 2017.

- Ronny Chevalier, David Plaquin, Maugan Villatel, and Guillaume Hiet. “Monitoring Control-Flow Integrity”. Pat. req. Hewlett-Packard Development Company, L.P. June 2017.

Finally, during these years I also contributed to another related area, but the following publication is not discussed in this thesis:

INTERNATIONAL CONFERENCE PAPER

- Ronny Chevalier, Stefano Cristalli, Christophe Hauser, Yan Shoshitaishvili, Ruoyu Wang, Christopher Kruegel, Giovanni Vigna, Danilo Bruschi, and Andrea Lanzi. “BootKeeper: Validating Software Integrity Properties on Boot Firmware Images”. In: *Proceedings of the 9th ACM Conference on Data and Application Security and Privacy*. CODASPY’19. ACM, Mar. 2019, pp. 315–325. DOI: `10.1145/3292006.3300026`.

ACKNOWLEDGMENTS – REMERCIEMENTS

No one does everything alone. Many people contributed—sometimes even without knowing it—to this research and dissertation either intellectually, financially, logistically, or personally. This is my attempt at acknowledging their help, interest, and contributions over the years.

First and foremost, I would like to thank Herbert Bos, Joaquin Garcia-Alfaro, Karine Heydemann, and Laurence Pierre for taking an interest in my work and for accepting to be members of the jury. Especially Herbert and Joaquin for their detailed review of this manuscript.

I was fortunate to be advised by Boris Balacheff, Guillaume Hiet, Ludovic Mé, and David Plaquin. They all shared different responsibilities and duties during this work, but they all provided me with their expertise, helpful criticism, and time. I want to thank in particular Guillaume and David whose advice, comments, and discussions helped shape this dissertation and my research in many respects.

During these three years I was also part of two teams: the CIDRE team at CentraleSupélec and the Security Lab at HP. I want to thank them for giving me an academic and industry perspective on research. I would like to thank the members of CIDRE for the scientific and technical discussions that I had with them over the years, but also all the "team building" we had at lUnchTime with the PhD students and interns. So thanks to all of them, especially to David Lanoë with his unrelenting force when hitting the cue ball, Pierre Graux with his nice collection of little orange men, Cédric Herzog with his love for Germany, Benoît Fournier who likes to keep a log of what we say, and Aïmad Berady le malicieux.

I also want to thank HP and especially the Security Lab at Bristol. Working with them gave me an insight at what it is like to work with a competent industry research lab. In particular, I would like to thank Philippa Bayley and Boris who worked hard to make sure that I could work at HP for my PhD. I would also like to acknowledge the expertise and time that Chris Dalton and Maugan Villatel provided me over the years; I was fortunate to have them as co-authors on some of my papers. I would also like to thank Vali Ali, Pierre Belgarric, Rick Bramley, Carey Huscroft, Jeff Jeansonne, and Thalia Laing for their feedback and technical discussions on my work. I am also grateful to Daniel Ellam, Jonathan Griffin, and Stuart Lees for their help in setting up and running some experiments in their malware lab, and Josh Schiffman for giving me opportunities to present my work at HP.

I would also like to thank François Bourdon and Laurent Jeanpierre. They were teachers of mine respectively in operating system and computer architecture during the first two years of my higher education. Both motivated me—probably without knowing it—to think about pursuing a career in research. François is also the reason why I went to Rennes for my studies. He told me that there was a research team in Rennes working on computer security, and he pointed me towards someone called Ludovic Mé. Unaware at the time that four years later I would be doing a PhD in this team and with Ludovic as my doctoral advisor.

I would also like to thank Jérémy and Martin who went from classmates, to friends, to best men at my wedding, and who also happened to follow a similar career path as PhD students.

Je vais terminer par remercier, en français, ma famille. En particulier, mes parents, qui m'ont toujours soutenu même s'ils ne réalisaient probablement pas que tout ce temps passé sur un ordinateur à l'époque allait être utile un jour. Puis, Léni, sans qui je ne me serais problablement jamais intéressé à l'informatique à l'origine. Enfin, Agathe, qui a relu le résumé français de ce manuscrit, et qui a surtout accepté de faire partie de ma vie.

CONTENTS

ABSTRACT iii
PUBLICATIONS v
ACKNOWLEDGMENTS – REMERCIEMENTS vii
ACRONYMS xv

I PROLOGUE
1 INTRODUCTION 3
1.1 Problem Statement . . . 3
1.1.1 Preventive Security is not Sufficient . . . 4
1.1.2 Commodity OSs Can Detect but Cannot Survive Intrusions . . . 5
1.1.3 Low-Level Components Increasingly Targeted . . . 6
1.2 Thesis . . . 7
1.3 Evaluation Approach . . . 8
1.4 Outline . . . 8
2 BACKGROUND: FROM X86 POWER-ON TO LOGIN PROMPT 9
2.1 BIOS and UEFI-Compliant Boot Firmware . . . 9
2.1.1 Platform Initialization . . . 9
2.1.2 Boot Time Security Mechanisms . . . 12
2.1.3 Runtime Services . . . 13
2.1.4 System Management Mode . . . 14
2.1.5 Attacks Against the SMM . . . 16
2.2 Operating Systems . . . 19
2.2.1 Kernel . . . 19
2.2.2 User Space Initialization, Service Manager, Login . . . 21
2.2.3 Isolation Primitives: The Case of the Linux Kernel . . . 21
2.2.4 Limitations of OS Security Principles . . . 23
2.3 Conclusion . . . 24
3 STATE OF THE ART: DETECTION AND SURVIVABILITY 25
3.1 Terms and Concepts . . . 25
3.1.1 Common Terms . . . 25
3.1.2 Intrusion Detection . . . 26
3.1.3 Intrusion Survivability and Related Concepts . . . 28
3.2 Intrusion Detection for Low-Level Components . . . 29
3.2.1 Isolation of the Monitor . . . 30
3.2.2 Hardware-Based Monitoring . . . 32
3.2.3 Detection Method . . . 35
3.2.4 Summary . . . 41

3.3 Intrusion Survivability for Commodity Operating Systems 42
    3.3.1 Isolation 42
    3.3.2 Intrusion Recovery 44
    3.3.3 Intrusion Response Systems 48
    3.3.4 Summary 54
3.4 Conclusion 55

II SURVIVING INTRUSIONS AT THE OS LEVEL

4 INTRODUCING AN INTRUSION SURVIVABILITY APPROACH 59
4.1 Motivation and Contributions 59
4.2 Approach Overview 61
4.3 Threat Model and Assumptions 62
4.4 Illustrating Examples 63
5 COST-SENSITIVE RESPONSE SELECTION 65
5.1 Models 65
    5.1.1 Malicious Behaviors and Responses 65
    5.1.2 Malicious Behavior Cost and Response Cost 67
    5.1.3 Response Performance 69
    5.1.4 Risk Matrix 70
    5.1.5 Policy Definition and Inputs 71
5.2 Optimal Response Selection 71
    5.2.1 Overview 71
    5.2.2 Pareto-Optimal Set 72
    5.2.3 Response Selection 72
6 ARCHITECTURE AND IMPLEMENTATION 75
6.1 Architecture and Requirements 75
    6.1.1 Overview 76
    6.1.2 Last Known Safe State 76
    6.1.3 Isolation of the Components 76
    6.1.4 Intrusion Detection System 77
    6.1.5 Service Manager 77
6.2 Linux-based Prototype Implementation 78
    6.2.1 Checkpoint and Restore 79
    6.2.2 Responses 80
    6.2.3 Monitoring Modified Files 81
    6.2.4 Bugs and Patches 81
7 EVALUATION AND RESULTS 83
7.1 Experimental Setup 83
7.2 Security Evaluation 84
    7.2.1 Responses Effectiveness 84
    7.2.2 Cost-Sensitive Response Selection 85

7.3 Performance Evaluation . . . 87
  7.3.1 Availability Cost . . . 87
  7.3.2 Monitoring Cost . . . 89
  7.3.3 Storage Space Overhead . . . 93
7.4 Stability of Degraded Services . . . 93
7.5 Summary . . . 93
8 CONCLUDING REMARKS 95
8.1 Discussion and Limitations . . . 95
8.2 Comparison with Related Work . . . 97
8.3 Conclusion and Future Work . . . 98

III DETECTING INTRUSIONS AT THE FIRMWARE LEVEL

9 INTRODUCING A SMM BEHAVIOR MONITORING APPROACH 103
9.1 Motivation . . . 103
9.2 Contributions . . . 104
9.3 Approach Overview and Requirements . . . 106
  9.3.1 Co-Processor . . . 106
  9.3.2 Communication with the Monitor . . . 107
  9.3.3 Instrumentation of the Target . . . 108
9.4 Threat Model and Assumptions . . . 108
10 DETECTION METHODS AND MODELS 111
10.1 Type-Based Control Flow Integrity . . . 111
  10.1.1 Overview and Motivation . . . 111
  10.1.2 Illustrating Examples . . . 112
  10.1.3 Code Analysis and Instrumentation . . . 114
  10.1.4 Shadow Call Stack . . . 115
10.2 Execution Context Integrity . . . 116
10.3 Isolation of the Models . . . 117
11 ARCHITECTURE AND IMPLEMENTATION 119
11.1 Co-Processor and Monitor . . . 119
11.2 Communication Channel . . . 120
  11.2.1 Existing Mechanisms . . . 120
  11.2.2 Restricted FIFO . . . 121
11.3 Instrumentation . . . 123
12 EVALUATION AND RESULTS 125
12.1 Experimental Setup . . . 125
  12.1.1 Simulator and Emulator . . . 125
  12.1.2 Simulated Communication Channel Delay . . . 126
  12.1.3 SMI Handlers . . . 126
12.2 Security Evaluation . . . 127
12.3 Performance Evaluation . . . 129

12.3.1 Runtime Overhead 129
12.3.2 Co-Processor Performance 130
12.3.3 Firmware Size 131
12.4 Summary 132
13 CONCLUDING REMARKS 133
13.1 Discussion and Limitations 133
13.2 Comparison with Related Work 134
13.3 Conclusion and Future Work 136

IV EPILOGUE

14 CONCLUSION 139
14.1 Summary of the Contributions Supporting Our Claims 139
14.2 Perspectives 140
14.2.1 Extend the Approaches and their Evaluation 141
14.2.2 Surviving and Adapting Intrusions 141

APPENDICES

A MALWARE SAMPLES 145
B GEM5 PARAMETERS 147

RÉSUMÉ SUBSTANTIEL EN FRANÇAIS 149
BIBLIOGRAPHY 157
COPYRIGHT PERMISSIONS 183

LIST OF FIGURES

Figure 1.1 The Maginot Line during WWII 4
Figure 1.2 Computer abstraction layers 6
Figure 1.3 Computer abstraction layers covered in this dissertation 7
Figure 2.1 Simplified diagram of a recent Intel x86 architecture 10
Figure 2.2 UEFI PI phases and the UEFI interfaces exposed during the boot sequence to the OS 12
Figure 2.3 Simplified diagram of various boot security mechanisms 13
Figure 2.4 SMM execution flow and SMRAM memory layout 15
Figure 2.5 Difference of isolation between main kernel types 20
Figure 3.1 Timeline of a transient attack 33
Figure 3.2 A prover attests its integrity via a challenge-response protocol . 36
Figure 4.1 High-level overview of our intrusion survivability approach . . 61
Figure 5.1 Example of a non-exhaustive malicious behavior hierarchy . . . 66
Figure 5.2 Example of a non-exhaustive per-service response hierarchy . . 67
Figure 6.1 Overview of the architecture of our intrusion survivability approach 75
Figure 7.1 Impact of checkpoints on the latency of HTTP requests made to an nginx server 89
Figure 7.2 Results of synthetic benchmarks to measure the overhead of the monitoring 91
Figure 7.3 Results of real-world workload benchmarks to measure the overhead of the monitoring 92
Figure 9.1 High-level overview of our co-processor-based monitoring approach 106
Figure 10.1 Simplified view of the stack frame of the function before the attacker overwrites `functions[0]` 112
Figure 10.2 Example of the mappings that the source code analysis outputs 114
Figure 10.3 How the monitor detects illegitimate indirect calls 115
Figure 10.4 How the monitor detects illegitimate returns using a shadow call stack 116
Figure 11.1 High-level overview of our architecture that monitors the SMM 119
Figure 12.1 Time to execute SMI handlers divided between the communication and the instrumentation overhead 129
Figure 12.2 Time to process all the messages sent by one execution of each SMI handler for the co-processor 131

LIST OF TABLES

Table 2.1 Summary of the main attacks targeting and vulnerabilities affecting the SMM with their countermeasures 18
Table 5.1 Example of a 5 × 5 risk matrix that follows the requirements for our risk assessment 70
Table 6.1 Projects modified for the implementation of our intrusion survivability approach 79
Table 7.1 Summary of the experiments that evaluate the effectiveness of the responses against various malicious behaviors 85
Table 7.2 Responses to withstand ransomware reinfection with their associated cost and performance for Gitea 86
Table 7.3 Time to perform the checkpoint operations of a service 88
Table 7.4 Time to perform the restore operations of a service 90
Table 8.1 Summary of the comparison between our intrusion survivability approach and the related work 97
Table 12.1 Effectiveness of our approach against state-of-the-art attacks . . 128
Table 12.2 Number of packets sent during the execution of one SMI handler 130
Table 13.1 Summary of the comparison between our SMM behavior monitoring approach and the related work 135
Table A.1 Malware used in our experiments with the SHA-256 hash of the samples 145
Table B.1 Parameters used with gem5 for the x86 and the ARM simulation 147

LIST OF CODE SNIPPETS

Listing 10.1 Example of a simulated non-SMM vulnerable code 113
Listing 10.2 Example of a vulnerable function from a real SMI handler based on decompiled code 113

ACRONYMS

AL	After Life. 11, 12
AP	Application Processor. 11
ATRA	Address Translation Redirection Attack. 34

BAR	Base Address Register. 18
BDS	Boot Device Selection. 11–13
BIOS	Basic Input/Output System. iii, 6, 7, 9–19, 24, 32, 45, 55, 103, 104, 109, 112, 123, 125, 136, 151, 152, 154, 155
BITS	BIOS Test Suite. 16, 120
BSP	Bootstrap Processor. 9

C&C	Command and Control. 64–66, 84, 85
CFG	Control-Flow Graph. 38–40, 111, 128, 132, 154
CFI	Control-Flow Integrity. 34, 38–41, 105, 106, 111–113, 115, 116, 118, 123, 128, 133–136
COW	Copy-On-Write. 79

DAC	Discretionary Access Control. 22
DDoS	Distributed Denial-of-Service. 64, 84
DKOM	Direct Kernel Object Manipulation. 37
DMA	Direct Memory Access. 15, 32, 33, 37
DNC	Democratic National Committee. 6
DOS	Denial Of Service. 121
DXE	Driver Execution Environment. 11–13, 15, 117, 123

FIFO	First In First Out. 121, 122, 125, 126, 129, 135, 155

IDS	Intrusion Detection System. 5, 27–31, 33, 35, 41–43, 45, 46, 50, 53, 61, 64, 66, 70, 71, 76, 77, 86, 87, 95, 98, 150
IDT	Interrupt Descriptor Table. 36
IETF	Internet Engineering Task Force. 25, 26

IPC	Inter-Process Communication. 20, 22, 77
IR	Intermediate Representation. 123
JOP	Jump-Oriented Programming. 38
KOH	Kernel Object Hooking. 37
MAC	Mandatory Access Control. 22, 76–78
MAEC	Malware Attribute Enumeration and Characterization. 65, 67, 77, 153
MBR	Master Boot Record. 11
MMU	Memory Management Unit. 9, 22, 23, 35, 36, 41, 135
MOO	Multi-Objective Optimization. 71, 72
NIST	National Institute of Standards and Technology. 28
Or-BAC	Organization-Based Access Control. 53
OS	Operating System. iii, xiii, 3–9, 11–16, 18–25, 29, 31, 34, 36, 42, 45, 46, 49, 50, 54, 55, 59, 61, 62, 66, 71, 76–78, 98, 103, 104, 109, 139, 141, 142, 150–152
P2P	Peer-to-peer. 84
PCH	Platform Controller Hub. 9–11
PCI	Peripheral Component Interconnect. 11, 18, 32, 33
PEI	Pre-EFI Initialization. 11–13
PI	Platform Initialization. xiii, 11, 12
PID	Process IDentifier. 21
POSIX	Portable Operating System Interface. 22
PSP	Platform Security Processor. 119–121
QPI	QuickPath Interconnect. 121, 125, 126, 155
ROP	Return-Oriented Programming. 38, 113–115
RPC	Remote Procedure Call. 77
RT	Runtime. 11, 12
SCM	Service Control Manager. 21

SCRTM	Static Core Root of Trust for Measurement. 13
SEC	Security. 11–13
SEP	Secure Enclave Processor. 119–121
SMI	System Management Interrupt. xiii, xiv, 15–18, 31, 108, 109, 112–114, 116, 117, 120–122, 125–127, 129–133, 140, 142, 154, 155
SMM	System Management Mode. iii, xiii, xiv, 14–19, 24, 25, 30–34, 36, 37, 39–41, 55, 103–106, 108, 109, 111, 112, 114, 116, 117, 119–123, 125–136, 140–142, 151, 153–156
SMRAM	System Management RAM. xiii, 15–19, 32, 109, 114, 115, 117, 126–128, 154
SMRAMC	System Management RAM Control. 15, 16
SMRR	System Management Range Register. 17, 18
SOO	Single-Objective Optimization. 72
STIX	Structured Threat Information eXpression. 69, 77
STM	SMI Transfer Monitor. 31, 142

TCB	Trusted Computing Base. 43
TEE	Trusted Execution Environment. 14
TLS	Transient System Load. 11, 12
TPM	Trusted Platform Module. 13, 36, 127

UEFI	Unified Extensible Firmware Interface. xiii, 11–15, 44, 104, 125, 127, 154

VLAN	Virtual Local Area Network. 83
VM	Virtual Machine. 43, 45, 83

Part I

PROLOGUE

1

INTRODUCTION

> *I am convinced that there are only two types of companies: those that have been hacked and those that will be. And even they are converging into one category: companies that have been hacked and will be hacked again.*
>
> — **Robert Mueller**
> Director of the FBI from 2001 to 2013

Organized crime, corporate espionage, opportunistic attacks, state-sponsored attacks, or activism, are all part of the threat landscape [188] that organizations and individuals have to take into account for the security of their computing platforms (e.g., laptops, servers, or smartphones). Decades of research in information security provided us with a large number of preventive security mechanisms—such as cryptography, access controls, and network security—that reduce the likelihood of these actors to successfully compromise such platforms.

Why, despite such mechanisms, do intrusions still happen? Why are we still not able to build secure systems by design? Multiple factors come into play that need to be explained to answer these questions. Before exposing the thesis behind this dissertation, we discuss these factors and the different problems that current systems face.

1.1 PROBLEM STATEMENT

Preventive security mechanisms generally aim to prevent an attacker from violating the following security properties during the execution of any system or application on those platforms:

CONFIDENTIALITY Information must not be disclosed to unauthorized parties.[1] For example, some cryptography primitives enforce this property. By encrypting the content of a document, only the parties in possession of the decryption key can read it—an attacker cannot.

INTEGRITY Information must not be tampered by unauthorized parties or without being detected. For example, access control mechanisms enforce this property when ensuring that only administrators can modify the configuration of a system.

AVAILABILITY Information must be available to authorized parties when requested. For example, on modern Operating Systems

[1] The word party is used in the general sense and applies to entities, processes, systems, or individuals.

(OSs), unprivileged processes cannot kill core components of the OS. Otherwise, an attacker could make the system unavailable to its users.

1.1.1 *Preventive Security is not Sufficient*

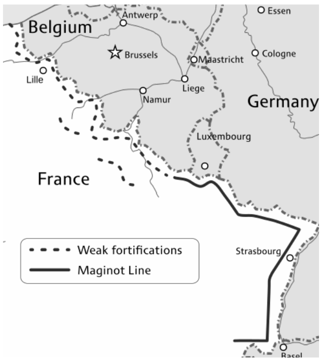


Figure 1.1: The Maginot Line during WWII

The mechanisms that we use to enforce the aforementioned properties, however, are not flawless. Indeed, much like the strong fortifications of the Maginot Line from World War II (WWII) that the Germans went around—by going through the Netherlands, Belgium, and Luxembourg—determined attackers will find a way to bypass defenses. One could easily think that France just had to expand the line to the coast, thus preventing any invasion from the Germans if they were to go through Belgium (which they did). France anticipated such an attack, but while it was straightforward to spot such weaknesses, expanding the line was not. Due to politics (Belgium was an ally when the construction started), wrong assumptions (they thought the German would not go through the Ardennes forest), time and budget constraints (hundreds of kilometers to cover with costly fortifications), only weak fortifications were eventually built along that border.[2]

> *"Security isn't a matter of absolutes; it's a matter of picking the best set of strategies given assorted constraints and objectives."*
> — Steven Bellovin, *Thinking Security*

Information security is similar in that regard. While many systems are designed with preventive security mechanisms (strong fortifications), the people designing and building these systems can make mistakes and are also constrained by time, money, or internal politics within their organization. These issues mean that information systems can only prevent the violation of security properties to a certain extent.

Furthermore, in contrast to troops on the battlefield, most threat actors do not have to take unnecessary risks, since they can attack information systems from a distance. Either via Ethernet, Wi-Fi, Bluetooth, or the cellular network, devices are exposed and can be attacked continuously, until compromised, without any attacker physically present. Thus, for example, it reduces the entry barrier for malicious individuals since it is perceived as a less risky endeavor than traditional crime.

> *"Real world attackers are of course not physically stopped by mathematical proofs."*
> — Cormac Herley and P.C. van Oorschot, "SoK: Science, Security and the Elusive Goal of Security as a Scientific Pursuit"

Even formally verified secure systems will not keep determined attackers from compromising a system. While they can eliminate classes of attacks, formal proofs rely on assumptions that can turn out to be wrong—much like the Ardennes forest that France assumed that the Germans will not cross. Such wrong assumptions can then be exploited by attackers to violate security properties.[3]

More technically speaking, we consider that intrusions—attacks that successfully compromise a system—may happen due to the combination of two sources: technical and economic reasons. Technical reasons—such as a misconfiguration, a wrong assumption, a system not updated, or an unknown vulnerability—render a system vulnerable. Attackers, or threat actors, can exploit a vulnerability to violate the security policy of the system to achieve their goals (e.g., steal confidential information). It is common to think that by only considering technical measures (e.g., by deploying firewalls and two-factor authentication) intrusions will be stopped. Economic reasons, however, are also in play.[4] They are the different incentives that drive the decisions that organizations and individuals make—attackers or defenders. For example, do the benefits of an intrusion for attackers outweigh their costs in terms of time and money? (cost-benefit principle)[5]

---

[2] Their strategy was to push the battles inside Belgium to avoid a front in French soil.

[3] See the work of Fonseca et al. [112] that studied formally verified systems and found several violations of assumptions leading to various bugs.

Given these factors, we arrive at the conclusion that when building systems with security in mind, one must always remember that *given time, an intrusion will occur*. It means that we should not only build systems to prevent intrusions, but also to *detect* and *survive* them.

1.1.2 *Commodity OSs Can Detect but Cannot Survive Intrusions*

The idea of Intrusion Detection Systems (IDSs)—systems that automatically detect intrusions—dates back to the 1980s.[6] Since then, more intrusion detection approaches were introduced, refined, and transferred from academia to industry. Most of today's commodity OSs are deployed with some kind of IDS—a well-known example would be anti-virus software which share many aspects of host-based IDSs.[7] Likewise, we find IDSs not only at the host level, but also at the network level [217]. However, as the name suggests, an IDS only focus on the detection and do not provide the ability to survive or withstand an intrusion once it has been detected.

Systems that aim at automatically withstanding intrusions exist. They are associated with various close and overlapping concepts from the literature such as intrusion tolerance, intrusion survivability, self-protecting systems, intrusion recovery, intrusion response, and intrusion resiliency. For example, the concept of intrusion tolerant systems—systems that can maintain their security properties even when some of their components are compromised—also dates back to the 1980s [116].

Most of the research on intrusion response, intrusion recovery, or intrusion survivability focuses on critical infrastructure, distributed systems, and networks [105, 157, 239, 294]. The ability of commodity OSs to withstand intrusions, however, is less studied. For example, nowadays when a system is compromised administrators have two choices while waiting for patches to fix the vulnerabilities. First, they can stop the compromised system. It would ensure the integrity and confidentiality properties, but they would lose availability. Unfortunately, the cumulative time to analyze the system to find the vulnerability, to either wait for a patch from the vendor or to develop the patch, to test the patch, and to finally deploy the patch, can be long (e.g., several days) depending on the organization. Hence, the system can be offline for a long time until it is patched. Second, administrators can restore the system to a previous safe state. It would lose some availability (some information might be lost due to the restoration), but the system would be online. However, it would remain vulnerable and nothing would stop attackers from reinfecting the system again.

---

[4] See the seminal work by Anderson [8] and Anderson and Moore [9] that studies information security with an economic perspective.

[5] Contrary to popular belief, the Maginot Line was successful in a way—it worked as a deterrent—since the German army considered that the cost of invading France through the north-east border was too high—avoiding a cross-border assault.

[6] See the work of Anderson [7] and Denning [90].

[7] See the work of Morin and Mé [203] that studied the link between IDSs and anti-virus software.

While there are solutions to help OSs to either recover from an intrusion [122, 286] or limit the impact of the intrusion on the system [20], these solutions have various limitations that hamper their deployment. For example, these solutions can incur a loss of availability by forcing a reboot of the system or an application. They can also apply coarse-grained responses that affect the rest of the system or the whole application in order to thwart one specific intrusion. Finally, they do not necessarily prevent the system from being reinfected or do not stop the attackers from achieving their goals if they manage to reinfect the system. We arrive at the conclusion that while commodity OSs can detect intrusions, current state-of-the-art solutions either from academia or industry do not allow these systems to survive intrusions once they have been detected.

1.1.3 *Low-Level Components Increasingly Targeted*

We argued that preventive security mechanisms are not sufficient and that while today's OSs can detect some intrusions, they cannot automatically withstand them without various limitations. The state of security of applications and OSs, however, improved nonetheless since the 1980s or 1990s for example. These improvements mean that it becomes more difficult for attackers to compromise systems at the application and OS abstraction layers, or at least to do it stealthily. It results in an increased focus by sophisticated and well-resourced attackers on lower abstraction layers [118, 176, 181, 226].

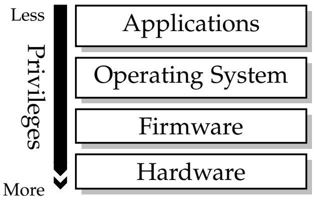


Figure 1.2: Computer abstraction layers

Firmware—software developed by device manufacturers—is one of such lower abstraction layers, as illustrated in Figure 1.2. It can be found in motherboards with the flash containing the Basic Input/Output System (BIOS), storage devices, network cards, graphic cards, or many other components that computers rely on. Firmware is present in all kind of platforms whether it is servers, laptops, or industrial systems.

Due to its direct access to the hardware and its often-early execution, such a low-level piece of software is highly privileged. Hence, any alteration to its expected behavior, malicious or not, can have dramatic consequences for the confidentiality, integrity, or availability of the system. We need to ensure that firmware, such as the BIOS, has not been compromised. Otherwise, attackers can control any upper layer software components, such as the OS, and can render moot any security solution present.

For example, in 2018, the malicious threat actor known as APT28, Fancy Bear, or Sednit,[8] used attacks against the BIOS to compromise some of its targets [226]. These attacks allowed them to have a stealthy malware implant in the BIOS that survived even if you reinstalled the OS or removed the storage devices.

---

[8] They are believed to be responsible for the attacks on the Democratic National Committee (DNC) [3] and the French television network TV5 Monde [273].

While solutions exist to ensure firmware integrity (with detection and recovery) at boot time, runtime firmware code—that executes when the OS is running—has not gotten the same level of attention. Runtime firmware code security relies mainly on preventive security mechanisms (e.g., memory protections [289]). However, as concluded previously, such mechanisms are not sufficient. Without intrusion detection mechanisms, the most privileged components of our computing platforms remain unmonitored and a prime target for sophisticated attackers.

Unfortunately, low-level components introduce challenges for any intrusion detection solution. Such components must respect hard constraints in terms of resource usage to not degrade the user experience, and they are the most privileged on the platform. Hence, any modification might increase their resource usage and attackers with the same privileges might impede any monitoring.

1.2 THESIS

This dissertation shows that computing platforms can be designed to detect intrusions at the firmware level and withstand intrusions at the OS level without significantly impacting the quality of service to users. First, by demonstrating that intrusion survivability is a viable approach for commodity OSs. Second, by developing a hardware-based approach that detects attacks at the firmware level.

More precisely, in this dissertation, we introduce and validate the two following claims:

Claim 1. OSs can survive intrusions by restoring the infected services to a previous state and by temporarily leveraging a degraded mode with fine-grained cost-sensitive responses while waiting for more long-term fixes.

Claim 2. Attacks targeting highly-privileged low-level components at runtime, such as the BIOS, can be detected by using an event-based and co-processor-based behavior monitoring approach without degrading the user experience of the platform.

Each claim focuses on different abstraction layers. As illustrated in Figure 1.3, the former focuses on the OS and applications layers, while the latter focuses on detecting intrusions at the firmware layer.

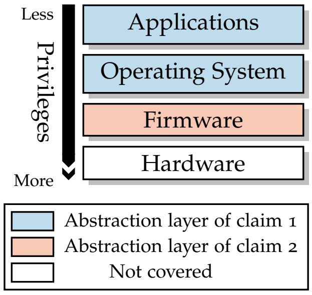


Figure 1.3: Computer abstraction layers covered in this dissertation

1.3 EVALUATION APPROACH

In this work, we proposed new approaches and architectures to tackle the aforementioned challenges. We took into account real-world constraints, and we relied on realistic use cases to drive our design choices.

In addition, we evaluated our solutions and validated our claims by developing proof-of-concept and prototype implementations that required significant development efforts. This work has thus a strong experimental focus.

For each claim, we evaluated both the security of the solution against real-world attacks and its performance (e.g., runtime overhead or storage space overhead) using real-world and synthetic benchmarks. Each time, we described our experimental setup, how we measured the performance overhead, and how we validated that our solution was successful.

Finally, we also discussed some limitations (or threats to validity) that our implementations and evaluations have, and how we addressed them. It allows us to reason about the level of confidence we have in the results and how they can be generalized.

1.4 OUTLINE

This dissertation contains four parts: a prologue, two independent main parts with our contributions, and an epilogue. Part I, the prologue, contains this introduction, Chapter 2, which provides an overview and general background on the various components involved in the boot process with a security perspective, and Chapter 3, a chapter on state-of-the-art intrusion detection, intrusion recovery, and intrusion response, focusing on our use cases (i.e., host-based approaches, low-level components, and commodity OSs). As mentioned previously, the two claims, introduced in this dissertation, target different abstraction layers. Hence, we split the rest of the dissertation in two main independent parts. Part II describes our first main contribution that about how to design an OS able to survive intrusions.[9] Part III describes our second main contribution about how to detect intrusions in runtime firmware.[10] Finally, Part IV, the epilogue, summarizes our contributions and provides some perspectives for future work.

[9] The initial idea was presented at RESSI'18 [59], then the final work was published and presented at ACSAC'19 [58].

[10] This work was published and presented at ACSAC'17 [60].

2

BACKGROUND: FROM X86 POWER-ON TO LOGIN PROMPT

In this chapter, we describe with a security perspective what happens from the moment you press the power up button of your computer until the OS can launch basic services—such as a login prompt. We do not describe necessarily in detail these steps, but we provide the required background to understand the subsequent parts of this dissertation. While our focus is on x86 platforms, the general ideas are similar to other platforms—such as ARM.

The rest of this chapter is structured as follows. First, we provide the various steps of the initialization of the platform performed by the BIOS, why such steps are required, their complexity, and most importantly how we can and why we must guarantee security properties from the start (Section 2.1). We also mention various attacks and vulnerabilities against some runtime BIOS code. Such a background is important to understand the context and the models we use in the work described in Part III, and the reasoning behind our contributions. Second, we describe the key components of an OS and the various isolation primitives that one can use to enforce security policies on services or applications (Section 2.2). This section is important to understand Part II.

2.1 BIOS AND UEFI-COMPLIANT BOOT FIRMWARE

After the power button has been pressed, the motherboard waits for a signal—known as the power good signal—that the power supply has stabilized its voltage output before allowing the main processor (CPU) to start executing. The initial state of the CPU and the platform, however, is limited. For example, the CPU has no cache enabled, the Memory Management Unit (MMU) is disabled, the system memory (RAM) is not initialized[11] and cannot be used, the CPU is in a special legacy 16-bit mode known as the unreal mode, and only one core—also known as the Bootstrap Processor (BSP)—is active [218].

2.1.1 *Platform Initialization*

The goal of boot firmware, such as the BIOS, is to initialize the hardware (e.g., CPU or RAM) before any subsequent software can take over (e.g., an OS). The BIOS code is stored inside a small flash (e.g., 16 MB or 32 MB) on the motherboard. The flash is connected to the Platform Controller Hub (PCH)[12] which is connected to the CPU—as

[11] It means there is no stack available so no higher-level languages, such as C, can be used.

[12] The PCH exists since the Intel Nehalem architecture (2008). Previously, Intel processors were connected with multiple chips often referred as north and south bridge. Nowadays, the northbridge is integrated into the processor, and the processor is connected to the PCH that handles the southbridge's features [261].

illustrated in Figure 2.1. When the CPU starts executing, it fetches code at a hard-coded physical address (`0xFFFFFFF0` also known as the reset vector). This address is mapped to the BIOS flash, meaning that when the CPU fetches code at this address it retrieves code from the BIOS flash.

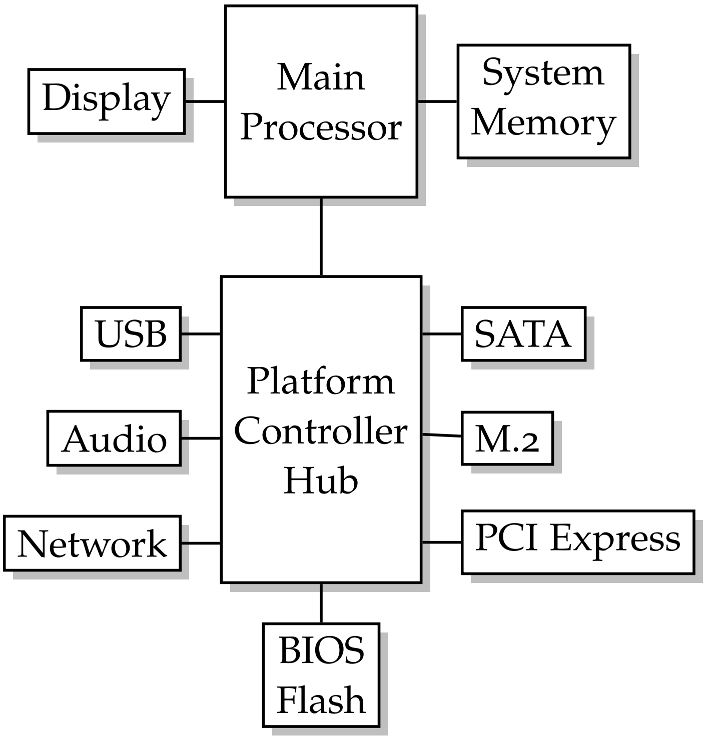


Figure 2.1: Simplified diagram of a recent Intel x86 architecture [270]

Since fetching code and data from the flash is slow, the goal of the initial BIOS code is to set up as soon as possible the RAM [218]:

1. Switch from 16-bit unreal mode to flat protected mode.
2. Apply processor microcode update.
3. Set the CPU cache as a RAM to set up a stack.[13]
4. Initialize the PCH.[14]
5. Initialize and test the RAM.[14]
6. Copy the BIOS flash content into the RAM.
7. Set up the stack in RAM and disable the previous cache-as-RAM mechanism.
8. Jump to BIOS code now stored in RAM.

This enumeration is only the early initialization phase. The BIOS code still needs to initialize interrupt controllers, interrupt tables, timers, real time clock, to discover and initialize other cores—also

[13] Now it is possible to use C code.

[14] While summarized in one line, such an initialization is a complex process to implement which is often described in documents of hundreds of pages.

known as Application Processors (APs). Moreover, it needs to discover Peripheral Component Interconnect (PCI) compliant devices and to assign them resources (e.g., memory space or interrupt request registers). For example, the various components connected to the left or the right of the PCH on Figure 2.1 are PCI compliant devices.

In our context, one important aspect of the initialization is the setup of runtime firmware code. It will be used to handle critical services when the OS is running. We will go into more details about such runtime firmware code in Section 2.1.4.

After the initialization, discovery, and configuration phases, the BIOS needs to select and hand-off the control to a boot loader or an OS. Legacy BIOSs rely on the Master Boot Record (MBR)—a 512-byte data structure at the beginning of hard drives. It contains a partition table and a bootstrap code used to execute the next phase.

Recent BIOSs follow the Unified Extensible Firmware Interface (UEFI) and the UEFI Platform Initialization (PI) specifications [271, 272, 298].[15] The former allows OSs or boot loaders to use standard interfaces to communicate with the BIOS, while the latter helps the various firmware vendors involved in the implementation of a BIOS to have implementations that follow standard interfaces between the different boot phases. The phases, illustrated in Figure 2.2, are the following:

1. The Security (SEC) phase contains the early initialization that, among other things, switches from real mode to protected mode, initializes the memory space to run stack-based C code, and discovers, verifies, and executes the next phase.
2. The Pre-EFI Initialization (PEI) phase initializes permanent memory, handles the different states of the system (e.g., recovery after suspending), and executes the next phase.
3. The Driver Execution Environment (DXE) phase contains most of the platform initialization, since it initializes the chipset, discovers and executes drivers which initialize platform components.
4. The Boot Device Selection (BDS) phase chooses the device (e.g., a hard drive) to boot from and executes its boot loader.
5. The Transient System Load (TLS) phase executes the boot loader from the OS or handles special UEFI applications.
6. The Runtime (RT) phase is when the OS is executing, but there are still runtime services available to communicate with the OS.
7. The After Life (AL) phase takes control back over the OS when it has shutdown, crashed, or it is hibernating.

In this dissertation, we use the term BIOS as an inclusive way to mean both *legacy BIOSs* and *UEFI-compliant firmware*. We only use

[15] Based on the initial work of Intel, 11 industry vendors (e.g., Intel, HP, and Apple) formed the UEFI Forum in 2005 that is responsible for the standardization of the UEFI and UEFI PI specifications [298, Chapter 1].

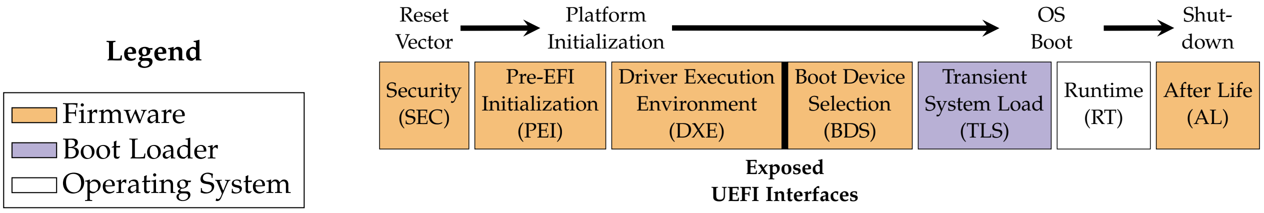

Legend
Firmware
Boot Loader
Operating System
Reset Vector
Platform Initialization
OS Boot
Shut-down
Security (SEC)
Pre-EFI Initialization (PEI)
Driver Execution Environment (DXE)
Boot Device Selection (BDS)
Transient System Load (TLS)
Runtime (RT)
After Life (AL)
Exposed UEFI Interfaces

Figure 2.2: UEFI PI phases and the UEFI interfaces exposed during the boot sequence to the OS

specifically the former term when we talk about boot firmware that do not follow UEFI specifications, and the latter term for when we want to specifically talk about firmware that follow these specifications.

2.1.2 *Boot Time Security Mechanisms*

As mentioned in the introduction,[16] due to its early execution and its direct access to the hardware, the BIOS is considered a highly privileged software. It has access to and can configure almost all pieces of hardware and software on the platform, and it is responsible for the execution of the next component in the boot sequence (e.g., an OS). Hence, if we want to trust that the various components of the platform, the OS, and the applications behave as expected, one must first have guarantees that the BIOS does as well.

The UEFI specifications increased the extensibility and the interoperability of firmware implementations, allowing multiple vendors to contribute to various drivers and parts of a BIOS. Unfortunately, it also increased the attack surface of the BIOS with more components exposed through various interfaces and a more complex code base due to its various features.[17] Moreover, some drivers or other software components might be coming from various vendors following potentially different practices in terms of software development and security. Finally, the platform manufacturer must also trust that the code in its platform is provided by the legitimate vendors and has not been compromised.

To guarantee the integrity of the platform software, various solutions (that can be combined) have been proposed and implemented. We illustrated them in Figure 2.3 and categorized them in the following list:

CRYPTOGRAPHICALLY SIGNED UPDATES The BIOS verifies that the update (UEFI Capsule) is cryptographically signed (→) by the legitimate vendor before applying it [68, 225].

VERIFIED BOOT Each component of the boot sequence verifies (→) that the subsequent ones are cryptographically signed by the

---

[16] See Section 1.1.3.

[17] For example, recent BIOSs have a network stack with support to boot from a remote image over HTTP [278] and have a Bluetooth stack [182] to be able to use devices such as a wireless mouse.

[18] In theory we could verify each components individually, but in practice for performance reasons a set of components are verified.

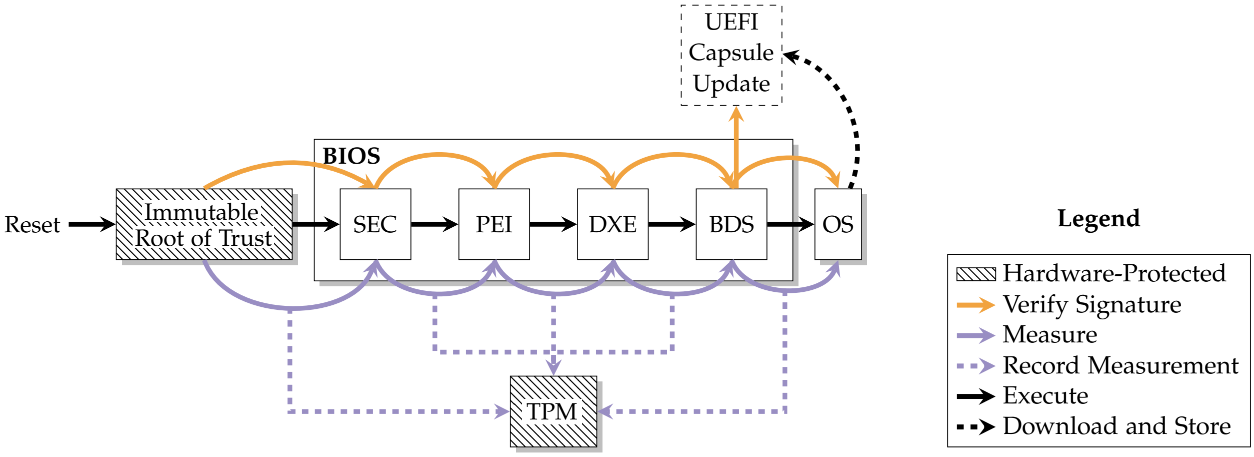


Figure 2.3: Simplified diagram of various boot security mechanisms

legitimate vendor before executing it.[18] For example, UEFI Secure Boot [272, section 32] has been designed to ensure that UEFI-compliant firmware only executes trusted OSs or boot loaders.

MEASURED BOOT Each component of the boot sequence takes cryptographic measurements (hashes) of the next ones (→) and records them (⇢) into a Trusted Platform Module (TPM) [268] chip—a passive tamper-resistant cryptographic co-processor. It allows the platform to prove to other services[19] its configuration and which components were executed.

IMMUTABLE CORE ROOT OF TRUST The core trust of the various solutions exposed relies on the integrity of the first component executed that performs these checks or measurements—also called the Static Core Root of Trust for Measurement (SCRTM). This component must be immutable and hardware protected to ensure its integrity.[20]

These various mechanisms can give us strong guarantees in terms of boot time security. Having described these mechanisms and why they are needed, we know discuss the phase that is less known about the BIOS which is its runtime part, and its security.

2.1.3 *Runtime Services*

UEFI-compliant firmware provides runtime services with standard interfaces that may be used by other software executed in the UEFI environment (e.g., an OS) [298, Chapter 5]. The services are exposed via the UEFI Runtime Services Table [272, Section 4.5] that contains pointers to the associated functions that the OS can call. These services

[19] BitLocker—Microsoft's drive encryption solution—relies on such guarantees. It stores the decryption keys within the TPM and the TPM only releases them if the platform has the expected state based on the measurements.

[20] HP Sure Start [130] and Intel Boot Guard [230] are two existing implementations of this component.

are OS-independent and platform-specific. We differentiate two types of runtime services: normal and privileged.

NORMAL RUNTIME SERVICES We call these services *normal*, because they have the same privileges as the kernel of the OS: they are executed on the same privilege level (ring 0) and on the same operating mode. It means that in terms of security, if an attacker managed to compromise the OS, it can also compromise these services. Here is a non-exhaustive list of these runtime services:

- set the system wake up alarm clock time;
- reset the system;
- get or set non-secure variables that are passed between OSs, boot loaders, or other UEFI applications on the platform.

PRIVILEGED RUNTIME SERVICES To handle critical services, the BIOS exposes runtime services that can transition into a more privileged and restricted environment. This environment is more privileged, because in addition to being in ring 0, it is also in a specific operating mode of the CPU called the System Management Mode (SMM) [139].

2.1.4 *System Management Mode*

SMM is a highly privileged operating mode of x86 processors.[21] It is not intended for general-purpose software, but only for the firmware. It provides the ability to implement critical OS-independent and platform-specific functions [139, 291]. Here is a non-exhaustive list of such functions:

- shutdown the system if it is in a danger of overheating;
- protect the access to the BIOS flash;
- BIOS update;
- get or set UEFI secure boot variables.

The peculiarity of the SMM is that it provides a separate execution environment, invisible to the OS—similar to a Trusted Execution Environment (TEE).[22] In terms of security for the platform, only the code executed in SMM can modify the firmware stored into flash to prevent any unintentional modifications by the OS, and in particular, to prevent malware—executing with kernel privileges—from overwriting the firmware and becoming persistent.

[21] x86 processors have 6 operating modes: unreal mode, real mode, protected mode, virtual 8086 mode, long mode, and SMM.

[22] Our focus is on Intel x86 platforms, but ARM platforms provide a TEE—known as the ARM TrustZone secure world [13]—that offers a similar environment than SMM.

SYSTEM MANAGEMENT RAM The code and data used in SMM are stored in a hardware-protected memory region only accessible in SMM, as illustrated in Figure 2.4, called System Management RAM (SMRAM). Access to the SMRAM depends on the configuration of the memory controller, done by the BIOS during the boot process. Once all the necessary code and data have been loaded in SMRAM by the BIOS, the firmware locks the memory region[23] by setting the `D_OPEN` bit to 0 and `D_LCK` bit to 1—both from the System Management RAM Control (SMRAMC) register. When protected, or locked, the SMRAM can only be accessed by code running in SMM, thus preventing an OS from accessing it and even preventing any Direct Memory Access (DMA) from other components of the platform. The memory controller can determine if a memory access is done in SMM thanks to the `SMIACT#` signal asserted by the processor.

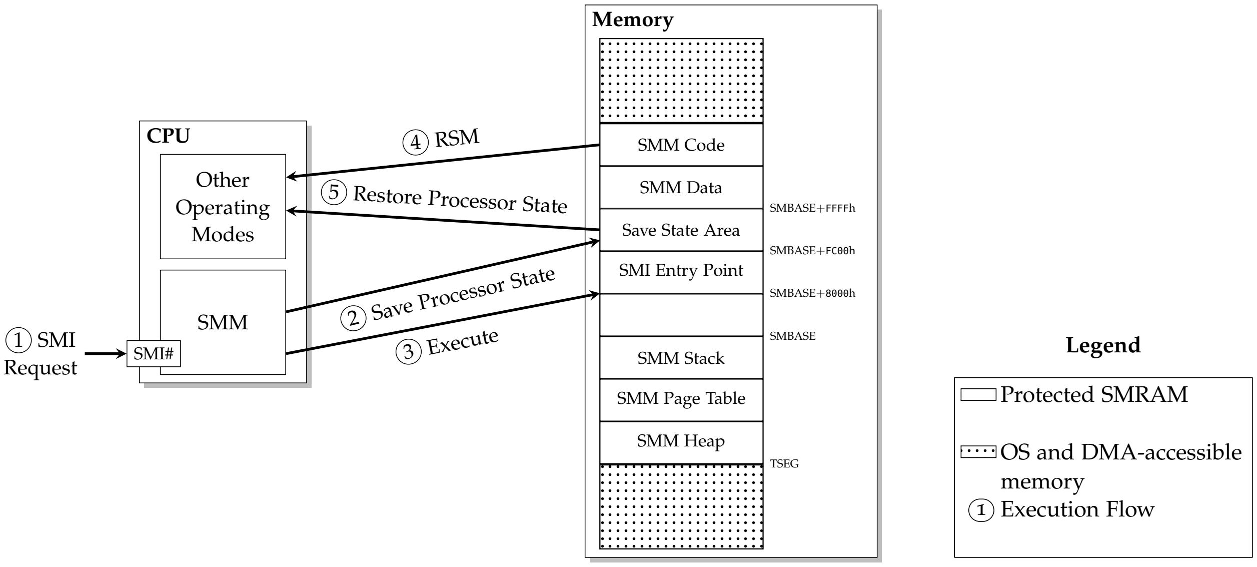

Memory
CPU
Other Operating Modes
SMM
SMI#
① SMI Request
② Save Processor State
③ Execute
④ RSM
⑤ Restore Processor State
SMM Code
SMM Data
Save State Area
SMI Entry Point
SMM Stack
SMM Page Table
SMM Heap
SMBASE+FFFFh
SMBASE+FC00h
SMBASE+8000h
SMBASE
TSEG
Legend
Protected SMRAM
OS and DMA-accessible memory
① Execution Flow

Figure 2.4: SMM execution flow and SMRAM memory layout

SYSTEM MANAGEMENT INTERRUPTS SMM is entered by generating a System Management Interrupt (SMI), as illustrated in Figure 2.4, which is a hardware interrupt. Software can also make the hardware trigger an SMI.[24] The BIOS sets up the SMRAM with code that processes the SMIs—also known as SMI handlers. The particularity of an SMI is that it makes all the CPU cores enter SMM. It is non-maskable and non-reentrant (i.e., when a handler processes an SMI it cannot handle other SMIs). In terms of security, SMI handlers are the main attack surface, since they can process attacker's-controlled data.

SAVE STATE AREA AND SMBASE Upon entry to SMM, after receiving an SMI, the CPU saves its state (i.e., registers and special data related to SMM) into a dedicated part of the SMRAM called the *save*

[23] In UEFI-compliant firmware, the SMM is usually set up and locked early during DXE phase—prior to any third-party code execution such as option ROMs.

[24] For example, writing a byte to the port `0xB2`—also known as `APM_CNT`—will trigger an SMI [132].

state area. When the CPU executes the `rsm` instruction, it exits the SMM and restores its state from this area. The save state area contains the value of various CPU registers. For example, an important CPU register in terms of security, the SMBASE, determines the entry point of an SMI in memory (`SMBASE + 8000h`).[25] SMBASE has a default value (`3000h`) and if the BIOS wants to relocate the SMRAM, it can modify the SMBASE value in the save state area while in SMM. When exiting, the processor will store the new value of the SMBASE and upon processing the next SMI, it will use this new value.

SMM PERFORMANCE CONSIDERATIONS Any time spent in SMM is time not available to the OS since it is paused. Therefore, for example, during that time it cannot process interrupts, network packets, audio streams, or run any task in general. Hence, SMIs must be processed as fast as possible to reduce the time the OS has to wait. The Intel BIOS Test Suite (BITS) defined the maximum acceptable latency of an SMI to 150 μs [135]. Delgado and Karavanic [88] showed that, when the latency exceeds this threshold, it causes a degradation of performance (e.g., I/O throughput or CPU time) or user experience (e.g., severe drop in frame rates in game engines).

2.1.5 *Attacks Against the SMM*

As explained in Section 2.1.4, SMM is one of the most privileged operating mode of x86 processors. It is the ideal target for attackers that want to control the OS without being detected, to tamper with the content of the flash containing the BIOS (e.g., for persistence), or in general to bypass various security mechanisms on the platform due to its unrestricted access.

To the best of our knowledge, Duflot et al. [97, 98] in 2006 were the first to consider the SMM as a way to circumvent OS security mechanisms. This idea motivated them to look at how one could have arbitrary code execution in SMM (e.g., how an attacker could inject code in SMRAM and then trigger an SMI to execute that code).

Their initial work, however, assumed that the platform was misconfigured, and that the `D_LCK` bit from the SMRAMC register—that determines if the configuration of the SMRAM is locked—was set to 0. It allowed them to set the `D_OPEN` bit to 1, thus allowing the modification of the SMRAM content from code not executing in SMM. In practice, if `D_LCK` is correctly set by the BIOS "there is no way to access SMRAM while in protected mode" [97].

Three years later two research teams [99, 283], including Duflot et al., independently discovered an attack that gave arbitrary code execution in SMM despite the `D_LCK` bit set to 1. Thus, contradicting the initial assumption that the `D_LCK` bit was sufficient to protect the

[25] Since each CPU core might have a different state, the BIOS set up the SMBASE of each core so that their save state area does not overlap with the others, thus having their own dedicated save state area and their own SMBASE.

SMRAM. Since then, additional attacks and vulnerabilities [23, 33, 64, 99, 100, 211, 212, 223, 282–284] were publicly disclosed.[26]

We now provide a summary of the main vulnerabilities that were discovered, how to exploit them, and we describe the existing countermeasures. We only focus on those vulnerabilities that can be exploited to take control of the SMM at runtime, since our work focuses in that area and existing boot time integrity solutions[27] try to address the other attacks. For example, we do not discuss attacks that exploit a vulnerability—involving the SMM—to overwrite the BIOS flash [146], that require a reboot or to resume from sleep mode (S3 resume) [282].

SMRAM CACHE POISONING ATTACK Two research teams [99, 283] independently discovered cache poisoning attacks in SMM. Since the cache is shared between all the operating modes of the CPU, the attack consists in marking the SMRAM region to be cacheable with a write-back strategy. Then, the attacker stores in the cache malicious instructions. After that, once an SMI is triggered, the processor fetches the instructions from the cache. Thus, the processor executes the malicious instructions of the attacker instead of the legitimate code stored in SMRAM. The solution proposed by Intel was to modify the behavior of the cache depending on whether the CPU is in SMM or not. They introduced a new special-purpose register called System Management Range Register (SMRR). This register can only be modified in SMM, it specifies the SMRAM range, and it decides the cache strategy of the SMRAM. If the processor is not in SMM, it considers the SMRAM as uncacheable, it ignores the write accesses, and it returns a fixed value for read accesses. Recent platforms deploying this solution should not be vulnerable to this attack anymore.

INSECURE CALL OUTSIDE OF SMRAM Multiple BIOS implementations [64, 284] provided SMM code that calls (or jumps to) code segments outside of the SMRAM. An attacker with kernel-level privileges can easily modify this code, thus providing the attacker with arbitrary code execution in SMM. These vulnerabilities have been fixed by forbidding the processor to execute instructions located outside of the SMRAM while in SMM.[28] If such a case happens, a machine-check exception—an unrecoverable error—is triggered.

INSECURE INDIRECT CALL POINTER HANDLING Other vulnerabilities due to indirect calls [212, 284] (i.e., a call to a function where the address of the function is known at runtime) allow attackers to perform code-reuse attacks against the SMM code. In 2009, Wojtczuk and Tereshkin [284] discovered that some SMI handlers perform an indirect call with the function address stored in a memory region outside of the SMRAM. It allows an attacker with kernel-level privileges to run arbitrary code by modifying the value of the pointer to

[26] While initially thought as hypothetical, recent discoveries—such as the exploitation of SMM vulnerabilities by APT28 [226]—show that attackers targeting the SMM and the BIOS are a real threat nowadays.

[27] See Section 2.1.2.

[28] One has to set the `SMM_Code_Chk_En` register to 1 [138].

a location controlled by the attacker. Oleksiuk [212] found a similar vulnerability in 2016 with a more recent platform where the pointer of an indirect call was inside a data structure controlled by the attacker.

In general, such attacks are usually prevented by patching these vulnerabilities. However, recent vulnerabilities might not have been discovered yet, or the patches might take some time to be applied by the users.

INSECURE POINTER HANDLING: ARBITRARY READ AND WRITE Some SMI handlers rely on data provided by the OS (i.e., controlled by the attacker). If they do not sanitize such data, the attacker can influence the behavior of the SMM.

For example, pointer vulnerabilities in an SMI handler can lead to arbitrary write into SMRAM [23, 211, 223]. It can occur because the SMI handler writes data into a buffer located at an address controlled by the attacker. For example, such an address can be provided thanks to a register that could have been modified by the attacker. Bulygin et al. [33] demonstrated a similar attack by modifying the Base Address Registers (BARs) used to communicate with PCI devices. Such an attack can be used to modify the SMBASE value stored in the save state area of the SMRAM. Upon entry of the next SMI, the attacker has control over the SMM. Similarly, SMI handlers can have vulnerabilities giving arbitrary read into SMRAM to the attacker.

It is the responsibility of SMI handlers to verify that the data given or controlled by the OS is valid. For example, they should check that the address of the communication buffer is not pointing into the SMRAM, and that the BARs point to valid addresses (i.e., not in RAM or SMRAM). Hence, since the BIOS developers bear this responsibility, recent platforms might be vulnerable to such vulnerabilities. Either because the developers forgot or were not aware of the need to sanitize the pointers.

Attack or Vulnerability	Countermeasure	Requires Platform Configuration	Requires Developer Discipline
SMRAM writable while not in SMM	Set the `D_OPEN` bit to 0 and the `D_LCK` bit to 1	●	❍
SMRAM cache poisoning attack [99, 283]	Configure the SMRR [139]	●	❍
Insecure call outside of SMRAM [64, 284]	Set the `SMM_Code_Chk_En` bit to 1 [138]	●	❍
Insecure indirect call pointer handling [212, 284]	Validate pointers	❍	●
Insecure pointer handling: arbitrary write [23, 211, 223]	Validate pointers	❍	●
Insecure pointer handling: arbitrary read [23]	Validate pointers	❍	●

Table 2.1: Summary of the main attacks targeting and vulnerabilities affecting the SMM with their countermeasures

CONCLUDING REMARKS When looking at the vulnerabilities and their related countermeasures, summarized in Table 2.1, we noticed that there are two main types: those that require a correct configuration of the platform and those that require a constant discipline from the developer to not introduce the vulnerability (e.g., by following coding practices or by using analysis tools). The former only needs to be found and fixed once for the BIOS.[29] The latter, however, is due to how lax the C language is. While secure coding practices and static or dynamic analysis of the code can reduce the number of memory corruption bugs, they can still be introduced. It requires constant discipline from the developers, analyzing and testing of the code.

2.2 OPERATING SYSTEMS

After initializing the platform, the BIOS lets an OS take over or executes a boot loader that subsequently executes an OS. The goal of the OS is to manage the resources of the computer to allocate them to various tasks or applications. The OS provides a set of functions that are used by these tasks to access and use the resources. In terms of security, the main goal of the OS is to isolate the tasks from each other and to ensure that the tasks respect certain security policies (e.g., the OS handles the access control for the file system).

2.2.1 *Kernel*

The most well-known component of an OS is its core component: the kernel. It provides all the basic code to control the hardware and the resources. For example, it is responsible for the management of the address space and the scheduling of the processes that share the CPU time. Depending on its type, however, the size of the kernel, its functions, and its complexity can vary.

We differentiate three common main types, as illustrated in Figure 2.5, depending on the security domains they provide: no isolation, monolithic kernel, microkernel.[30]

NO ISOLATION There is no isolation between the kernel and the applications. If an attacker compromised an application, it can also infect the rest of the OS without difficulty. Such a design is used for some embedded and resource constrained systems.

MONOLITHIC KERNEL The kernel comprises most of the OS components, but it is isolated from the applications. The applications are executed in a lower privilege level—called user mode (ring 3)—while the kernel is executed in a higher privilege level—called kernel mode (ring 0).[31] The applications are also isolated from the others by the kernel. It means that it is more difficult for attackers to compromise the OS, but this architecture is still

[29] Tools such as chipsec [61] can be used to detect misconfigured platforms. Moreover, there is work on using formal methods to detect, at design-time, inconsistencies in hardware specifications that can help to bypass security mechanisms of the SMM [173] (e.g., SMRAM cache poisoning attacks).

[30] Other more specialized or related kernel architectures exist, such as unikernels or hybrid kernels, but their isolation is similar to the ones presented here.

[31] The terms user space (or userland) and kernel space are commonly used to refer to memory areas, such as code, reserved respectively for user mode or kernel mode.

efficient in terms of performance. If an attacker compromises any of the kernel components (e.g., device drivers), however, the rest of the kernel is compromised as well.[32]

Microkernel The kernel only comprises the minimum requirements to schedule tasks, to isolate them, and for them to perform Inter-Process Communication (IPC). The drivers, however, are executed in user mode [256]. Due to such a design, if a driver is compromised it does not directly lead to a complete compromise of the OS. While the increased isolation of the OS components helps in terms of security, it incurs an overhead due to each component needing to perform IPC when interacting with other components.

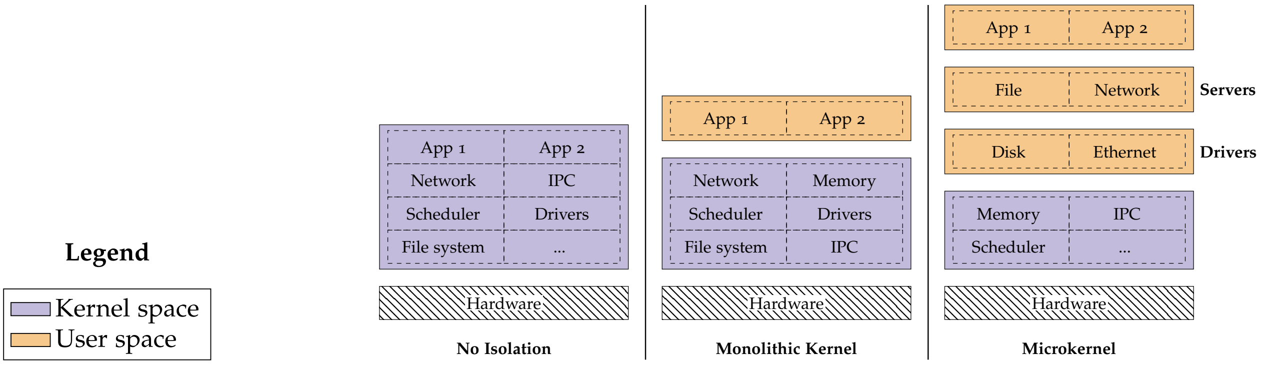

Legend
Kernel space
User space
App 1
App 2
Network
IPC
Scheduler
Drivers
File system
...
Hardware
No Isolation
App 1
App 2
Network
Memory
Scheduler
Drivers
File system
IPC
Hardware
Monolithic Kernel
App 1
App 2
File
Network
Servers
Disk
Ethernet
Drivers
Memory
IPC
Scheduler
...
Hardware
Microkernel

Figure 2.5: Difference of isolation between main kernel types

While there are still debates in the community regarding which architecture should be preferred, current commodity OSs that dominate the market share (e.g., Android, Linux, or Windows) are based on a monolithic kernel.[33] Hence, in this dissertation, and most importantly in Part II, we assume that the OS follows this design.

When the kernel initializes itself, it sets up the various structures it needs to manage the memory and the tasks, it sets up the interrupt handlers, and it loads drivers needed to control various hardware components on the platform (e.g., a Wi-Fi adapter or a webcam).

The kernel also sets up a communication interface between the user mode and the kernel mode—known as system calls—so that the processes executed in user mode can perform privileged operations. Modern OSs configure the platform so that programs executing in user mode cannot interact directly with the hardware (e.g., to write files on storage devices or to use network devices), only code executing in kernel mode can. Such an interface ensure that programs use a safe implementation to communicate with the hardware.[34] Moreover, in terms of security, it allows the kernel to implement security policies by checking that the calling process has the right to perform such an action.

[32] Unfortunately, experience shows us that most of the kernel vulnerabilities originate from device drivers (third-party code) [48].

[33] Some OSs (e.g., Windows or Mac OS X) are based on a hybrid kernel that attempts to share aspects of both a monolithic and a microkernel, but in terms of security most drivers are still executed in kernel mode.

[34] In practice, programs rarely directly use the system calls, but they rely on a library that provides wrapper functions for the system calls (e.g., glibc on Linux).

System calls are initiated by interrupts[35] that puts the CPU in ring 0 and passes the execution control to the kernel. The kernel then executes the appropriate system call handler that exits by executing an instruction that puts back the CPU in ring 3 before returning the control to the calling program.

2.2.2 *User Space Initialization, Service Manager, Login*

After the kernel initialized, it launches one process that is responsible for the initialization of the user space. This process creates the environment expected by the user mode processes (e.g., it mounts file systems or starts system services) and can start multiple initialization processes.[36] On Windows, the first user mode process is the Session Manager Subsystem (smss.exe) which then starts subsequent processes such as the Windows Initialization Process (wininit.exe) and the Service Control Manager (SCM) (services.exe) [231, Chapter 13]. On Linux, the initialization process—also known as *init* or PID 1—depends on the distribution.

An important component of an OS is its service manager that is started during the initialization phase of the user space. On Windows, it is the SCM that is responsible for starting services, some device drivers, and interacting with their processes. On Linux, at the time of writing, major distributions rely on systemd [253]—a system and service manager—to act as their initialization process, and to start and maintain their user space services.

Finally, the login components are initialized (winlogon.exe on Windows and systemd-logind on many Linux distributions). At the end of the system startup, the login prompt appears (e.g., logonui.exe for Windows, and getty or GDM for Linux distributions).

2.2.3 *Isolation Primitives: The Case of the Linux Kernel*

Without support from the kernel and the OS in general, applications would share the same global state (i.e., they would share the same memory space, and they would all have access to the same data on the storage devices). In terms of security, however, it means that if a process is compromised, the attacker also has access to this same global state.

The principle of least privilege states that: “Every program and every user of the system should operate using the least set of privileges necessary to complete the job.” [233, p. 1282] The reasoning is that the less privileges a program has, the less likely it is to compromise the system stability or security.

Even if determining the least set of privileges is a difficult and error-prone process, this principle of least privilege is what drives the development and adoption of many security features provided by OSs.

[35] On Intel x86 processors, the instructions `int 80h` or, more recently, `sysenter` can be executed to trigger such an interrupt [137].

[36] This process is therefore a highly privileged process of the OS since it is executed early and can configure the privileges of the other processes.

Among the common features are the memory address space isolation between processes (by configuring the MMU), access control lists—such as Mandatory Access Control (MAC)[37] or Discretionary Access Control (DAC)—or privilege levels.[38] Modern OSs also offer more advanced primitives to isolate a process, reduce its set of privileges or its quotas to access some resources. Here, we focus on Linux-based systems as a use case, and we give a non-exhaustive list of such primitives.

CONTROL GROUPS A control group [125], or cgroup, is a hierarchical set of processes bound to a set of limits, quotas, or parameters, affecting the availability of system resources (e.g., CPU or memory).[39] For instance, one can limit the memory usage of a set of processes to 200 MB. It is also possible to freeze the processes of a cgroup (i.e., to remove them from the scheduling queue).

NAMESPACES Linux has the notion of namespaces [149] where system resources can be seen differently by processes depending on the namespace they belong to.[40] Hence, one can isolate the access to some resources by creating a namespace in that regard. Linux has namespaces for the mount points (file system), process identifiers, network, IPC, user identifiers, cgroups, and the system identifiers (hostname). For example, one can make a process see the `/home` directory (normally containing the user files) as empty by creating a new mount namespace and binding `/home` on an empty directory. The other processes, however, all see the normal `/home` directory.

CAPABILITIES Linux can divide the privileges of the superuser (or *root*) in multiple distinct chunks called capabilities—also known as POSIX capabilities [236].[41] In particular, with ambient capabilities, a process can retain only the specific capabilities it needs (e.g., clock synchronization services would require `CAP_SYS_TIME` to set the system clock) while being a non-privileged user (not root). Such POSIX capabilities, however, are limited [148] since some of them (e.g., `CAP_SYS_ADMIN`) are coarse-grained and some can even lead to gaining back full root privileges using multiple transitions [251].

SYSTEM CALL FILTERS Linux offers the ability to filter the system calls made by processes with seccomp [104]. Whether it is a blacklist or whitelist filtering policy, the process that does not follow the policy can be terminated or the system call can return an error. Filtering system calls reduces the overall privileges of a process and it also reduces the attack surface of the kernel for attackers that try to exploit kernel vulnerabilities to escalate privileges and bypass kernel restrictions.[42] For example, we can use this feature to block a system call (e.g., `socket`), a system

[37] On Linux, one can implement a MAC policy using e.g., SELinux [210] or AppArmor [40].

[38] As a more thorough overview and analysis on OS security designs, principles, models, isolation primitives, or hardening, we recommend the report from Bos [30].

[39] In addition to applying resource constraints on services, systemd also relies on cgroups to keep track of the processes they create.

[40] Inspired from the namespaces feature in the Plan 9 [221] distributed OS.

[41] Capabilities and capability-based system were formalized by Dennis and Van Horn [91] in 1966. Linux, however, does not implement their model, and the naming "capabilities" should not be confused with their capability-based system, since Linux capabilities do not have the notion of objects.

[42] For instance, kernel vulnerabilities are often exploited by using specific system calls. Either because the vulnerability is in a system call [69, 80] or because some system calls are needed to set up the exploit [79, 82].

call with specific parameters (e.g., `socket` with the `AF_PACKET` domain), or a set of system calls.

NETWORK PACKET FILTERS Instead of entirely disabling access to the network (with a network namespace) of a process, Linux also provides the ability to filter network packets per-process [111]. It thus differs from a system-wide firewall that would apply rules for the whole system. These filters can be attached to a cgroup in order to affect a set of processes. For example, one can whitelist the port 80 and 443 for the processes of an HTTP server, or whitelist the IP addresses used by a clock synchronization service.

Nowadays, services can be configured with the principle of least privilege in mind by using these features, thus reducing the impact of an intrusion. For example, systemd relies on some of those solutions to allow administrators or developers to set the least set of privileges required to run their services [92].[43] Similarly, Docker—a service used to manage containers—also uses some of them to implement their technology [209] that can host single services.[44]

Finally, while we focused on Linux, other OSs can provide similar primitives. For example, on Windows, one could use the Integrity Mechanism [196], Restricted Tokens [195], or the Job Objects [193].

2.2.4 Limitations of OS Security Principles

As argued in the introduction,[45] we should assume that given time, an intrusion will occur. There are various technical reasons why an intrusion can occur. For instance, it might be a memory corruption bug[46] (e.g., a use-after-free or an integer overflow) or a misconfiguration (e.g., a known default password has not been changed). Memory protections (e.g., read-only code using the MMU), hardening or mitigation features (e.g., stack canaries or address space layout randomization), safer programming languages (e.g., Rust or Java), or access controls, help to reduce the number of successful attacks on OSs and their services, but eventually a vulnerability is found. Therefore, we should assume that services (e.g., an HTTP server, a clock synchronization service, or a database) will get compromised.

When a service gets compromised, we have two possible cases. First, if the service has not been configured with the principle of least privilege in mind—thus having unnecessarily highly privileged components—attackers can exploit the privileges of the service to further compromise the rest of the system, or simply achieve their goal (e.g., compromise data availability or data theft). Second, if the service has been fully configured with the principle of least privilege in mind,[47] the attackers would still be able to exploit the privileges

[43] Unfortunately, at the moment, only a small set of these options are used in practice by developers and administrators.

[44] Docker, however, did not initially use these features for security, but to easily package and ship applications.

[45] See Section 1.1.1.

[46] If you need more information about how attackers can exploit memory corruption bugs, we recommend the study from Szekeres et al. [254] (in addition to the papers that they reference).

[47] In practice, it is not binary but more a spectrum of services that restrict to a certain extent their privileges.

left to the service. For example, if a service has access to some files, the attacker also has access to them, and can steal or corrupt them.

In both cases, it shows the limitation of relying only on the isolation of a service to withstand an intrusion. In Part II, we describe one of our contributions that addresses this issue by allowing a compromised service to survive an intrusion. We first restore the compromised service to a previous state that we assume safe, then we apply responses that prevent attackers from achieving their goals—a fact that is often not possible with the initial preventive isolation since it must maintain the privileges needed by the service.

2.3 CONCLUSION

In this chapter, we described the software components involved during the boot process with a security perspective. First, the BIOS that plays a key role in initializing the platform. We also described the mechanisms that ensure the integrity of the boot process and we discussed the privileged runtime BIOS code running in SMM along with the threats that it faces. In Part III, we use the services executed in SMM as a use case, and we rely on the background described here to develop threat models and build detection models.

Second, we described the OS that manages the resources and the applications, and that enforces security policies on them. We described succinctly the role of the user space initialization processes and of the service manager. Moreover, we detailed the isolation primitives that modern OSs provide by mentioning multiple features from the Linux kernel. In Part II, we work at the OS and application abstraction layers where we rely on the service manager and the isolation primitives provided by the OS to orchestrate and implement our approach.

Finally, in both cases, we mentioned that we can expect to be compromised. Thus, with such an assumption, we should not focus on the specific vulnerabilities that might be exploited, since we do not know in advance their existence or which ones will be targeted. Instead, we can rely on specific behaviors that are exhibited by an intrusion. We can use the behaviors exhibited by the attacker when exploiting a vulnerability to detect an intrusion. We can also use the malicious behaviors exhibited by the intrusion to know how to respond and to survive it. We will explain in Part II and Part III how we make use of these behaviors to improve the detection capability and survivability of the platform.

Before discussing our contributions, we end the first part of this dissertation with a review of state-of-the-art solutions related to our work.

3

STATE OF THE ART: DETECTION AND SURVIVABILITY

> *Progress was often achieved by a "criticism from the past"*
>
> — **Paul Feyerabend**
> Philosopher of science

The high-level goal of this work is to study how to improve the resiliency of a computing platform at its various abstraction layers. Preventive security mechanisms are not sufficient to ensure the security of a system. In our work, we assume that the system is under attack and that the attackers have either compromised the system or will. The ability of computer systems to detect intrusions and to be resilient against them, however, has been heavily studied since the 1980s.

In this chapter, we present the state of the art by first describing the definitions and concepts related to our work. Second, we detail the existing work on intrusion detection for low-level components (e.g., the kernel of an OS or the SMM code). Third, we review existing work on intrusion recovery and intrusion response to help achieve intrusion survivability for commodity OSs. Finally, we conclude on the gaps that we identified in the literature, and we present what motivated our contributions.

3.1 TERMS AND CONCEPTS

In this section, we define terms that we use in this dissertation. Then, we describe the concepts in which our work depends: intrusion detection and intrusion survivability.

3.1.1 *Common Terms*

While we assume that the reader is knowledgeable in the field of information security, we use various terms related to this field throughout this dissertation that can bring confusion: vulnerability, attack, intrusion, threat, or risk. We first give an overview of these terms and how they are related. Then, we quote more standard definitions of these terms from the Internet Engineering Task Force (IETF) [246].

A vulnerability is any weakness (e.g., a bug or a misconfiguration) in a system that an attacker can exploit to violate a security policy. When attackers *attempt* to exploit vulnerabilities, it is called an attack. When attackers *successfully* compromise a system or violate a security policy, it is called an intrusion. A threat is any entity or event that has the potential to exploit a vulnerability to inflict harm on any resources (e.g., a computer or data) of the organization. Security mechanisms are often designed to protect against certain threats, but not all. Finally, a risk is the potential that a given threat compromise the system by exploiting a vulnerability and inflicting harm on resources of the organization. A risk is often expressed as the product of the likelihood of occurrence of a particular threat times the severity of the impact if it were to occur.

The IETF defined these terms in the RFC 4949 as follows:

VULNERABILITY "A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy." [246, p. 333]

ATTACK "An intentional act by which an entity attempts to evade security services and violate the security policy of a system. That is, an actual assault on system security that derives from an intelligent threat." [246, p. 22]

THREAT "A potential for violation of security, which exists when there is an entity, circumstance, capability, action, or event that could cause harm." [246, p. 304]

RISK "An expectation of loss expressed as the probability that a particular threat will exploit a particular vulnerability with a particular harmful result." [246, p. 251]

INTRUSION "A security event, or a combination of multiple security events, that constitutes a security incident in which an intruder gains, or attempts to gain, access to a system or system resource without having authorization to do so." [246, p. 165]

Except for the last definition (an intrusion), our definitions are similar, and this manuscript assumes these definitions. In the case of an intrusion, however, we do not consider "attempts to gain access" as an intrusion, we only consider a successful compromise of a system as an intrusion. Indeed, we use the word attack to refer to "attempts to gain access" in order to differentiate both an attempt and a successful compromise, since the impact of the latter is more important to the security of the system than the former.

3.1.2 *Intrusion Detection*

In the information security field, intrusion detection dates back to the 1980s with the work of Anderson [7] and Denning [90]. While initially thought as a way to automatically detect intrusions on a system based on audit trails, it evolved since then with many new approaches proposed and studied.

In general, intrusion detection refers to a set of techniques, or approaches, that monitors information systems in order to automatically detects intrusions, or any malicious activity. An implementation of an intrusion detection approach is referred as an Intrusion Detection System (IDS).

In practice, many IDSs detect both attacks and intrusions. When an IDS wrongly considers a legitimate event as an intrusion, we call it a false positive.[48] Likewise, if it detects an attack that was not an actual intrusion, we call it a false positive. When it erroneously considers a malicious activity as a legitimate one, we call it a false negative. On the opposite side, we have the true positives that refer to malicious behaviors that have been correctly categorized by the IDS as intrusions. Finally, true negatives refer to legitimate events that have been correctly categorized as normal by the IDS.

Based on the taxonomy from Debar et al. [86, 87], we can classify an IDS depending on five characteristics:

DETECTION METHOD We can differentiate two types of IDSs based on the method they use or the kind of model they use to detect intrusions. First, knowledge-based IDSs—also known as signature-based or misuse-based—rely on information or patterns about attacks to detect their exploitation. Second, behavior-based IDSs—also known as anomaly-based—do the opposite by relying on a model of the expected legitimate behavior to detect any discrepancy between the reference model and what is observed.

BEHAVIOR ON DETECTION An IDS can either be passive (it only generates alerts) or active by responding to attacks or intrusions (proactively or reactively).

DETECTION PARADIGM An IDS can either analyze the states of a system or the transitions between the states (events). State-based IDSs analyze the current state of a system and determine whether it is in a secure state, or a non-secure state (compromised). Transition-based IDSs analyze the transitions and detect those that lead to a non-secure state.

LOCATION Depending on where the probe—the source of the information that the IDS relies on—is located, we can characterize an IDS as either host-based or network-based. For example, a network-based IDS could analyze network packets. An host-based IDS, on the other hand, could analyze the behavior of an application. In our work, we focus only on host-based IDSs.

USAGE FREQUENCY An IDS can either monitor continuously its target or it can run periodically checks or scans.

Even if this taxonomy originated in 1999 and 2000, its five high-level characteristics are still relevant to classify today's IDSs. We rely

[48] The first term "false" is about the characterization of the IDS. The IDS was wrong, it made a *false* statement.

on some of these characteristics to discuss various related work in subsequent sections.

Finally, IDSs are evaluated based on three measures:

ACCURACY The ability of the IDS to correctly generate an alert when an intrusion occurred (true positive) and to not generate an alert when no actual intrusion occurred (false positive).

COMPLETENESS The ability of the IDS to generate alerts for all intrusions (true positives).

PERFORMANCE The rate at which the IDS process events or the time to analyze a state.

For example, in general, knowledge-based IDSs have a high accuracy since they have a precise model of attacks, while anything that deviate from that model is considered legitimate. Their completeness, however, depends on how up-to-date their models are. On the other hand, we have behavior-based IDSs that usually have a high completeness, but a lower accuracy. Their advantage is that they can detect new intrusions or attacks, but one must build a reference model that limit the number of false positives.

3.1.3 *Intrusion Survivability and Related Concepts*

Our contributions in Part II, and the idea behind this thesis in general, are related to concepts such as security, dependability, resiliency, and survivability. As a reference, we recommend the work of Avizienis et al. [16] ("Basic Concepts and Taxonomy of Dependable and Secure Computing") that defined, compared, and summarized most of these concepts. They describe in detail the various attributes related to these concepts (e.g., reliability, integrity, or availability), the threats they face (e.g., faults or errors), and how to achieve them (e.g., fault prevention or fault tolerance).

The United States National Institute of Standards and Technology (NIST) and MITRE Corporation also publish standards and guidelines that list existing engineering techniques,[49] approaches, procedures, and provide frameworks to help organizations anticipate attacks, withstand intrusions, recover from them, and adapt to the current threat [229].

Here we give an overview of core concepts related to our work, and how we interpret them for the rest of this dissertation. For more details, the reader can consult the aforementioned references.

RESILIENCY AND DEPENDABILITY Laprie defined the concept of resilience as "the persistence of dependability when facing changes" [163], where dependability is "the ability of a system to avoid service failures that are more frequent or more severe than is acceptable" [16].

[49] They mention overlapping system engineering disciplines such as resilience engineering, intrusion-tolerant systems, survivability, or self-healing systems. The NIST and MITRE publications and especially their appendices [28, 229] summarize well these disciplines and how they overlap.

To achieve dependability, or other close concepts, four categories of techniques exist [16]:

- fault prevention (e.g., design choices in software development),
- fault tolerance (e.g., error detection or recovery),
- fault removal (e.g., static or dynamic analysis),
- or fault forecasting (e.g., probabilistic evaluation with fault models).

INTRUSION TOLERANCE The first problem exposed in the introduction of this dissertation—that preventive security is not sufficient—emerged from the concept of fault tolerance, and more specifically intrusion tolerance (i.e., an intentional malicious fault). We should assume that a system is vulnerable and that it will be compromised at some point, therefore we should design systems that can tolerate intrusions (to a certain extent). The goal of intrusion tolerance is to continue to deliver service and to maintain security properties despite intrusions. Intrusion survivability is related to this concept.

INTRUSION SURVIVABILITY Ellison et al. defined survivability as "the capability of a system to fulfill its mission, in a timely manner, in the presence of attacks, failures, or accidents" [105]. Avizienis et al. [16] suggested that—based on this definition—dependability and survivability were similar concepts. Knight et al. [158] weighed that survivability distance itself from dependability, since it should encompass the notion of degraded service, and a trade-off between the availability of some functions and the cost to maintain and provide them. Intuitively, Knight et al. defined survivability as the ability "to provide one or more alternate services (different, less dependable, or degraded) in a given operating environment" [158] essentially providing "a trade-off between functionality and resources" [158].

In the rest of this dissertation, we assume that survivability refers to such a degradation and trade-off. More specifically, since we care about *intrusion survivability*, we consider the trade-off to be between the availability of the different functionalities of a vulnerable service and the security risk associated to maintaining them.

Moreover, the definition of survivability has always be applied to networked systems or critical information systems. In our case, however, we apply the concept of intrusion survivability to commodity OSs (e.g., Linux-based distributions or Windows).

3.2 INTRUSION DETECTION FOR LOW-LEVEL COMPONENTS

Building a sound and reliable IDS for low-level components raise three main questions:

1. How to isolate?

   It pertains to the approaches that we can use to isolate the monitor from the component it is observing—its target.

2. How to monitor?

   It relates to the approaches that we can use to gather information about the behavior or the state of the target.

3. How to detect?

   It refers to the methods that we can use to determine if an intrusion occurred based on the information gathered.

The second and the third are respectively related to the detection paradigm and the detection method of the taxonomy from Debar et al. The first one, however, was not part of the taxonomy, but it is still an important question to answer. For the rest of the taxonomy, while the behavior on detection is an important aspect, our contribution related to the detection of intrusions targeting the SMM services focused only on the detection, hence in this section we only mention the detection aspect of the approaches from the state of the art. Moreover, as mentioned previously, since our goal is to monitor the SMM, our work only relates to host-based IDSs.

3.2.1 *Isolation of the Monitor*

In this document, we use the term monitor[50] to refer to the component of an IDS that observes the target and from this observation determines if an intrusion occurred.[51] The integrity of the monitor is crucial, because it is a trusted component that we rely on to detect the intrusions. The monitor could also be used to start remediation strategies. If the attacker compromises the monitor, we cannot trust the detection nor the remediation anymore. Therefore, we first discuss *how to isolate* the monitor from the target. We distinguish two types of monitor regarding their isolation from their target: inlined monitors and external monitors.

INLINED MONITOR Inlined monitors are executed on the same layer of privilege as their target. For instance, they add the full logic of the monitoring by instrumenting the code to perform runtime checks [1, 107, 264]. In such a case, the monitor corresponds to all the instructions added in the code. The main drawbacks of inlined monitors are their performance impact and their limited isolation. Since all the logic is integrated in the target, it can incur a significant performance overhead if the logic is complex. Moreover, the target and the monitor share the same memory space and the same execution environment. It provides a wide attack surface for the attacker. If the target is compromised, the monitor can be compromised as well.

[50] Depending on the field (e.g., runtime monitoring, network security, or system security), the words monitor and IDS can refer to the same concept. In the hardware community, we observed that the term monitor is often used to refer to an IDS that executes on a co-processor or similar. In this document, we also often use this term.

[51] Anderson [6] described the concept of a reference monitor that observes software execution at runtime. Schneider [234] formalized the concept of enforcing security policies by monitoring system execution.

EXTERNAL MONITOR External monitors are either one layer beneath the target (e.g., a hypervisor), thus more privileged, or are fully isolated from the target they are monitoring (e.g., a co-processor). In both cases, solutions rely on hardware features to isolate the monitor. In the case of the SMM, recent work from Intel provides us with the ability to run SMM code in a virtualized environment [288]. An SMI Transfer Monitor (STM) acts as a hypervisor that supervises SMI handlers to ensure that they only access the resources they need. While the STM has been designed only to limit the impact of a potentially compromised or malicious SMI handler, it could also be used to monitor the behavior of SMI handlers to detect intrusions. Instead of a hypervisor, Bulygin and Samyde [34] proposed to use an embedded microcontroller to monitor the code and data of the SMM. Such a co-processor offers more flexibility to the IDS than a hypervisor solution, since it can combine different monitoring approaches and it can be asynchronous. Indeed, hypervisors-based approaches only rely on specific events (e.g., hypervisor calls such as Intel VT-x instructions) to trigger their monitoring.[52] It makes it more difficult to implement some approaches (e.g., taking periodic snapshots of the memory of the target—an approach that we describe in the next section). Finally, a main issue with external monitors is the more isolated they are, the less information they have on their target. This issue—known as the semantic gap [49, 142]—means that the monitor might only have a partial view of the state of its target.[53] It can potentially help an attacker to bypass the IDS [143].

Coudray et al. [74] mix both inline and external approaches, since they instrument a program (the target) to send information to another program (the external monitor) about its behavior. The instrumentation is similar to inline-based approaches, but the instrumentation does not contain the full logic, it is only used to send information. The benefit of such an approach (mixing both worlds) is that the instrumentation is minimal in comparison to inline-based approaches [1, 264, 276] that include the monitoring and full detection logic with their instrumentation. By externalizing the detection logic, and by only keeping the communication of behavioral information, the attacker cannot leak secret or corrupt data structures used for the detection, since they are isolated. Their approach, however, targets applications of an OS and only externalize the monitor in another program of the system. Moreover, they do not evaluate the performance impact of their solution only the security aspect. Unfortunately, the performance of approaches using external monitors is important aspect due to the extra steps needed to communicate information to the monitor.

In our work (Part III), we use a similar hybrid approach, since we externalize as much as possible the IDS from our target. We isolate our monitor and the detection logic thanks to a dedicated co-processor. However, since we need to have information about our target, we

[52] For more details about hypervisor-based monitoring approaches in general and their challenges, see the survey from Bauman et al. [22].

[53] Jain et al. [142] explored in depth the challenges due to the semantic gap for a hypervisor and its guests, and the existing solutions to bridge this gap.

also instrument the code of the target (SMM code) to send information to our external monitor—bridging the semantic gap introduced by the isolation. We send this information through a low-latency communication channel to minimize the performance impact of the communication.

3.2.2 *Hardware-Based Monitoring*

We now discuss approaches that focus on *how to monitor* low-level components—such as firmware and kernels—-using a hardware-based approach. To the best of our knowledge, the only commercially available technology that offers SMM integrity monitoring is HP Sure Start [128–130]. It uses the chipset and the CPU to monitor SMM code integrity and relies on additional hardware to take actions per a predefined policy. The details of its implementation, however, are not public. Thus, we cannot compare it in detail to other approaches in the literature.

The approaches presented here are not necessarily focused on monitoring the SMM, but they could be adapted to that aim. Since our work focuses on the SMM—the runtime part of the BIOS—we only discuss approaches that provide runtime intrusion detection, and do not consider, for example, approaches that provide boot time integrity.[54] We distinguish two different types of approaches: snapshot-based approaches are presented in Section 3.2.2.1 and event-based approaches in Section 3.2.2.2.

3.2.2.1 *Snapshot-Based Monitoring*

The first approach consists in taking periodic snapshots of all or any part of the target state and then to analyze these snapshots to detect intrusions. In the taxonomy of Debar et al. [87], it relates to the state-based detection paradigm.

To the best of our knowledge, Zhang et al. [297] were the first to propose a co-processor for intrusion detection using a snapshot-based approach. However, they did not implement their design. Notable implementations of such approach are Copilot [220], DeepWatch [34], and HyperSentry [18].

Copilot is a kernel integrity monitor using a co-processor on a PCI card to take periodic snapshots of the main memory using DMA requests.[55] The authors also described how to write rules describing the relationships between kernel objects to detect the presence of kernel rootkits [219]. Copilot, however, cannot monitor the SMM since it does not have access to SMRAM. Even if it was shown that Copilot was effective against 12 common kernel rootkits at that time, it has two main limitations. First, since the monitor runs an out of context measurement there is a semantic gap between the monitor and the host. For example, it is difficult to have knowledge over virtual to

[54] See Section 2.1.2.

[55] DMA allows peripheral devices to directly access the main memory without using the CPU during the transfer.

physical memory mappings. Moreover, race conditions between a kernel process and the DMA requests from the monitor can occur, which create inconsistent states of kernel data structures. Second, Rutkowska [232] shows the practicability of a data substitution attack by modifying the memory controller. An attacker can remain undetected by redirecting memory accesses from the PCI card to another location where is located a copy of the original non-tampered kernel.

DeepWatch uses a similar approach to Copilot, but the monitor runs on an embedded core in the chipset, which allows the monitoring of the SMM. HyperSentry leverages the SMM to perform measurements giving access to the CPU-context, but it impedes its ability to monitor the SMM itself.

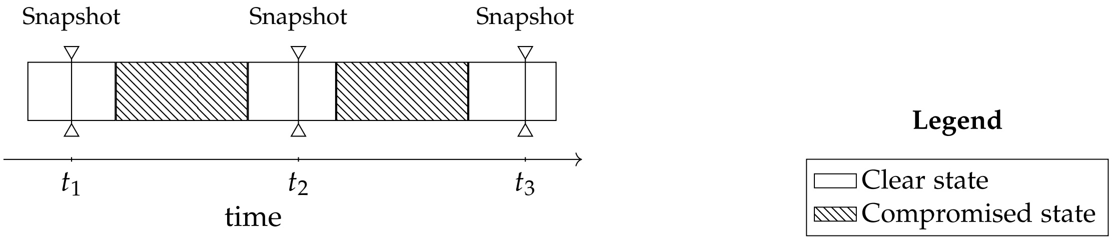


Figure 3.1: Timeline of a transient attack where the attacker clears any trace of its presence on the system, before each snapshot at t_1, t_2 and t_3

Nevertheless, a major limitation of snapshot-based approaches that takes periodic snapshots, is that attackers could erase their traces before each snapshot, or leave a minimum amount of traces making the detection of its presence unlikely, as illustrated in Figure 3.1. Such attacks, that do not make persistent changes to the system, are called transient attacks [18, 202]. In general, snapshot-based solutions suffer from this limitation. A solution against this attack would be to increase the frequency at which a snapshot is taken, but at the price of high performance overhead and a race between the attacker and the monitor. Another solution would be to randomize the snapshot interval, it would make it more difficult for the attackers to predict when to erase their traces, but the time between the attack and its detection would be non-deterministic. Even if there is a non-zero probability of detecting the intrusion, the attacker could remain undetected even for extended periods of time. For example, a worst-case scenario would be that the IDS always takes a snapshot at the wrong time just after the attackers erased their traces.

3.2.2.2 *Event-Driven Monitoring*

The inability to detect transient attacks motivated researchers to develop event-driven approaches where the monitor observes events generated by the monitored system to know whether an intrusion occurred. In the taxonomy of Debar et al. [87], they relate to the transition-based detection paradigm. All the following event-driven approaches require a new specific hardware component or a modification of an existing hardware component.

Vigilare [202] snoops the memory bus traffic of the host by using an external hardware component to detect modifications of immutable regions of a kernel. This approach does not suffer from transient attacks: as soon as an illegal modification is made it is detected. KI-Mon [167]—its successor—also monitors mutable kernel objects. MGuard [180] follows a similar approach but incorporates the integrity monitor inside a DRAM DIMM device. One limitation, however, that affects these solutions, is their inability to access the CPU state of the host they are monitoring (semantic gap issue). Jang et al. [143] demonstrated the practicability of an evasion scheme—called Address Translation Redirection Attack (ATRA)—that modifies the page table entries and the CR3 register that contains the physical base address of the page directory. By modifying the memory mappings, such an attack can deceive hardware monitoring solutions that assume legitimate mappings for their monitoring.

Instead of monitoring memory accesses, Duflot et al. [101] propose a runtime firmware integrity verification system for a network adapter which monitors every instruction executed on the network adapter processor using available debugging features. Their approach is two-fold: first, they use a step-by-step verification of each instruction executed to detect code injection; second, they use a shadow call stack [114] to detect control flow alterations.[56] Since they monitor every instruction executed they have a full knowledge of the current CPU state, hence they are not affected by ATRA. This approach can detect multiple attacks, and more specifically the ones that the authors previously found on network adapter firmware [102]. The monitoring required to verify each instruction executed, however, incurs a high computational cost (100% CPU-usage when the firmware is in a busy loop).

Finally, Lee et al. [169] took a similar approach that monitor instructions, where they used debugging features available in ARM processors to monitor branch behaviors (e.g., jumps or calls) of a program and compare it to a reference model built via static analysis.[57] In comparison, their solution incurred only a 2.39 % running time overhead in average, because they observe branch behaviors and not all instructions executed. Davi et al. [84] extend the instruction set of the processor to enforce a similar policy. A document from Intel [136] suggests that future Intel processors will have a backward-compatible technology to monitor branch behaviors in hardware (and available for the SMM). The main limitation of such approaches, in terms of monitoring, is their lack of flexibility. The hardware modifications made can only provide control flow events to the monitor. If the de-

[56] They implement their monitor in the OS, but we believe that their approach could also work via an external hardware component.

[57] Such a policy is also known as Control-Flow Integrity (CFI). Here we only focus on the *monitoring* aspect, we talk about detection methods—including CFI—in subsequent sections.

tection method or the detection model evolves in the future—due to an evolution of the threats and the vulnerabilities exploited—more hardware modifications will be required.

3.2.3 *Detection Method*

We now focus our attention to the last question that is *how to detect* an intrusion based on the information (a state or an event) obtained by the monitor.

3.2.3.1 *Knowledge-Based vs Behavior-Based Approaches*

The knowledge-based detection is a popular approach among commercial IDSs due to its simplicity. For example, host-based IDSs such as Prelude-LML [274] or OSSEC [63] monitor log files and try to match the signatures to each log entry, if one matches it raises an alert. This approach has also been used to detect intrusions in low-level components by DeepWatch [34].

This approach, however, has multiple limitations. First, it lacks flexibility, because it cannot detect novel attacks since it needs signatures for them. Second, if a signature is too generic it creates false positives, if it is too specific it does not match with variants of an attack. Third, it imposes a maintenance burden, because the freshness of the signature database is an important factor for the quality and efficiency of the IDS.

Due to these limitations, in the rest of this work, we focus on behavior-based approaches. Such approaches rely on a model of the expected behavior, and they detect any discrepancy between the reference model and what the monitor observed. The advantage of these approaches is that they can detect new attacks, but great care needs to be taken when constructing the reference model to limit the number of false positives.

3.2.3.2 *Code Integrity*

Code integrity is often guaranteed by existing hardware features, like the MMU (e.g., a non-writable page of memory), even for low-level software such as a kernel or a firmware. Hence, attackers focus on either trying to violate the integrity of the data or the control-flow, as we see in the subsequent sections.

In terms of detection—in case the hardware features are not available or if they are bypassed—a common approach is to perform a measurement of the code and to compare it against a known legitimate value. For example, Copilot [220]—which is a snapshot-based approach—uses cryptographic hashes to ensure the integrity of a kernel against rootkits.

Another approach, called software-based attestation allows a software component—the prover—to attest its integrity to a trusted third-party—the verifier [174, 175, 238]. It uses a challenge-response protocol where the prover computes a checksum of its memory and returns the result to the verifier.[58] The verifier expects that the prover responds within a predictable time. To detect any modification in the program, the verifier computes the checksum using the expected memory content, and compares its value with the received one. A mismatch means that the prover is likely compromised.

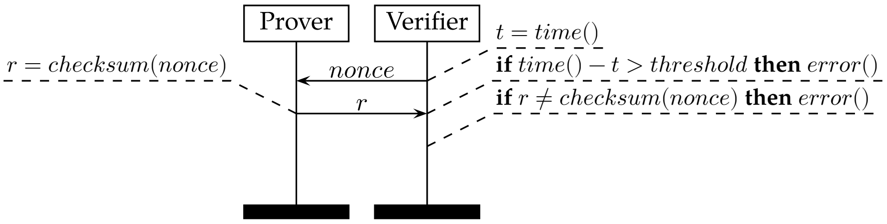

Prover
Verifier
$r = checksum(nonce)$
$nonce$
r
$t = time()$
if $time() - t > threshold$ **then** $error()$
if $r \neq checksum(nonce)$ **then** $error()$

Figure 3.2: A prover attests its integrity via a challenge-response protocol

Software-based attestation systems, however, have various limitations [43, 174]: 1) they require a predictable computation time 2) the software is unresponsive during the computation 3) the device is reset after a verification 4) it requires an optimal implementation of the checksum function so that any modification is noticeable. These limitations make us question the suitability of software-based attestation for software components like firmware or OSs [43, 101].

In the case of the SMM code, the memory protection provided by the MMU is now used by firmware vendors [287, 289]. Since current platforms can ensure code integrity at runtime, we consider that this aspect is out-of-scope in the rest of our work, and we now focus on other detection methods.

3.2.3.3 *Data Integrity*

Data integrity approaches aim at detecting any malicious modification of the data used by the software (e.g., argument of a function or authentication data). Like code integrity, data integrity can be guaranteed using hardware features, like the MMU (e.g., a non-writable page of memory), when it is known to be immutable. Other approaches exist for when such features are not available, if we assume it can be bypassed, or if the data is mutable.

Vigilare [202]—which is an event-driven approach—focuses on detecting modifications of immutable regions of an OS kernel, such as the system call table or the Interrupt Descriptor Table (IDT). Vigilare filters out the traffic to only handle write accesses, and it detects violations of the integrity if a write access is performed between the bounds of an immutable region.

[58] This approach is usually intended for resource-constrained systems without cryptographic hardware features (e.g., systems that cannot embed a TPM chip).

The main limitation of Vigilare is that it cannot monitor mutable kernel objects. Such data are modified during runtime and should only be modified in a specific context (e.g., the user identifier of a process is modified when calling the `execve` or `setuid` system calls). Thus, attacks such as Direct Kernel Object Manipulation (DKOM) [39] which modify or remove dynamic kernel data, or Kernel Object Hooking (KOH) which is a subset of DKOM that only modifies function pointers, are not detected.

As a follow-up work, Lee et al. [167] proposed KI-Mon—an event-triggered kernel integrity monitor, similar to Vigilare—that handles dynamic kernel objects. They added the notion of monitoring rules. Each rule defines what region needs to be monitored and a callback function. This callback function is called by KI-Mon when it noticed that the region is modified. Then, it is possible for the callback function to send DMA requests when a dynamic object is modified to fetch its current value and to verify invariants. They demonstrated that it detects attacks such as hooking functions to hide traces of a rootkit or hiding a kernel module (usually stored in a linked list). For instance, to detect illegitimate function hooking, a rule can define the data structure containing function pointers as a region to monitor, and the callback function can verify if a modification is legitimate by checking that the new value of a function pointer is within a whitelist. Such an approach, however, is low-level and time-consuming since you need to specify implementation details about the kernel.

As a follow-up work to Copilot [220], Petroni Jr et al. [219] proposed a specification-based approach to detect the presence of kernel rootkits. Their approach uses a high-level security specification that defines invariants and relationships among kernel data objects. This specification is then compiled into low-level machine code executed by the monitor at runtime against the kernel memory to ensure that the specification is followed. The difference with KI-Mon is that they write a high-level specification—leaving the low-level details to the compiler of the rules. For example, to detect hidden processes, they can write a rule using predicates that states that for each process t that is in the list of running tasks, t must be in the list of all the tasks.

Currently, those approaches are difficult to automate, and they require an expert with deep knowledge of the monitored kernel. That is why, in our work described in Part III, we focus on a subset of data integrity that allows us to develop approaches that are easier to automate. First, we implement the detection method described in the next section, as it can be automated more easily with less involvement from the developers. Second, we rely on data invariants to ensure the integrity of security-relevant CPU in the context of the SMM. The general framework of our approach, however, could be used in theory to implement data integrity approaches similar to the ones previously mentioned.

3.2.3.4 *Control Flow Integrity*

Widely used defense mechanisms such as non-executable data and non-writable executable code impede attackers in their ability to exploit low-level vulnerabilities. Nevertheless, if an attacker managed to modify an instruction pointer due to a vulnerability, then program execution would be compromised. For example, in an x86 architecture, programs store the return address of function calls on the stack. An attacker could exploit a buffer overflow to overwrite the return address with an arbitrary one that redirect the execution flow. Code-reuse attacks, such as Return-Oriented Programming (ROP) [228] or Jump-Oriented Programming (JOP) [27, 47], use indirect branch instructions (i.e., indirect call to a function, return from a function and indirect jump) to chain together short instruction sequences of the existing code to perform arbitrary computations.[59]

The enforcement of a policy over the control-flow can prevent such an attack. This defense mechanism, called Control-Flow Integrity (CFI), enforces integrity properties for each indirect branch where the control-flow transfer is determined at runtime. It ensures that a given execution of a program follows only paths defined by a Control-Flow Graph (CFG). This graph represents all the legitimate paths that the program can follow. The CFG needs to be defined ahead of time and it can be computed via source code analysis [1], binary analysis [296], or execution profiling [285].

THREAT MODEL The idea of checking the integrity of the control flow of a program at runtime dates back to the 1980s [183, 292]. At that time, however, their threat model and constraints were different.[60] Memory protections, such as non-writable executable code and non-executable data, are widely deployed nowadays on modern platforms.[61] Therefore, CFI approaches that target such platforms focus on indirect branches and ignore direct branches. The target of a direct branch is encoded in the instruction performing the control-transfer, hence with a non-writable executable code an attacker is unable to change the target. On the other hand, the target of an indirect branch is encoded in some data (e.g., in a register or in memory). Thus, a vulnerability can be exploited to modify such data and change the target of an indirect branch even if the code section cannot be modified. Some implementations of CFI [1] also require non-executable data, because each potential destination of an indirect branch is encoded by a unique identifier added next to it. If the data section could be executed, an attacker could exploit a vulnerability to inject instructions into a writable data section and add the identifier of a legitimate function. Then, when the control-flow is redirected to this area, the runtime check passes.

[59] It leads to, what is called in the community, a “weird machine” [32] with its own “weird instructions”.

[60] It illustrates the fact that, in computer science, many ideas at a high level, that have already been proposed, are "rediscovered". However, in practice, the context, the threat model, or the constraints evolved since then. It results in new problems that need to be solved, even if at a high level the ideas are similar.

[61] For example, on x86 platforms, the NX bit (no-execute) in the entry of a page table allows a system to mark a page as non-executable.

INLINED VS EXTERNAL MONITOR A typical way to enforce CFI is by instrumenting the code, e.g., during the compilation phase. This inlined-based approach adds runtime checks before each indirect branch [1, 264, 276]. If the address is not within a finite set of allowed targets, the program stops. Inlined-based approaches, however, share the same execution environment than the attacker, therefore if attackers manage to leak secrets or corrupt data structures that the runtime checks rely on, they could bypass the detection.

In contrast to inlined CFI, which predominates the current implementations of CFI [1, 115, 208, 216, 264, 276, 296], Coudray et al. [74] chose to externalize the mechanism enforcing the policy. Here the monitored program communicates with an external process before every control-flow transfer. If the external monitors detect a violation of the policy, it kills the monitored process. In comparison to an inlined monitor, the external monitor and its data are isolated from the program, but the communication induces a runtime overhead. Coudray et al. use a push-down automaton to detect policy violations, instead of validating if the target of a control transfer is within a finite set of targets, like previous approaches. They instrument the code during compilation where they add a call to the external monitor at different sites: before and after a call to a function, at the start of a function, at the end of a function, before and after a jump.

Full hardware-based CFI solutions [136, 169] also externalize the verification, but do not require any instrumentation. However, they either need to modify the CPU [136] or to rely on debugging features from the CPU [169]. Moreover, the full detection logic gets locked into hardware.

Depending on the use case and the constraints, one approach would be preferred over the others. As mentioned in Section 3.2.1, we decided to use a hybrid approach in our work (Part III) on the behavior monitoring of the SMM. We use a dedicated co-processor to isolate our monitor and an instrumentation step to send behavioral information to the monitor—bridging the semantic gap. Indeed, our solution does not focus specifically on CFI but aims at supporting various detection approaches. To limit the hardware modifications, we chose to add one generic hardware component that can be reused to implement various detection approaches. Moreover, in comparison to inline-based approaches, we ensure the confidentiality and integrity of the secrets, data structures, or models used by the detection approaches.

FINE-GRAINED VS COARSE-GRAINED A fine-grained[62] CFI combines a shadow call stack (i.e., an independent protected stack that only stores return addresses) and a precise CFG (i.e., a CFG with a small approximation regarding indirect branches) to enforce CFI on all indirect control transfers.

[62] The qualitative terms fine-grained or coarse-grained that we use to compare the various CFI approaches are limited [35]. Such terms, however, are sufficient to give an overview. For a more in-depth classification, taxonomy, and analysis of CFI, we recommend the survey of Burow et al. [35].

Some implementations [115, 296] sacrifice security over performance by building a less precise CFI. They either focus on protecting the backward-edge on the CFG (e.g., with a shadow call stack) or on protecting the forward-edge (e.g., indirect calls). Davi and Monrose [85] demonstrated that such implementations, called coarse-grained CFI, fail to protect against control-flow hijacking. Carlini et al. [41] also raised awareness on this issue by consolidating the argument that *without* stack integrity (i.e., without using a shadow call stack), CFI is insecure.

Evans et al. [108], however, demonstrated that preventing attacks via a fine-grained CFI is difficult for real-world applications due to software engineering good practices making the construction of a precise CFG a hard task. When constructing a CFG via static analysis, pointer analyses are required if the code use indirect jumps or indirect calls. Such analyses help to infer the set of possible destinations (e.g., functions) for each indirect jump or call. Since a sound and complete pointer analysis is undecidable [224], a CFG is built using an incomplete pointer analysis leading to an over-approximation of the possible values of each pointer. This over-approximation can help an attacker to redirect the control-flow to an address within the sets of over-approximated targets of a branch, but that should not have been considered. Hence, the more precise this analysis is, the less chance an attacker can take advantage of it. Evans et al. [108] argued that a solution to improve the precision of pointer analyses would be that developers add annotations alongside the source code. Other approaches [208, 216, 264] improve the precision by relying on the types of the functions called indirectly—a type-based CFI—instead of relying on pointer analyses.

As mentioned previously, future Intel processors will have the ability to enforce a CFI policy [136], which could be used in SMM. It will use a shadow call stack and state machines to track indirect branches. Intel, however, decided to enforce a coarse-grained CFI when tracking the indirect branches. All indirect calls will be able to call any function in the code (over-approximation), and not just the expected legitimate functions.

In Part III, we implement two detection methods where one is based on CFI. It enforces a type-based CFI on forward edges ensuring that each function called indirectly has an expected type signature known at compile time, and a shadow call stack for the backward edges. We chose a type-based CFI since it outperforms the pointer analysis,[63] and we combined it with a shadow call stack to have a fine-grained CFI enforcement.

[63] We initially tried a pointer analysis to extract the precise set of possible destinations, but we had poor results forcing us to over-approximate to avoid false positives.

3.2.4 *Summary*

In this section, we reviewed the state-of-the-art IDS approaches for low-level components. While we focused our reasoning on the SMM, we did not ignore other approaches that were designed for the SMM that could be adapted. We articulated our review around three questions about the isolation of the monitor, the methods to monitor the target, and the detection models. We now summarize our review of the literature and our findings:

- External monitors offer a better isolation than inlined monitors at the cost of a semantic gap. The semantic gap must be bridged otherwise attackers could potentially bypass the monitoring.
- Snapshot-based approaches will always be limited due to transient attacks.
- Event-based approaches solve this issue, but they might introduce other challenges. For example, some solutions process too many events leading to performance issues. Other solutions limit the events they process, but they rely on CPU modifications or debug interfaces that limit them to specific detection models and make them not flexible.
- Vendors already rely on the MMU for SMM code and data integrity. Detection approaches exist, but they are either not suitable in the context of the SMM, require experts, or cannot be easily automated.
- Few existing approaches were designed to monitor the SMM at runtime [34, 180]. They focus only on ensuring code or data integrity but cannot enforce CFI for example. In addition, they are either vulnerable to transient attacks [34] or suffering from a semantic gap [180].
- A recent document suggests that future Intel processors will be able to enforce a CFI in SMM [136]. Such a technology is, however, unavailable at the moment, and when it is released it will be coarse-grained.

Based on this review of the state of the art, we introduce in Part III a new SMM behavior monitoring approach that aims at addressing the aforementioned limitations. Our approach relies on a dedicated co-processor to isolate the monitor and it bridges the semantic gap by introducing a secure low-latency communication channel. We enforce the communication of behavioral information via this channel by instrumenting the SMM code. With this approach, we demonstrate the feasibility of a flexible SMM behavior monitoring by implementing a type-based CFI policy and execution context policy.

In the next section, we switch to the OS abstraction layer, and we review the state-of-art approaches to achieve intrusion survivability for a commodity OS.

3.3 INTRUSION SURVIVABILITY FOR COMMODITY OPERATING SYSTEMS

Existing intrusion tolerance or intrusion survivability approaches often focused on safety-critical systems[64] and distributed systems [105, 157, 239, 294]. The reasoning is that these systems and the organization that maintain them have the most to lose (e.g., human lives or big financial impact due to a cascade of failures). Non safety-critical systems and single platforms (e.g., non safety-critical embedded systems, desktop or enterprise systems) have been less studied and are currently limited on their ability to survive intrusions. In our work, we focus on such systems.[65] We structure the rest of this section based on three questions that one needs to answer when building systems that needs to survive or withstand intrusions:

1. How to isolate?

   Like IDSs in the previous section, it refers to the approaches that we can use to isolate the components we rely on.

2. How to recover?

   It relates to the approaches that can recover from a compromised system, or an application, to a previous state that we assume not compromised.

3. How to respond?

   It pertains to the approaches (e.g., containment) that respond to an intrusion to limit the future actions of the attackers on the system and also the approaches that select such responses.

In the rest of this manuscript, we differentiate a recovery from a response as explained above. While one could argue that a recovery is a specific form of a response, a recovery only limits the impact of previous malicious actions and it does not prevent the attacker from e.g., reinfecting the system. Moreover, a recovery has specific challenges in comparison to other form of responses, and since various papers focused on the recovery, we decided to separate the notion of recovery from the notion of response.

3.3.1 *Isolation*

As mentioned by Avizienis et al., “it is essential that the mechanisms that implement fault tolerance should be protected against the faults that might affect them” [16]. Indeed, if attackers can compromise any

[64] “Systems whose failure could result in loss of life, significant property damage, or damage to the environment.” [156]

[65] The work we present in Part II was not designed for safety-critical systems, but in theory one could adapt it for such systems.

of the components used to achieve intrusion survivability, they could disable the recovery logic and circumvent any response.

Most of the work that we mention in subsequent sections often consider the kernel as trusted[66] and they rely on the kernel to isolate any of the components used to implement their approach [122, 131, 153, 242, 277]. While such an assumption might be reasonable, in practice the kernel is often the source of many vulnerabilities.

For example, the Linux kernel security as a long history of not being taken seriously [265]. Moreover, the life span of kernel vulnerabilities (i.e., how long a vulnerability might be exploitable before it is fixed) is long. An analysis made in 2010 [72] shows that Linux kernel vulnerabilities have on average a 5 years life span.[67] Another analysis made 6 years later [65] has a similar conclusion.[68] The issue with these kernel vulnerabilities is that they can be used to bypass security features offered by the kernel (e.g., to escalate privileges or bypass restrictions). Thus, the high number of kernel vulnerabilities combined with their ability to bypass kernel security features makes us question the suitability of trusting the kernel to isolate our components when facing advanced and determined attackers.

Fortunately, the Linux kernel community is starting to take the security of its kernel seriously. For instance, Kees Cook has been working these past few years on upstreaming strategies that go "beyond bug fixing" [66, 67, 73], based on out-of-tree (i.e., not included in the mainline kernel) security patches, such as grsecurity [124] and PaX [257].

Nevertheless, the less trust we put in the kernel, the smaller our Trusted Computing Base (TCB) is. Therefore, some approaches [19, 242, 286] ran their target inside a Virtual Machine (VM) and moved the recovery or response logic in a hypervisor or another machine.

Unfortunately, as mentioned in Section 3.2.1 about the isolation of the IDS, the more isolated we are, the less information we have on our target. In the case of intrusion response and recovery, the issue is less about information, and more about control. For example, to recover from an intrusion, our solution must be able to control the target (e.g., to restore a file or a process to a previous state). Therefore, if we do not have access to the kernel interfaces, we have a semantic gap that needs to be bridged to control the abstraction and notions of processes or files—only understood by a kernel or a user space process in general. For example, Xiong et al. [286] built SHELF—an intrusion recovery system—on top of a hypervisor. The authors implemented a basic prototype that can record the state of processes (i.e., copying the address space and some kernel data structures) and rollback their state to a previous checkpoint by introspecting the guest kernel. Their prototype, however, does not fully handle the state of modern applications (e.g., external resources, invisible files, or namespaces). Handling such cases would significantly increase the complexity of

[66] Since we only consider host-based approaches, and not network-based ones.

[67] Such analysis is entirely based on the number of CVEs, thus it does not include vulnerabilities with no CVEs assigned. The fact of not assigning CVEs for vulnerabilities in the Linux kernel is a recurrent criticism [250].

[68] This does not even include the fact that there is also a time between when a vulnerability is fixed, and when the kernel on the platform is actually updated with the relevant patches.

the code introspecting the guest kernel.[69] In addition, one would need to maintain the introspecting code for each new version of the kernel.

In general, there is a trade-off between the control and the isolation that depends on the context and the engineering effort to bridge the semantic gap. Such an engineering effort might introduce complexity into the solution, thus a higher maintenance burden, and more importantly a higher chance to introduce vulnerabilities. In Part II, our approach works at the level of services, and it needs the abstractions and features of the kernel to manage them. We considered that trying to isolate our components without trusting the kernel would introduce too much complexity and engineering effort. Finally, since this aspect was not the focus of our work, we decided to trust and use the kernel to isolate our components.[70]

3.3.2 Intrusion Recovery

Intrusion recovery solutions focus on system integrity after a system has been compromised. They either recover legitimate persistent data that may have been infected, remove any non-legitimate data present (e.g., malware payloads or illegitimate settings in the system), or both.

First, we discuss the industry solutions that are used nowadays by consumers and administrators when they know that one of their system has been compromised. Second, we describe various log-based approaches proposed by the academic community to address the limitations of the industry, and we also mention their limitations as well. Third, we describe container-based solutions. Finally, we explain the main limitation of intrusion recovery solutions in regard to intrusion survivability.

3.3.2.1 Traditional Industry Solutions

Many industry solutions exist to remove the traces of malware on a system (i.e., remove non-legitimate files and restore legitimate but compromised files). For example, anti-malware or anti-virus solutions, such as Windows Defender [198], can remove the traces of malware or the changes they made based on the knowledge of the malware. Paleari et al. [214] also introduced an approach to automatically generate the remediation procedures for malware infection by analyzing their behavior. One main limitation with such approaches is that they can only remove illegitimate objects (e.g., files) from the system, they cannot restore destructive malicious actions such as files being deleted by the malware. Moreover, they might also have false negatives with unknown malware or malware with only scarce information about its behavior (e.g., they might not know that it created some malicious registry keys). Finally, it only works for malware with a predictable or deterministic behavior, but for instance it fails for malware controlled remotely by an attacker.

[69] As an example, Linux kernel maintainers rejected merging a checkpoint and restore functionality in the kernel due to its complexity and lack of benefits in contrast to a userland solution [71].

[70] While we consider it out-of-scope, approaches exist that aim at improving the security of the kernel. For example, Azab et al. [17] introduced a hardware-based security monitor to ensure the integrity of the kernel at runtime, and UEFI-secure boot [272, section 32] can be used to ensure the integrity of the kernel at boot time.

Another common solution, to help to recover to a safe state after an infection, is by reimaging the system, or by reinstalling it, completely. While it ensures that all traces of the malware have been removed,[71] this approach also incurs a high cost to the users since they lose their files (no false negatives, but many false positives).

Other industry solutions exist that rely on periodic snapshots of the file system to restore a system to a previous safe snapshot in case it gets compromised.[72] For example, Windows System Restore [197] allows a user to restore system objects—no user data is restored. Another traditional approach is to use a VM where one performs sensitive tasks in the VM and takes periodic snapshots of the VM. In case of an intrusion, one can revert the VM to a previous snapshot. Such solutions, however, perform a coarse-grained recovery that recover to a previous state many non-infected and legitimate data. Another issue is that to limit the data loss, users might have to identify the files that needs to be restored or not—which might be a difficult and error-prone task.

The approaches proposed by the academic community try to address these limitations. They differ from industry solutions since they try to minimize the number of false positives, the number of false negatives, or the availability cost while doing the process automatically. We now discuss these approaches.

3.3.2.2 *Container-Based Approaches*

As mentioned before, one can execute a monitored system in a VM and can take periodic snapshots to recover in case of an intrusion. Such a recovery approach, however, lose the accumulated modifications of many legitimate and non-infected objects (e.g., files or processes) since the last non-infected snapshot. In addition, starting or restoring a VM is a heavy-weight process. Thus, it might incur a high availability cost to the user.

Shan et al. [242] proposed an approach to address this issue using containers.[73] They introduce a secure commitment approach where a container commits, or merges, only legitimate persistent data changes to the host. They track information flow in the OS and attach labels to objects to put them into clusters. Then, they identify malicious clusters based on their behaviors, and they filter them out when committing the changes back to the host. However, they only focus on persistent data, and they do not preserve availability (i.e., they lose the processes' state), since the commitment only occurs when the container is shutdown.

CRIU-MR [277] restores infected systems running within a Linux container. When an IDS notifies CRIU-MR that a container is infected, CRIU-MR checkpoints the container's state (using CRIU [76]), and identifies malicious objects (e.g., files) using a set of rules. Then, it restores the state of the container while omitting the restoration of

[71] If we assume that neither the BIOS nor other firmware on the platform have been compromised.

[72] While they do not necessarily focus on recovering from a compromised system, it can be used for such a case.

[73] In their paper, they use the term "OS-level virtual machine", also known as "feather-weight virtual machines" [293]. Nowadays, however, the term container is more common.

the malicious objects previously identified. It differs from the work of Shan et al. [242], since CRIU-MR does not filter out malicious objects during a commitment phase, but it identifies the malicious objects at checkpoint time and filter them out during the restoration. In addition, they do not need to reboot the container, and they manage to keep the state of active TCP connections during the operation. However, they rely on the ability of the IDS to identify the illegitimate objects. Hence, much like traditional approaches, this solution can suffer from false negatives (e.g., the IDS does not know that a specific file has been compromised or is malicious).

In comparison to these approaches, in our work presented in Part II, we do not restrict our solution to containers. Indeed, while many services can be containerized, some system services still need access to many resources from the system—using a container would hamper their function. We do, however, use some features from the OS used by containers to simplify our recovery process (i.e., we can track all the processes created by a service).[74]

3.3.2.3 *Log-Based Approaches*

Other approaches focus on trying to minimize the number of false positives and false negatives when doing a recovery. Most of these approaches log all system events to later replay legitimate operations [19, 122, 153, 286] or rollback illegitimate ones [131], thus providing a fine-grained recovery.

Taser [122] recovers legitimate persistent file system data after an intrusion by relying on taint tracking, audit logs, a legitimate file system snapshot, and replay operations. During the normal operation of the system, Taser logs all system events (e.g., arguments of system calls or file system operations) and the identity of the objects (e.g., processes) associated with those events. After an intrusion occurred, Taser uses taint tracking by following the information flow generated by systems calls (e.g., if a process is tainted and it writes to a file, this file is also tainted) to identify all the events associated with an intrusion. It first finds the source object (e.g., a process or a socket) that is responsible for the intrusion, and from there taints all the events that follows their dependency rules. Finally, the system is shut down and Taser replays only the legitimate operations (i.e., non-tainted) on an immutable file system snapshot.

In practice, Taser has many false positives—it marks as tainted legitimate events and rollbacks many legitimate data. To mitigate the number of false positives, Taser uses whitelists (e.g., never taint `/dev/null`) or ignores completely some events (e.g., reading file attributes, names, or content). While such policies reduce the number of false positives, it introduces false negatives (e.g., it does not rollback illegitimate events).

---

[74] We also rely on other features from the OS used to implement containers for the responses that we apply, but in this section we only focus on the recovery.

Retro [153] follows an approach close to the one of Taser. During the normal operation of the system, it records periodic checkpoints of the file system, and it logs system events to build an action history graph that represents objects in the system and their dependencies. After an intrusion is detected, it relies on its action history graph to find the first intrusion point. During the recovery phase, the system is rebooted, and Retro restores the affected files to a previous state, and changes malicious system call arguments to benign ones (e.g., instead of executing malware, a program executes `/bin/true`). More importantly, to minimize the recovery of legitimate data and to avoid inconsistencies, it re-executes the actions affected by the intrusion with the new benign arguments since their behavior might be different. Finally, it only re-executes actions with dependencies that are semantically different after a repair. For example, if a legitimate program reads the last line of a file and an illegitimate process only modified the first line of a file, Retro does not re-execute the former action since it would be equivalent.

In comparison to Taser, Retro has less false positives and false negatives. They claim to achieve such better results based on their action history graph, on the re-execution and selective re-execution, instead of Taser's taint tracking.

A limitation of intrusion recovery systems that only focus on persistent data, such as Taser or Retro, is that they do not preserve the availability of the system, nor any of the services. Their restore procedure either forces a system shutdown or reboot [19, 122, 153].

SHELF [286], however, does not suffer from this limitation. During the normal operation of the system, it takes periodic snapshots of both the state of the processes and the files, and it logs system events like Taser. In case of an intrusion, it allows SHELF to recover the state of processes using a previous snapshot and to identify infected files using taint tracking on the logs. During the recovery procedure, SHELF quarantines infected objects by freezing processes or forbidding access to files. SHELF, however, removes this quarantined state as soon as it restores the system. The main advantage of SHELF is its ability to minimize the availability loss. First, it does not lose the state of processes by restarting them, but checkpoint and restore them to their last checkpoint. Second, it does not require shutting down the system, shutting down applications, or closing network connections. Therefore, it maintains a higher availability than previous approaches.

The approaches previously discussed, that relies on logging system events, have all a limitation related to the explosion of the number of dependencies per object. They monitor all the system events and follow the dependencies between these events (e.g., a process created a file, the file is read by another process, this process creates another process etc.). In practice, unfortunately, a system generates many events during its lifetime, and they need to store this information to reuse it in case of an infection. It results in gigabytes of logs generated per day inducing a high storage cost.[75]

In our work, described in Part II, we restore services to a previous state if they are compromised. Our approach is close to the log-based approaches that we mentioned, but we make a trade-off between the precision of the recovery and the amount of logs generated by restricting the monitoring per-service and by focusing on specific events. In addition to restoring the files, we restore their processes to maintain as much as possible the availability. The intrusion recovery, however, is only one component of our work among others that together aim at achieving intrusion survivability.

3.3.2.4 *Application for Intrusion Survivability*

While intrusion recovery is necessary to have a system that can survive intrusions (otherwise illegitimate content would still be present), it is not sufficient. Indeed, one common limitation affecting all the prior work discussed in that area is that they prevent neither the attacker from reinfecting the system nor the attackers from achieving their goals (e.g., integrity violation or propagation) when successfully reinfecting the system.

It means that the system is restored to a safe state, but it cannot withstand a reinfection. After the recovery, the attacker can still attack the system, and the vulnerabilities used initially to compromise the system are not fixed. A worst-case scenario could be a loop of infections and recoveries impacting the availability of the system, or its services, since the cost of the recovery is not negligible.

3.3.3 *Intrusion Response Systems*

Having discussed systems that recover from intrusions and showed that it is not sufficient to withstand them, we now discuss intrusion response systems that focus on applying responses or countermeasures to limit the impact of an intrusion.

Shameli-Sendi et al. showed that there is an abundance of prior work on intrusion response systems [239, 240]. They classify existing approaches based on:

- the level of automation (e.g., manual or automated),
- the evaluation of the response cost (e.g., static or dynamic),
- the response time, the ability to adjust (e.g., adapt or not the responses based on their successes or failures),
- the response selection (e.g., static or cost-sensitive),
- the location of the response enforcement (e.g., a firewall, the infected host, or another intermediary),

[75] There is, however, some work [168, 185] on how to reduce the size of the logs.

- and the response lifetime (i.e., temporary or long-term).

Yuan et al. [294] also made a survey on self-protecting software systems—systems that are “capable of detecting and mitigating security threats at runtime”—that includes intrusion response systems. While other taxonomies and surveys have been proposed in the past [42, 110, 123, 252], these are the most recent, they took into consideration limitations of previous surveys, and they provide an overview of the existing approaches.

While we reuse some classifications presented in these surveys, we do not go into as many details for each paper we mention as they are. In addition, we take into account aspects that are missing from these taxonomies, but that we consider relevant and needed to characterize and differentiate our work. The rest of this section is thus structured as follows. In Section 3.3.3.1, we discuss the lack of work on single platforms that are not part of a distributed system. In Section 3.3.3.2, we discuss the methods used to select responses and the models they rely on. Finally, in Section 3.3.3.3, we study the granularity of the response strategy of various approaches.

3.3.3.1 *Response Topology: Global vs Local Approaches*

One point that we notice in the literature is that prior work mostly studied intrusion responses that work at the level of a network of hosts, and not at the level of a single host that is not part of a globally managed network. Yuan et al. [294] characterize systems based on their “control topology”: local only, globally centralized, or globally decentralized. Only 30 % (32/107) of the papers they reviewed had the ability to “respond” with a local only control topology.[76] Moreover, Shameli-Sendi et al. [239] presented almost only network-centric approaches. Indeed, they even considered as a limitation the fact that an approach cannot be applied to a network of several hosts. While we understand the need to scale intrusion response approaches to globally managed networks (e.g., for enterprise environments), the focus on a global scale also means that most of the prior work do not apply to single hosts that are not part of globally managed networks or distributed systems (e.g., embedded systems, single servers, or consumer laptops and desktops). This analysis was already made by Balepin et al. in 2003: “The analysis of related work leads us to the conclusion that the primary area of interest so far has been a computer network that consists of multiple hosts. The idea of responding at a level of a single host has received relatively little attention.” [20]

Some ideas of network-based and host-based approaches are similar at a high level. The difference in scale, however, means that in practice different problems need to be solved. The different context and environment also have an impact on how we approach the constraints (e.g., performance) and how we design the solutions.[77]

---

[76] Their category of response systems, however, is broad since (unlike us) it also includes intrusion recovery systems.

[77] While we focus this part of our study of the state of the art on intrusion response approaches for an OS, we also mention some related work initially designed as a network-based solution since some ideas could be adapted.

3.3.3.2 *Response Selection and Models*

One area of focus of prior work is on how to select optimal responses. Most of recent approaches use a cost-sensitive response selection process that take into account both the cost of the intrusion and the cost of the responses [239]. Many approaches also use graphs to help select responses. For example, they rely on one or a combination of directed graphs about system resources [20], attack graphs [113], attack defense trees [241], or service dependencies [151].

A graph about system resources [20] have nodes that represent resources of the system, and its edges represent the functional dependencies between them. Each node contains its type, its cost (or importance), and a list of response actions that can be triggered to restore the resource to a working state if it gets compromised. The graph is built manually, but some nodes can be added dynamically based on alerts from the IDS if it mentions resources not present on the graph. Kheir et al. [151] also propose a functional graph that focuses on services and their dependencies to evaluate the impact of an intrusion and its responses on services.

Attack graphs represent the paths that attackers can use to compromise a system or a network. Logical ones have nodes that contain information about the preconditions that attackers need to carry out an attack, and the edges represent the dependencies between them. Topological ones have nodes that represents assets (e.g., a host) and the edges represent possible attacks between each node. They can be generated automatically e.g., from vulnerability scans and network topology [213, 245]. Attack-defense trees [160] adds the ability to model the various defensive measures that one can take to thwart the progress of the attacker by adding defense nodes.[78] Various approaches have been proposed using such models.

Foo et al. [113] proposed an automated intrusion response system using attack graphs to contain intrusions in a distributed e-commerce environment. When an alert occurs, the system determines which goals the attackers most likely achieved in the graph and which goals the attackers will most likely target next. Then, it chooses the responses to deploy in order to contain the intrusion according to the effectiveness of the response against the ongoing attack and the perceived disruptiveness to legitimate users. Their solution, however, is aimed at a distributed system, and does not work at the granularity of the service of an OS.

Kheir et al. [151] rely on both attack graphs and a service dependency graph to evaluate the impact of an intrusion and the responses. They take into account the confidentiality, integrity, and availability impact on the services. They also rely on the notion of privileges that attackers need for their attack. They model the privileges that one service needs from another service, and may remove such privileges to thwart an intrusion. However, while their model aims at limiting

[78] For more details about attack graphs, attack trees, or defense tree see the work of Shandilya et al. [243] on the "Use of Attack Graphs in Security Systems" and the survey of Kordy et al. [161] on attack and defense modeling techniques.

the impact of an intrusion, it only focuses on limiting the propagation of an intrusion from one service to another. The privileges that they consider are basic access controls mechanisms, such as credentials or whether an IP address is approved. In addition, they only evaluated their model via a simulation, but they did not evaluate the feasibility, security, or performance of their approach on real systems. In our work (Part II), our approach makes the system survive an intrusion by removing privileges not only to limit propagation, but to stop the attackers from achieving their goal in general. In addition, we use a more low-level notion of privileges (e.g., the ability to perform system calls or accessing specific files in the system) and resource constraints (e.g., CPU quotas) allowing us to target precise malicious behaviors.

Shameli-Sendi et al. [241] also use a service dependency graph, but they rely on an attack-defense tree to model the attacker and defender actions. They also use multi-objective optimization methods to select an optimal response based on the performance, the deployment cost, and the quality of service cost of a response.

One thing that these approaches [113, 151, 241] have in common is that they rely on some information about the vulnerabilities affecting the system for their models. They either assume the knowledge of vulnerabilities present on the system, or that they know which vulnerabilities the attacker exploits. In practice, however, one is not necessarily aware of all the vulnerabilities present on the system. First, with all known vulnerabilities,[79] one needs to continuously check the systems to determine whether they are vulnerable to those. Second, there are unknown vulnerabilities[80] that can be exploited by attackers to compromise the system. For example, Ablon and Bogart [2] found that “exploits and their underlying vulnerabilities have a rather long average life expectancy (6.9 years)” and that “for a given stockpile of zero-day vulnerabilities, after a year, approximately 5.7 percent have been discovered by an outside entity”. It shows that intrusion response systems that only rely on the assumption of prior knowledge about the vulnerabilities present in the system cannot always select appropriate responses, since attackers can use unknown paths and bypass the underlying logic.

Balepin et al. [20] designed a specification-based automated intrusion response system that focuses on one host in comparison to other approaches. It relies on a directed graph about system resources, and cost models of responses, to select responses minimizing the cost of the intrusion (the sum of the costs of each resource nodes affected) and the cost of the responses against the resources. While their approach does not rely on information about vulnerabilities, they do not describe any models regarding the attacker, the intrusion, or the malicious behaviors to help them select responses. Indeed, in addition to the difficulty of building a precise graph of system resources, they need precise information about the resources affected by an intrusion

[79] We do not necessarily mean all publicly known vulnerabilities, since vulnerabilities can be public, but still unknown to the organization. Moreover, the organization can be aware of vulnerabilities not yet public.

[80] Privately known vulnerabilities that have not been addressed are often referred as zero-days.

to compute its cost. In our work (Part II), we build and rely on a model of intrusions based on the various malicious behaviors that they can exhibit. It allows our approach to handle the case where we have precise information about an intrusion (we know exactly what it is doing and how), and the case where we only have a generic understand of its behavior (e.g., we only know it is ransomware but not any more details).

Others also proposed non graph-based solutions. For example, Gehani and Kedem [119] defined a formal framework for real-time risk management. It allows one to dynamically reconfigure an access control subsystem to limit the exposure of the system when the risk is above a tolerance threshold. They compute the risk based on multiple factors such as the likelihood of a threat, the consequences, and the exposure of the system. Their approach, however, is intrusive since they require a reference monitor to be present in the applications to enforce their policy.

Finally, we can also distinguish approaches based on whether their risk assessment methodology is quantitative or qualitative. Quantitative methods rely on numbers such as monetary values to compute the risk using statistical or probabilistic models. For example, Motzek et al. [204] propose quantitative methods to help select response plans by assessing the financial impact and the operational impact of a response. Qualitative methods use linguistic constants (e.g., low or high) to rate the risks and rely on security experts to define them. Both methods have benefits and limitations [279]. Quantitative ones are complex and require an accurate value of assets and historical data of previous intrusions to be effective. Qualitative ones are prone to biases and inaccuracies, but they are easier to understand, to reason about, and to implement. In our work (Part II), we rely on qualitative risk assessment, since we do not assume to have historical data of previous intrusions, and we want to limit the difficulty for a user to provide input to our models—thus improving its usability.

3.3.3.3 *Response Granularity: Precision and Scope*

The response granularity is linked to the scope and the precision of the responses that intrusion response systems apply. The scope of the response is the amount of components impacted by a response. We say that responses are fine-grained if they only affect compromised components. Coarse-grained responses, on the other hand, impact additional non-infected components. For example, an intrusion response system might apply a response that impacts all the hosts of a network (e.g., modify the firewall rules of a firewall node or a router), a single host (e.g., add additional local firewall rules or restart the system), or a service inside a host (e.g., deny the HTTP server to access specific files).

The precision of the response is its ability to target specific malicious behaviors. A fine-grained response should only impact a specific malicious behavior or threat while minimizing the impact on the compromised component. Coarse-grained responses may impact other non-malicious functions of a component. For instance, if a response stops a compromised service to withstand an infection, it is considered coarse-grained, because the response impacted the whole service just to thwart one specific threat.

It is important to distinguish the granularity from the topology discussed in Section 3.3.3.1. One approach can be global but fine-grained if it is able to apply a response only to infected hosts and if it targets only the specific malicious behavior.

The response cost Shameli-Sendi et al. [239] is a close aspect of the response granularity. Indeed, the more coarse-grained a response is, the more components of the system it impacts, therefore it has a greater cost.

The response granularity is an aspect that has not been subject to the same level of attention in prior work and surveys in comparison to the other aspects (e.g., computation of response selection and response cost). Indeed, most of the prior work apply coarse-grained responses to compromised hosts [20, 113, 241]. They often list simple coarse-grained responses, such as restarting a service, modifying the firewall, rebooting the system, deleting a file, or changing file permissions. Such responses do not offer enough flexibility to withstand intrusions that exhibit various malicious behaviors, without impacting the availability of the compromised components.

Some approaches offer more fine-grained responses than others. Gehani and Kedem [119] provided a fine-grained approach since it can reconfigure access control policies per application if the risk is above a tolerance threshold. As mentioned in Section 3.3.3.2, their approach, however, is intrusive.

Thomas [262] described a threat response approach that automatically updates a security policy—built upon Organization-Based Access Control (Or-BAC) [144]—according to alerts from the IDS. He also introduces a response strategy taxonomy that includes the target layer of the response. The approach takes into account the need for more fine-grained response where, for example, it gives the ability to deploy responses either at the network or service level depending on the threat. However, it only offers basic responses at the service level. For example, at the service level, it can restart, stop, patch, or reconfigure a service (e.g., change the listening port). More importantly, the responses may impact the whole host even if they were meant to target a specific service (e.g., blocking the port 80 on the host and not just for one specific service running on the host).

Our approach that we present in Part II takes into account the need for fine-grained responses. Indeed, the more coarse-grained a response is, the more impact it has on the system. Therefore, our solution applies responses that target precise malicious behaviors. In addition, it can apply responses per-service that only impact one specific service (e.g., an HTTP server) and not the rest of the system. Finally, we designed our approach so that it does not require to modify the services, nor their configuration. It applies transparent responses that are agnostic to the type of service.

3.3.4 *Summary*

In this section, we reviewed the state-of-the-art approaches to help achieve intrusion survivability for a commodity OS. Much like the previous section, we articulated our review around three questions about the isolation, the approaches to recover, and the approaches to respond. We now summarize our review of the literature and our findings:

- Intrusion recovery is necessary but not sufficient to withstand intrusions. Regardless of the details of the approaches, current intrusion recovery solutions cannot withstand intrusions—an attacker can reinfect the system at the end of the recovery. We noticed that none of the approaches combined intrusion recovery and fine-grained intrusion response with an intrusion survivability approach.

- The preferred approach for the moment to recover the files or to enforce the responses is to trust the kernel. The high number of kernel vulnerabilities, however, makes us question the suitability of such an approach when facing advanced and determined attackers. Nevertheless, we need to take into account the engineering effort and complexity introduced by solutions that do not trust the kernel.

- Most industry intrusion recovery solutions incur a high false positive rate. Most of the academic solutions address this issue, but they have a high resource cost and monitoring cost in return.

- Many approaches need to reboot or stop the system to start their recovery procedure. Few intrusion recovery approaches [277, 286] studied how to maintain the availability of the system and its applications when restoring.

- Most of the intrusion response systems focus on a globally managed network of hosts, distributed systems, or safety-critical systems. Few approaches [20] studied how to design intrusion response systems for a single platform (e.g., an embedded system, a single server, or a consumer laptop) with a commodity OS.

- Several approaches assume to know the vulnerabilities present to select responses. In practice, however, attackers can exploit unknown ones.
- Finally, most of the intrusion response systems apply coarse-grained responses that impact the whole system and not just the compromised component. In addition, their responses are not precise and impact significantly the compromised component. Other approaches were able to apply more fine-grained responses [119, 262], but they are either intrusive, limited in their type of response, or require a semantic understanding of the configuration files of each service.

The next section summarizes this chapter, concludes our review of the various disciplines and existing approaches, and it introduces our contributions.

3.4 CONCLUSION

In this chapter, we gave an overview of concepts such as intrusion detection, intrusion survivability, intrusion recovery, and intrusion response. These concepts are essential to the understanding of our work, since we derived our ideas and subsequent contributions from them. Moreover, related to these concepts, we reviewed the state-of-art approaches and techniques that we consider the most relevant to our work and the problems detailed in Section 1.1.2 and Section 1.1.3. In particular, we took an in-depth look at intrusion detection approaches for low-level components, and intrusion recovery and response approaches for commodity OSs.

While studying the state of the art in these fields, we noticed gaps in the literature that we summarized in Section 3.2.4 and Section 3.3.4. These gaps motivated our research and our contributions. While we do not pretend to fill all the gaps, the approaches we present in Part II and Part III address limitations that we identified in the literature.

More specifically, in Part II, we address the inability of commodity OSs to withstand intrusions by introducing a new intrusion survivability approach aimed specifically at such systems. Our approach differs from the state of the art, since we combine both intrusion recovery and intrusion response, apply fine-grained responses without modifying applications, and maintain the availability of the system.

In Part III, we address the lack of intrusion detection approaches for components at the firmware level. To demonstrate our approach, we use the runtime part of the BIOS—the SMM code—as a target for an event-based and co-processor-based monitoring approach. Our approach differs from the state of the art by the fact that it manages to isolate the monitor while bridging the semantic gap, it is flexible with regard to the detection models, and it takes into account the performance considerations of the target.

We give more details about these approaches, their differences with state-of-the-art approaches, and the experiments we ran in subsequent chapters.

Part II

SURVIVING INTRUSIONS AT THE OS LEVEL

4

INTRODUCING AN INTRUSION SURVIVABILITY APPROACH

Having explained the necessary background and discussed the current state-of-the-art solutions, we now introduce our first main contributions.

The rest of this chapter is structured as follows. First, we recall the motivation behind our work and introduce our contributions. Second, we give an overview of our approach and its goals. Third, we provide the threat model and the assumptions that we make throughout the rest of this part. Finally, we provide some illustrative examples about how our approach can withstand different intrusions.

4.1 MOTIVATION AND CONTRIBUTIONS

To limit the damage done by security incidents, intrusion recovery systems help administrators restore a compromised system into a sane state. Common limitations of such solutions are that they do not preserve availability [122, 131, 153] (e.g., they force a system shutdown) or that they neither stop intrusions from reoccurring nor withstand reinfections [122, 131, 153, 277, 286]. If the recovery mechanism restores the system to a sane state, the system continues to run with the same vulnerabilities, and nothing stops attackers from reinfecting it. Thus, the system could enter a loop of infections and recoveries.

Existing intrusion response systems, on the other hand, apply responses [113] to stop an intrusion or limit its impact on the system. However, existing approaches apply coarse-grained responses that affect the whole system and not just the compromised services [113] (e.g., blocking port 80 for the whole system because a single compromised service uses this port maliciously). They also rely on a strong assumption of having complete knowledge of the vulnerabilities present and used by the attacker [113, 241] to select responses.

These limitations mean that they cannot respond to intrusions without affecting the availability of the system or of some services. Whether it is due to business continuity, safety reasons, or the user experience, the availability of services is an important aspect of a computing platform. For example, while web sites, code repositories, or databases, are not safety-critical, they can be important for a company or for the workflow of a user. Therefore, the problem that we address is the following: how to design an OS so that its services can survive ongoing intrusions while maintaining availability?

Our approach distinguishes itself from prior work on three fronts. First, we *combine* the restoration of files and processes of a service with the ability to apply responses after the restoration to withstand a reinfection. Second, we apply *per-service* responses that affect the compromised services instead of the whole system (e.g., only one service views the file system as read-only). Third, after recovering a compromised service, the responses we apply can put the recovered service into a *degraded mode*, because they remove some privileges normally needed by the service.

The degraded mode is on purpose. When an intrusion is detected, we do not necessarily have precise information about the vulnerabilities exploited to patch them, or we do not have a patch available. The degraded mode allows the system to survive the intrusion for two reasons. First, after the recovery, the degraded mode either stops the attackers from reinfecting the service, or it stops the attackers from achieving their goals. Second, the degraded mode keeps as many functions of the service available as possible, thus maintaining availability while waiting for a patch.

We maintain the availability by ensuring that core functions of services are still operating, while non-essential functions might not be working due to some responses. For example, a web server could have "provide read access to the website" as core function, and "log accesses" as non-essential. Thus, if we remove the write access to the file system it would degrade the service's state (i.e., it cannot log anymore), but we would still maintain its core function. We developed a cost-sensitive response selection where administrators describe a policy consisting of cost models for responses and malicious behaviors. Our solution then selects a response that maximize the effectiveness while minimizing its impact on the availability of the service based on the policy.

This approach gives time for administrators to plan an update to fix the vulnerabilities exploited by the attacker (e.g., wait for a vendor to release a patch). Finally, once they patched the system, we can remove the responses, and the system can leave the degraded mode.

Our main contributions are the following:

- We propose a novel intrusion survivability approach to withstand ongoing intrusions and maintain the availability of core functions of services (Section 4.2).
- We introduce a cost-sensitive response selection process to help select optimal responses (Chapter 5).
- We design an architecture and we develop a Linux-based prototype implementation (Chapter 6).
- We evaluate our prototype by measuring the effectiveness of the responses applied, the ability to select appropriate responses, the

availability cost of a checkpoint and a restore, the overhead of our solution, and the stability of the degraded services (Chapter 7).

4.2 APPROACH OVERVIEW

Since we focus our research on intrusion survivability, our work starts when an IDS detects an intrusion in a service.

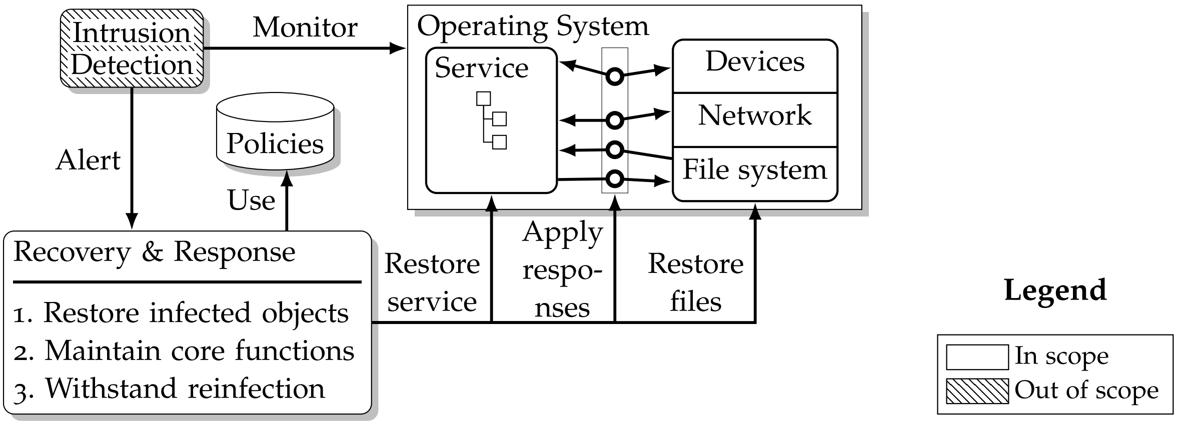

Intrusion Detection
Monitor
Operating System
Service
Devices
Network
File system
Policies
Alert
Use
Recovery & Response
1. Restore infected objects
2. Maintain core functions
3. Withstand reinfection
Restore service
Apply respo-nses
Restore files
Legend
In scope
Out of scope

Figure 4.1: High-level overview of our intrusion survivability approach

When the IDS detects an intrusion, we trigger a set of responses. The procedure must meet the following goals: restore infected objects (e.g., files and processes), maintain core functions, and withstand a potential reinfection. We achieve these goals using recoveries, responses, and policies.

RECOVERY Recovery actions restore the state of a service (i.e., the state of its processes and metadata describing the service) and associated files to a previous safe state. To perform recovery actions, we create periodic snapshots of the file system and the services, during the normal operation of the OS. We also log all the files modified by the monitored services. Hence, when restoring services, we only restore the files they modified. This limits the restoration time and it avoids the loss of known legitimate and non-infected data.[81] To perform recovery actions, we do not require for the system to be rebooted, and we limit the availability impact on the service.

RESPONSE A response action removes privileges, isolates components of the system from the service, or reduces resource quotas (e.g., CPU or RAM) of one service. Hence, it does not directly affect any other component of the system (including other services).[82] Its goal is to either prevent an attacker to reinfect the service or to withstand a reinfection by stopping attackers from achieving their goals (e.g., compromising data availability or integrity)

[81] Other files that the service depends on can be modified by another service, we handle such a case with dependencies information between services (see Section 6.1.5).

[82] However, if components depend on a degraded service, they can be affected indirectly.

after the recovery. Such a response may put the service into a degraded mode, because some functions might not have the required privileges anymore (or limited access to some resources).

POLICIES We apply appropriate responses that do not disable core functions (e.g., the ability to listen on port 80 for a web server). To refine the notion of core functions, we rely on policies. Their goal is to provide a trade-off between the availability of a function (that requires specific privileges) in a service and the intrusion risk. We designed a cost-sensitive response selection process based on such policies. Administrators, maintainers, and developers of the services can provide the policies by specifying the cost of losing specific privileges (i.e., if we apply a response) and the cost of a malicious behavior (exhibited by an intrusion).

4.3 THREAT MODEL AND ASSUMPTIONS

We make assumptions regarding the platform's firmware (e.g., BIOS or UEFI-compliant firmware) and the OS kernel where we execute the services. If attackers compromise such components at boot time or runtime, they could compromise the OS including our mechanisms. Hence, we assume their integrity. Such assumptions are reasonable in recent firmware using an hardware-protected root of trust [130, 230] at boot time and protection of firmware runtime services [60, 288, 290].[83] For the OS kernel, one can use UEFI Secure Boot [272, section 32] at boot time, and rely on e.g., security invariants [248] or a hardware-based integrity monitor [17] at runtime. The main threat that we address is the compromise of services inside an OS.

We make no assumptions regarding the privileges that were initially granted to the services. Some of them can restrict their privileges to the minimum. On the contrary, other services can be less effective in adopting the principle of least privilege. It depends on the design choices and the skills of the developers or maintainers of the services. The specificity of our approach is that we deliberately remove privileges that could not have been removed initially, since the service needs them for a function it provides. Finally, we assume that the attacker cannot compromise the mechanisms we use to checkpoint, restore, and apply responses (Section 6.1 details how we protect such mechanisms).

We model an attacker with the following capabilities:

- Can find and exploit a vulnerability in a service,
- Can execute arbitrary code in the same context as the compromised service,

[83] We address such issues in the Part III of this dissertation.

- Can perform some malicious behaviors even if the service had initially the minimum amount of privileges to accomplish its functions,
- Can compromise a privileged service or elevate the privileges of a compromised service to superuser,
- Cannot exploit software-triggered hardware vulnerabilities (e.g., side-channel attacks [154, 159, 177, 235]),
- Cannot perform hardware attacks (e.g., fault attacks [29]),
- Do not have physical access to the platform.

4.4 ILLUSTRATING EXAMPLES

We now describe different examples based on common malicious behaviors and capabilities [155, 200] that malware, or intrusions in general, could exhibit in a compromised service. For each example, we explain how our approach can survive it while maintaining the availability of the compromised service. The following lists, however, have no intention to be exhaustive.

We first discuss how our approach can thwart malicious behaviors that are the explicit goal of the intrusion.

COMPROMISE DATA AVAILABILITY Let us consider the case of encryption ransomware that infect vulnerable web servers [205]. Such malware renders files unavailable by encrypting them and then asking the owners to pay a ransom to obtain the decryption key. Depending on how the web server was initially confined, the damage can be limited only to the files that the web server should have access to, or it could affect the whole system. Using our approach, we can apply the following process: erase any trace of the malware in the service by restoring its state, restore the encrypted files, and put the service into a degraded mode by making some parts of the file system read-only for the web server, or block the access to some cryptographic APIs [150].

CONSUME SYSTEM RESOURCES Let us consider the case of cryptocurrency mining malware. The specificity of such malware is that it can consume all the CPU (or GPU) resources of the system. Hence, it can hinder other services that would be less responsive or unusable. In this case, for example, our approach can restore the service, and put it into a degraded mode by applying quotas (or limits) to the CPU usage of the service. Thus, if the service is compromised again by the cryptominer, the impact on the resources of the system will be limited. Only this service will suffer and could be less responsive.

DATA THEFT AND EXFILTRATION Let us consider an attacker using a compromised service to steal and exfiltrate confidential data. In this case, when the IDS detects the intrusion (e.g., using honeyfiles, traps, or decoys [31, 295]) our approach can restore the service and put it into a degraded mode by removing read access privileges from the service to a subset of the file system containing confidential data.

We now give examples of how our approach can instead survive an intrusion by targeting the malicious behaviors or capabilities which are needed or exhibited but are not the actual goal of the intrusion.

COMMAND AND CONTROL Let us consider malware remotely controlled by a Command and Control (C&C) server. The attacker might use the compromised host to send e-mail spam, participate in a Distributed Denial-of-Service (DDoS) attack, or run some cryptocurrency miner as we previously mentioned. In each case, the attacker needs to provide the targets or the payloads to the malware. Using our approach, we might not necessarily try to hamper the malware actual purpose (e.g., DDoS or cryptomining), but its ability to fetch or request information from the C&C to perform malicious actions. For example, our approach can restore the service and ban IP addresses associated with the C&C server (where only the service will have these IP banned but not the rest of the system).

PRIVILEGE ESCALATION Let us consider malware that rely on privilege escalation (e.g., brute forcing `su` or `sudo`). While this is not their primary goal, malware may elevate privileges before other steps. In this case, during the mitigation procedure, our approach can remove access to the program, API, or system call required to elevate privileges, thus stopping the next malicious steps.

5

COST-SENSITIVE RESPONSE SELECTION

For a given intrusion, multiple responses might be appropriate, and each one incurs an availability cost. We devised a cost-sensitive response selection process that minimize such a cost and maintain the core functions of the service.

We use a qualitative approach using linguistic constants (e.g., low or high) instead of a quantitative one (e.g., monetary values). Quantitative approaches require an accurate value of assets, and historical data of previous intrusions to be effective, which we assume missing. Qualitative approaches, while prone to biases and inaccuracies, do not require such data, and are easier to understand [279]. In addition, we would like to limit the input from the user so that it improves the approach's usability and its likelihood to be adopted in production.

In this chapter, we first describe the models that we rely on. Then, we detail how we select cost-sensitive responses using such models.

5.1 MODELS

In comparison to previous approaches, we do not introduce a model and a response selection based on vulnerability graphs, attack graphs, or similar [113, 151, 239, 241]. We consider that we do not know how the attacker managed to compromise the service at the moment. We also assume that we cannot predict the next steps of the attacker. Instead, we use a different paradigm where we consider that we have information about the characteristics of the intrusions, or the behaviors exhibited by the attacker.

5.1.1 *Malicious Behaviors and Responses*

Intrusions may exhibit multiple malicious behaviors that need to be stopped or mitigated differently. In our approach, we work at the level of a malicious behavior, and we select a response for each malicious behavior of an intrusion.

Our models rely on a hierarchy of malicious behaviors where the first levels describe high-level behaviors (e.g., compromise data availability), while lower levels describe more precise behaviors (e.g., encrypt files). The malware capabilities hierarchy [201] from the project Malware Attribute Enumeration and Characterization (MAEC) [155] of MITRE is a suitable candidate for such a hierarchy.[84] Figure 5.1 illustrates an example of a non-exhaustive malicious behavior hierarchy with behaviors that relates to availability violations and to C&C.

[84] Another project that can help is the MITRE ATT&CK knowledge base [199], but it does not provide a hierarchy.

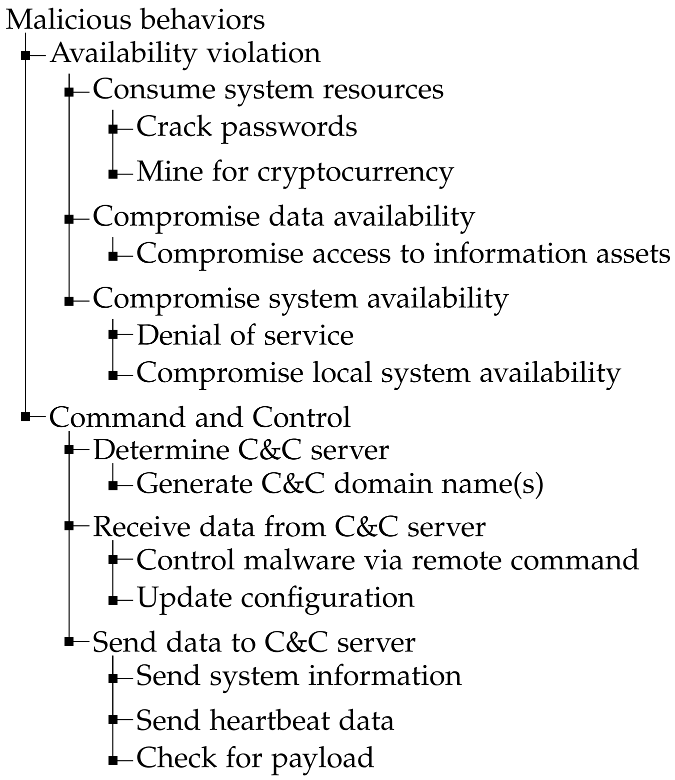

Malicious behaviors
Availability violation
Consume system resources
Crack passwords
Mine for cryptocurrency
Compromise data availability
Compromise access to information assets
Compromise system availability
Denial of service
Compromise local system availability
Command and Control
Determine C&C server
Generate C&C domain name(s)
Receive data from C&C server
Control malware via remote command
Update configuration
Send data to C&C server
Send system information
Send heartbeat data
Check for payload

Figure 5.1: Example of a non-exhaustive malicious behavior hierarchy

We model this hierarchy as a partially ordered set $(\mathbf{M}, \prec_{\mathbf{M}})$ with $\prec_{\mathbf{M}}$ a binary relation over the set of malicious behaviors $\mathbf{M}$. The relation $m \prec_{\mathbf{M}} m'$ means that m is a more precise behavior than m'. Let $\mathbf{I}$ be the space of intrusions reported by the IDS. We assume that for each intrusions $i \in \mathbf{I}$, we can map the set of malicious behaviors $M^i \subseteq \mathbf{M}$ exhibited by i. By construct, we have the following property: if $m \prec_{\mathbf{M}} m'$ then $m \in M^i \implies m' \in M^i$.

We also rely on a hierarchy of responses where the first levels describe coarse-grained responses (e.g., block the network), while lower levels describe more fine-grained responses (e.g., block port 80). We define the hierarchy as a partially ordered set $(\mathbf{R}, \prec_{\mathbf{R}})$ with $\prec_{\mathbf{R}}$ a binary relation over the set of responses $\mathbf{R}$ ($r \prec_{\mathbf{R}} r'$ means that r is a more fine-grained response than r'). Let $R^m \subseteq \mathbf{R}$ be the set of responses that can stop a malicious behavior m. By construct, we have the following property: if $r \prec_{\mathbf{R}} r'$ then $r \in R^m \implies r' \in R^m$. Such responses are based on the OS-features available to restrict privileges and quotas on the system.

Figure 5.2 is an example of this response hierarchy. Note that for each response with arguments (e.g., read-only paths or blacklisting IP addresses), the hierarchy provides a sub-response with a subset of the arguments. For example, if there is a response that puts `/var` read-only, there is also the responses that puts `/var/www` read-only.

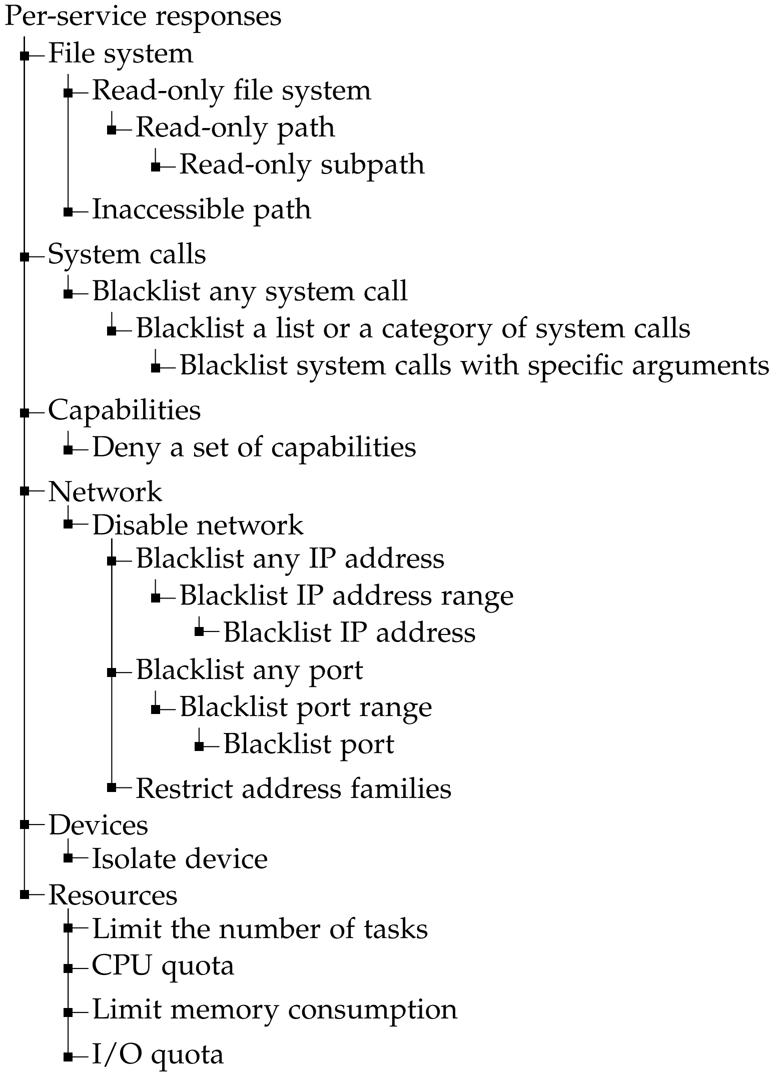

Per-service responses
File system
Read-only file system
Read-only path
Read-only subpath
Inaccessible path
System calls
Blacklist any system call
Blacklist a list or a category of system calls
Blacklist system calls with specific arguments
Capabilities
Deny a set of capabilities
Network
Disable network
Blacklist any IP address
Blacklist IP address range
Blacklist IP address
Blacklist any port
Blacklist port range
Blacklist port
Restrict address families
Devices
Isolate device
Resources
Limit the number of tasks
CPU quota
Limit memory consumption
I/O quota

Figure 5.2: Example of a non-exhaustive per-service response hierarchy

5.1.2 *Malicious Behavior Cost and Response Cost*

Let the space of services be denoted **S** and let the space of qualitative linguistic constants be a totally ordered set, denoted **Q** composed as follows: none < very low < low < moderate < high < very high < critical. We extend each service configuration file with the notion of malicious behavior costs and response costs (in terms of quality of service lost) that an administrator needs to set.

A malicious behavior cost $c_{mb} \in \mathbf{C_{mb}} \subseteq \mathbf{Q}$ is the qualitative impact of a malicious behavior $m \in \mathbf{M}$. We define $mbcost\colon \mathbf{S} \times \mathbf{M} \to \mathbf{C_{mb}}$, the function that takes a service, a malicious behavior, and returns the associated cost.

We require for each service that a malicious behavior cost is set for the first level of the malicious behaviors hierarchy (e.g., there are only 20 elements on the first level of the hierarchy from MAEC).

We do not require it for other levels, but if more costs are set, then the response selection will be more accurate. The *mbcost* function associates a cost for each malicious behavior m. The cost, however, could be undefined. In such a case, we take the cost of m' such that $mbcost(s, m')$ is defined, $m \prec_{\mathbf{M}} m'$, and $\nexists\, m''$ such that $m < m'' < m'$ with $mbcost(s, m'')$ defined.

For example, the policy of a web server w could express that any malicious behavior that violate availability has a high cost in general:

$$mbcost(w, \text{"availability-violation"}) = \text{"high"}$$

It means that malicious behaviors that are children of this node in the hierarchy also have their cost automatically defined, such as:

$$mbcost(w, \text{"compromise-data-availability"}) = \text{"high"}$$

We could consider that the cost of this malicious behavior does not need to be refined since an intrusion that compromise data availability (e.g., ransomware) have a high cost, since the web server would not provide access to the websites anymore. However, if not all availability violations have the same cost, they could be refined. For instance, the policy could express that an intrusion that only consumes system resources (e.g., a cryptocurrency mining malware) has a moderate cost since it would only slow down the system or the service. It would be expressed by overriding the cost for a more specific malicious behavior:

$$mbcost(w, \text{"consume-system-resources"}) = \text{"moderate"}$$

A response cost $c_r \in \mathbf{C_r} \subseteq \mathbf{Q}$ is the qualitative impact of applying a response $r \in \mathbf{R}$ on a service to stop a malicious behavior. We define $rcost : \mathbf{S} \times \mathbf{R} \to \mathbf{C_r}$, the function that takes a service, a response, and returns the associated response cost.

Response costs allow an administrator or developer of a service to specify how a response, if applied, would impact the overall quality of service. The impact can be assessed based on the number of functions that would be unavailable and their importance for the service. More importantly, with the value critical, we consider that a response would disable a core function of a service and thus should never be applied. Similarly, the cost could be undefined, hence we take the cost of r' such that $rcost(s, r')$ is defined, $r \prec_{\mathbf{R}} r'$, and $\nexists\, r''$ such that $r < r'' < r'$ with $rcost(s, r'')$ defined.

Following the same example, the policy could express that the network, and especially the ability to listen on ports 80 and 443 is critical for the core functions of the web server, while removing the ability to listen on other ports does not have any impact on the service:

$$\begin{aligned} rcost(w, \text{"network"}) &= \text{"critical"} \\ rcost(w, \text{"blacklist-any-port"}) &= \text{"none"} \\ rcost(w, \text{"blacklist-port-80"}) &= \text{"critical"} \\ rcost(w, \text{"blacklist-port-443"}) &= \text{"critical"} \end{aligned}$$

Similarly, having access to the file system is critical, but if the web server would lose write access to the file system, the cost could be high and not critical, since it can still provide access to websites for many use cases:

$$\begin{aligned} rcost(w, \text{"filesystem"}) &= \text{"critical"} \\ rcost(w, \text{"read-only-filesystem"}) &= \text{"high"} \end{aligned}$$

Both costs need to be configured depending on the *context* of the service. For example, a web server that provides static content does not have the same context, hence the same costs as one that handles transactions.

5.1.3 *Response Performance*

While responses have varying costs on the quality of service, they also differ in performance against a malicious behavior (i.e., their efficiency at stopping a specific malicious behavior). For example, making a subset of the file system read-only is less efficient to stop some ransomware than making the whole file system read-only. Hence, we consider the performance as a criterion to select a response, among others. The most effective response in terms of performance would be to stop the infected service. While our model allows it, in this work we only mention fine-grained responses that aim at maintaining the availability. Other work described in Section 3.3.3.3 already studied the use of such coarse-grained responses.

The space of qualitative response performances is denoted $\mathbf{P_r} \subseteq \mathbf{Q}$. We define $rperf \colon \mathbf{R} \times \mathbf{M} \to \mathbf{P_r}$, that takes a response, a malicious behavior, and returns the associated performance.

In contrast to the cost models previously defined that are specific to a system and its context (and need to be set, e.g., by an administrator of the system), such a value only depends on the malicious behavior and is provided by security experts that analyzed similar intrusions. This response performance come from threat intelligence sources that are shared, for example, using Structured Threat Information eXpression (STIX) [21].[85] For example, STIX has a property called "efficacy" in its "course-of-action" object that represent responses.

---

[85] Organizations such as the Information Technology - Information Sharing and Analysis Center (IT-ISAC) [140] or national Computer Emergency Response Teams (CERTs) [258] provide threat intelligence feeds to their members using STIX.

5.1.4 *Risk Matrix*

We rely on the definition of a risk matrix that satisfies the axioms proposed by Anthony Tony Cox [11] to provide consistent risk assessments: weak consistency, betweenness, and consistent coloring. While he defines these axioms more formally, we summarize them as follows:

WEAK CONSISTENCY Each risk qualified as high should have a quantitative risk higher than all risk qualified as low.

BETWEENNESS A small increase in confidence or in cost should not change the risk rating from low to high (there should always be an intermediate).

CONSISTENT COLORING Equal quantitative risks should have the same qualitative risk rating if either one of them is rated as high or low.

The risk matrix needs to be defined ahead of time by the administrator depending on the risk attitude of the organization: whether the organization is risk averse, risk neutral, or risk seeking. The 5×5 risk matrix shown in Table 5.1 is one instantiation of such a matrix.

	Malicious Behavior Cost				
Confidence (Likelihood)	Very low 0 – 0.2	Low 0.2 – 0.4	Moderate 0.4 – 0.6	High 0.6 – 0.8	Very high 0.8 – 1
Very likely 0.8 – 1	L	M	H	H	H
Likely 0.6 – 0.8	L	M	M	H	H
Probable 0.4 – 0.6	L	L	M	M	H
Unlikely 0.2 – 0.4	L	L	L	M	M
Very unlikely 0 – 0.2	L	L	L	L	L

Table 5.1: Example of a 5×5 risk matrix that follows the requirements for our risk assessment

The risk matrix outputs a qualitative malicious behavior risk $k \in \mathbf{K} \subseteq \mathbf{Q}$. The risk matrix depends on a malicious behavior cost (impact), and on the confidence level $i_{cf} \in \mathbf{I_{cf}} \subseteq \mathbf{Q}$ that the IDS has on the intrusion (likelihood).

We define *risk*: $\mathbf{C_{mb}} \times \mathbf{I_{cf}} \rightarrow \mathbf{K}$, the function representing the risk matrix that takes a malicious behavior cost, an intrusion confidence, and returns the associated risk.

5.1.5 *Policy Definition and Inputs*

Having discussed the various models we rely on, we can define the policy as a tuple of four functions $\langle rcost, rperf, mbcost, risk \rangle$. The *risk* function is defined at the organization level. It needs to be defined once, it is the same for each service, and unless the risk tolerance of the organization evolve it should not change. Moreover, $rperf$ is constant and can be applied for any system. However, *mbcost* and *rcost* are defined for each service depending on its context. Hence, the most time-consuming parameters to set are *mbcost* and *rcost*.

The function *mbcost* can be defined by someone that understands the impact of malicious behaviors based on the service's context (e.g., an administrator). *rcost* can be defined by an expert, a developer of the service, or a maintainer of the OS where the service is used, since they understand the impact of removing certain privileges to the service. For example, some Linux distributions provide the security policies (e.g., SELinux or AppArmor) of their services and applications. Much like SELinux policies, *rcost* could be provided this way, since the maintainers would need to test that the response do not render a service unusable (i.e., by disabling a core functionality).

5.2 OPTIMAL RESPONSE SELECTION

We now discuss how we use this policy to select cost-sensitive responses. Our goal is to maximize the performance of the response while minimizing the cost to the service. We rely on known Multi-Objective Optimization (MOO) methods [189] to select the most cost-effective response, as does other work on response selection [204, 241].

For conciseness, since we are selecting a response for a malicious behavior $m \in \mathbf{M}$ and a service $s \in \mathbf{S}$, we now denote $rperf(r, m)$ as p_r, $rcost(s, r)$ as c_r, and $mbcost(s, m)$ as c_{mb}.

5.2.1 *Overview*

When the IDS triggers an alert, it provides the confidence $i_{cf} \in \mathbf{I_{cf}}$ of the intrusion $i \in \mathbf{I}$ and the set of malicious behaviors $M^i \subseteq \mathbf{M}$ corresponding to this intrusion. Before selecting an optimal response, we filter out any response that have a critical response cost from R^m (the space of responses that can stop a malicious behavior m). Otherwise, such responses would impact a core function of the service. We denote $\hat{R}^m \subseteq R^m$ the resulting set:

$$\hat{R}^m = \{\, r \in R^m \mid c_r < \text{critical} \,\}$$

For each malicious behavior $m \in M^i$, we compute the Pareto-optimal set from $\hat{R}^m$, where we select an optimal response from. We now describe these last steps.

5.2.2 *Pareto-Optimal Set*

In contrast to a Single-Objective Optimization (SOO) problem, a MOO problem does not generally have a single global solution. For instance, in our case we might not have a response that provides both the maximum performance and the minimum cost, because they are conflicting, but rather a set of solutions that are defined as optimum. A common concept to describe such solutions is Pareto optimality.

A solution is Pareto-optimal (non-dominated) if it is not possible to find other solutions that improve one objective without weakening another one. The set of all Pareto-optimal solutions is called a Pareto-optimal set.[86] More formally, in our context, we say that a response is Pareto-optimal if it is non-dominated. A response $r \in R^m$ dominates a response $r' \in R^m$, denoted $r \succ r'$, if the following is satisfied:

$$[p_r > p_{r'} \land c_r \leq c_{r'}] \lor [p_r \geq p_{r'} \land c_r < c_{r'}]$$

MOO methods help to choose solutions among the Pareto-optimal set using preferences (e.g., should we put the priority on the performance of the response or on reducing the cost?) [189]. They rely on a scalarization that converts a MOO problem into a SOO problem. One common scalarization approach is the weighted sum method that assigns a weight to each objective and compute the sum of the product of their respective objective. However, this method is not guaranteed to always give solutions in the Pareto-optimal set [189].

Shameli-Sendi et al. [241] decided to apply the weighted sum method on the Pareto-optimal set instead of on the whole solution space to guarantee to have a solution in the Pareto-optimal set. We apply the same reasoning, so we reduce our set to all non-dominated responses. We denote the resulting Pareto-optimal set $\mathcal{O}$:

$$\mathcal{O} = \{\, r_i \in \hat{R}^m \mid \nexists r_j \in \hat{R}^m, r_j \succ r_i \,\}$$

5.2.3 *Response Selection*

Before selecting a response from the Pareto-optimal set using the weighted sum method, we need to set weights, and to convert the linguistic constants into numerical values.

We rely on a function l that maps the linguistic constants to a numerical value[87] between 0 and 1. In our case, we convert the linguistic constants critical, very high, high, moderate, low, very low, and none, to respectively the value 1, 0.9, 0.7, 0.5, 0.3, 0.1, and 0.

---

[86] Also known as a Pareto front.

[87] An alternative would be to use fuzzy logic to reflect the uncertainty regarding the risk assessment from experts when using linguistic constants [89].

We use the risk matrix to set the weight of the response performance:

$$\begin{aligned} k &= risk(c_{mb}, i_{cf}) \\ w_p &= l(k) \end{aligned}$$

Then, we set the weight of the response cost as follows:

$$w_c = 1 - w_p$$

This means that we prioritize the performance if the risk is high, while we prioritize the cost if the risk is low.

We obtain the final optimal response by applying the weighted sum method:

$$\arg\max_{r \in \mathcal{O}} w_p l(p_r) + w_c(1 - l(c_r))$$

6

ARCHITECTURE AND IMPLEMENTATION

The architecture and an implementation of our solution is discussed in this chapter. We provide the reasoning behind our design, what we require, and the choices we made to build our prototype.

6.1 ARCHITECTURE AND REQUIREMENTS

Our approach relies on four components. In this section, we first give an overview of how each component works and interacts with the others, as illustrated in Figure 6.1. Then, we detail requirements about our architecture.

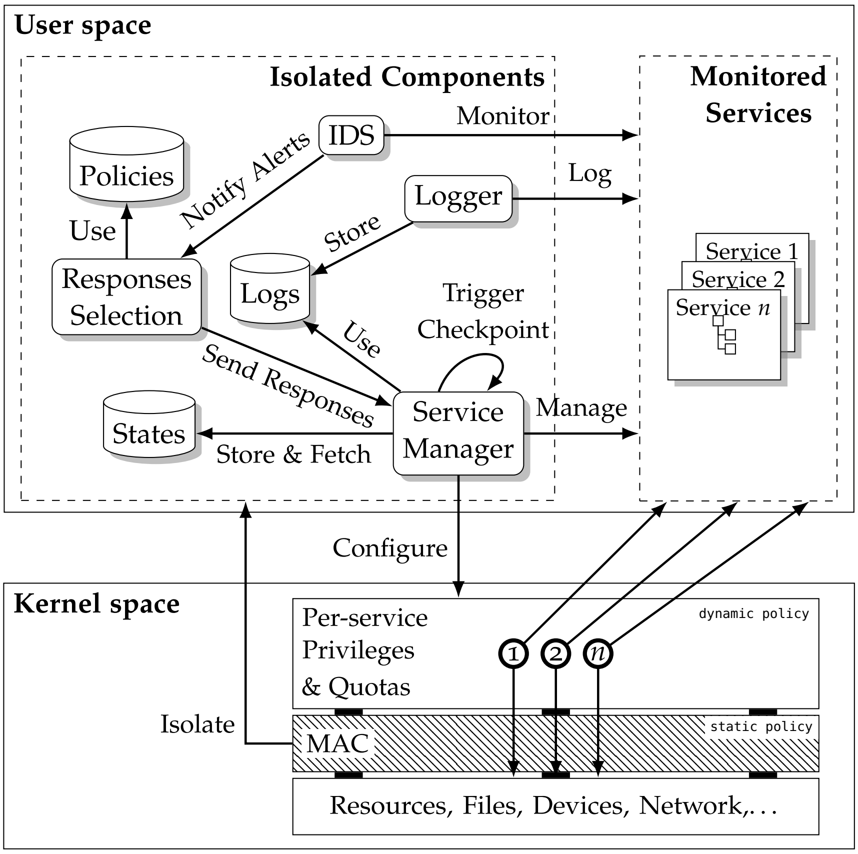

User space
Isolated Components
Monitored Services
IDS
Monitor
Policies
Notify Alerts
Logger
Log
Use
Store
Responses Selection
Logs
Service 1
Service 2
Service n
Trigger Checkpoint
Use
Send Responses
Service Manager
Manage
States
Store & Fetch
Configure
Kernel space
Per-service Privileges & Quotas
dynamic policy
①
②
n
Isolate
MAC
static policy
Resources, Files, Devices, Network,...

Figure 6.1: Overview of the architecture of our intrusion survivability approach

6.1.1 *Overview*

During the normal operation of the OS, the *service manager* creates periodic checkpoints of the services and snapshots of the file system. In addition, a *logging facility* logs the path of all the files modified by the monitored services since their last checkpoint. The logs are later used to filter the files that need to be restored.

The *IDS* notifies the *response selection* component when it detects an intrusion and specifies information about possible responses to withstand it. The selected responses are then given to the service manager. The service manager restores the infected service[88] to the last known safe state including all the files modified by the infected service. Then, it configures kernel-enforced per-service privilege restrictions and quotas based on the selected responses.

Finally, we rely on a static MAC policy to isolate our components. This policy is also enforced by the kernel, but in comparison to the per-service responses, it does not change over time.

6.1.2 *Last Known Safe State*

To select the last known safe state, we rely on the IDS to identify the first alert related to the intrusion and to provide us with a timestamp. Then, we consider that the first state prior to the timestamp of this alert is safe. In practice, however, the IDS might not be aware of the intrusion until a certain point in time. For example, if an intrusion happens at time t, but the IDS is only aware of this intrusion at time $t + 2$, we would select the state taken at $t + 1$ as the last known safe state. Therefore, we would restore the state of the service to an infected state.

Nevertheless, even if the restored state is infected because the IDS was not aware of the intrusion at the time, we apply responses. The per-service privilege restrictions and quotas that we apply stop the exploitation of a vulnerability (i.e., the attacker is not able to reinfect the service) or withstand a reinfection (i.e., attackers successfully reinfected the service, but they are restricted).

6.1.3 *Isolation of the Components*

For our approach to be able to withstand an attacker trying to impede the detection and recovery procedures, the integrity and availability of each component is crucial. Different solutions (e.g., a hardware isolated execution environment or a hosted hypervisor) could be used. In our case, we rely on a kernel-based MAC mechanism, such as SELinux [210], to isolate the components we used, as illustrated in Figure 6.1. Such a mechanism is available in commodity OSs, can express our isolation requirements, and does not modify the applications.[89]

[88] We could also take a snapshot just before restoring the infected service to a previous state. This snapshot would be helpful to perform further offline forensic analyses. Such an approach, however, was not implemented in our prototype.

[89] It also means that we can sustain a privileged attacker (e.g., with `root` privileges).

We now give guidelines on how to build a MAC policy to protect our components.

First, the MAC policy must ensure that none of our components can be killed.[90] Otherwise, e.g., if the responses selection component is not alive, no responses will be applied.

Second, the MAC policy must ensure that only our components have access to their isolated storage (e.g., to store the logs or checkpoints). Otherwise, attackers might e.g., erase an entry to avoid restoring a compromised file.

Third, the MAC policy must restrict the communication between the different components, and it must only allow a specific program to advertise itself as one of the components. Otherwise, an attacker might impersonate a component or stop the communication between two components. In our case, we assume a Remote Procedure Call (RPC) or an IPC mechanism that can implement MAC policies.For example, the D-Bus [83] IPC mechanism—which is adopted by most Linux distributions and used by many services—is SELinux-aware [275].

6.1.4 *Intrusion Detection System*

Our approach requires an IDS to detect an intrusion in a monitored service. We do not require a specific type of IDS. It can be external to the system or not. It can be misuse-based or anomaly-based. We only have two requirements.

First, the IDS should be able to pinpoint the intrusion to a specific service to apply per-service responses. For example, if the IDS analyzes event logs to detect intrusions, they should mention the service that triggered the event.

Second, the IDS should have information about the intrusion. It should map the intrusion to a set of malicious behaviors (e.g., the malware capabilities [201] from MAEC [155]), and it should provide a set of responses that can stop or withstand them. Both types of information can either be part of the alert from the IDS or be generated from threat intelligence based on the alert. Generic responses can also be inferred due to the type of intrusion if the IDS lacks precise information about the intrusion. For example, a generic response for ransomware consists in setting the file system hierarchy as read-only. Information about the alert, the responses, or malicious behaviors, are shared between members of threat intelligence sources [140, 258] using standards such as STIX [21] and MAEC [155, 200].

6.1.5 *Service Manager*

Commodity OSs rely on a service manager executed in user space (e.g., the Service Control Manager [231] for Windows, or systemd [253] for Linux distributions) to launch and manage services. In our architecture,

[90] One can also use a watchdog to ensure that the components are alive.

we rely on it, since it provides the appropriate level of abstraction to manage services and it has the notion of dependencies between services. Using such information, we can restore services in a coherent state. If a service depends on other services (e.g., if one service writes to a file and another one reads it), we checkpoint and restore them together.

We extend the service manager to checkpoint and restore the state of services. Furthermore, we modify the service manager so that it applies responses before it starts a recovered service. Since such responses are per-service, the service manager must have access to OS features to configure per-service privileges and resource quotas.

The service manager must be able to kill a service (i.e., all alive processes created by the service) in case it is compromised and needs to be restored. Therefore, we bound processes to the service that created them, and they must not be able to break the bound. For example, we can use cgroups [125] in Linux or job objects [193] in Windows.

The MAC policy must ensure that only the service manager can manage the collections of processes (e.g., `/sys/fs/cgroup` in Linux). Otherwise, if an attacker breaks the bound of a compromised service, it would be difficult to kill the escaped processes. Similarly, the MAC policy must protect configuration files used by the service manager.

Finally, the service manager must have the notion of dependencies between services. Using such information, provided by the administrator or the vendor, we can restore services in a coherent state. If a service depends on other services (e.g., if one service writes to a file and another one reads it), we checkpoint and restore them together. For example, with systemd, one has to state in the configuration file of a service the dependency requirements on other services with the directive `Requires=` followed by the name of the services [93]. If service A depends on service B, it needs to state `Requires=A.service`.

6.2 LINUX-BASED PROTOTYPE IMPLEMENTATION

We implemented a Linux-based prototype by modifying several existing projects. While our implementation relies on Linux features such as namespaces [149], seccomp [70], or cgroups [125], our approach does not depend on OS-specific paradigms. For example, on Windows, one could use Integrity Mechanism [196], Restricted Tokens [195], and Job Objects [193]. In the rest of this section, we describe the projects we modified, why we rely on them, and the different modifications we made to implement our prototype. You can see in Table 6.1 the different projects we modified where we added in total nearly 3600 lines of C code.

At the time of writing, the most common service manager on Linux-based systems is systemd [253]. We modified it to checkpoint and

Project	From version	Code added
CRIU	3.9	383 lines of C
systemd	239	2639 lines of C
audit		
user space	2.8.3	79 lines of C
Linux kernel	4.17.5	460 lines of C
Total		3561 lines of C

Table 6.1: Projects modified for the implementation of our intrusion survivability approach

to restore services using CRIU [76] and snapper [247], and to apply responses at the end of the restoration.

6.2.1 *Checkpoint and Restore*

CRIU is a checkpoint and restore project implemented in user space for Linux. We chose CRIU because it allows us to perform transparent checkpointing and restoring (i.e., without modification or recompilation) of the services.[91] It can checkpoint the state of an application by fetching information about it from different kernel APIs.[92] For each process that is checkpointed, CRIU collects information such as its memory mappings (`/proc/[pid]/maps`), the files it opened (`/proc/[pid]/fd` and `/proc/[pid]/fdinfo`), or other information about the process (`/proc/[pid]/stat`). Then, CRIU needs to gather information that only the process has access to. To achieve that, it injects "parasite code" inside the process address space and it makes the process execute that code.[93] Using this "parasite code", CRIU can dump the memory content or the system call filters for instance. CRIU stores all the information gathered during the checkpointing inside an image. To restore the application, CRIU reuses this image and use other kernel APIs to recreate the exact same state.

Snapper provides an abstraction for snapshotting file systems and handles multiple Linux file systems (e.g., BTRFS [227]). It can create a comparison between a snapshot and another one (or the current state of the file system). In our implementation, we chose BTRFS due its Copy-On-Write (COW) snapshot and comparison features, allowing a fast snapshotting and comparison process.

When checkpointing a service, we first freeze its cgroup (i.e., we remove the processes from the scheduling queue) to avoid inconsistencies. Thus, it cannot interact with other processes nor with the file system. Second, we take a snapshot of the file system and a snapshot of the metadata of the service kept by systemd (e.g., status informa-

[91] We also modified CRIU to use it as a bundled library inside systemd.

[92] It mainly relies on the `/proc` special file system that provides an access to many kernel data structures about processes.

[93] They rely on the `ptrace` system call that allows a privileged process to control the execution of another process.

tion). Third, we checkpoint the processes of the service using CRIU. Finally, we unfreeze the service.

When restoring a service, we first kill all the processes belonging to its cgroup. Second, we restore the metadata of the service and ask snapper to create a read-only snapshot of the current state of the file system. Then, we ask snapper to perform a comparison between this snapshot and the snapshot taken during the checkpointing of the service. It gives us information about which files were modified and how. Since we want to only recover the files modified by the monitored service, we filter the result based on our log of files modified by this specific service[94] and restore the final list of files. Finally, we restore the processes using CRIU. Before unfreezing the restored service, CRIU calls back our function that applies the responses. We apply the responses at the end to avoid interfering with CRIU that requires certain privileges to restore processes.

6.2.2 *Responses*

Our implementation relies on Linux features such as namespaces [149], seccomp [70, 104], and cgroups [125], to apply responses. Here is a non-exhaustive list of responses that our implementation supports:

- file system constraints (e.g., putting all or any part of the file system read-only),
- system call filters (e.g., blacklisting a list or a category of system calls),
- network socket filters (e.g., denying access to a range or a specific IP address),
- resource constraints (e.g., enforcing CPU quotas or limit memory consumption).

As mentioned previously, all the responses we apply are per-service. For example, if we put the file system read-only or deny access to a range of IP addresses, only the service that we target is impacted and degraded, none of the other processes or services on the system are impacted.

We modified systemd to apply most of these responses just before unfreezing the restored service, except for system call filters. Seccomp only allows processes to set up their own filters and prevent them to modify the filters of other processes. Therefore, we modified systemd so that when CRIU restores a process, it injects and executes code inside the address space of the restored process to set up our filters.

[94] See Section 6.2.3 for more details.

6.2.3 *Monitoring Modified Files*

The Linux auditing system [141, 260] is a standard way to trigger events from the kernel to user space based on a set of rules. Linux audit can trigger events when a process performs write accesses on the file system. However, it cannot filter these events for a set of processes corresponding to a given service (i.e., a cgroup). Hence, we modified the kernel side of Linux audit to perform such filtering in order to only log files modified by the monitored services. Then, we specified a monitoring rule that relies on such filtering.

We developed a userland daemon that listens to an audit netlink socket and processes the events generated by our monitoring rules. Then, by parsing them, our daemon can log which files a monitored service modified. To that end, we create a file hierarchy under a per-service private directory. For example, if the service `abc.service` modified the file `/a/b/c/test`, we create the following empty file `/private/abc.service/a/b/c/test`. This solution allows us to log modified files without keeping a data structure in memory.

6.2.4 *Bugs and Patches*

During the development of our prototype, we encountered various bugs in the projects we used. We reported them and contributed to several patches to fix them. We found a use-after-free in the Linux kernel audit code [53], parsing bugs in systemd [52, 57], missing APIs in CRIU [55, 56], and performance issues in snapper [54].

One of particular relevance was the one in snapper. When using BTRFS as a backend, and when trying to perform a comparison between two snapshots, snapper first tries to use the built-in BTRFS in-kernel features to perform a fast comparison. If this were to fail, snapper would fallback to using a slow userland operation that would call the system call `stat` to compare each file one by one. Unfortunately, snapper always failed to use the BTRFS features (due to a breaking change in the BTRFS library) and was always falling back to the slow method. Therefore, to perform a comparison, snapper would take around 30 s. We reported and fixed the bug; the operation now takes less than 300 ms.

7

EVALUATION AND RESULTS

We performed an experimental evaluation of our approach to answer the following questions:

1. How effective are our responses at stopping malicious behaviors in case a service is compromised?
2. How effective is our approach at selecting cost-sensitive responses that withstand an intrusion?
3. What is the impact of our solution on the availability or responsiveness of the services?
4. How much overhead our solution incurs on the system resources?
5. Do services continue to function (i.e., no crash) when they are restored with less privileges that they initially needed?

The rest of this chapter is structured as follows. First, we describe our experimental setup (Section 7.1). Second, we address security related questions 1 and 2 (Section 7.2). Third, we address performance related questions 3 and 4 (Section 7.3). Finally, we address the stability related question 5 (Section 7.4), before summarizing our results (Section 7.5).

7.1 EXPERIMENTAL SETUP

For the experiments, we installed Fedora Server 28 with the Linux kernel 4.17.5, and we compiled the programs with GCC 8.1.1. We ran the experiments that used live malware in a virtualized environment to control malware propagation.

The setup consisted of an isolated network connected to the Internet with multiple Virtual Local Area Networks (VLANs), two VMs, and a workstation. We executed the infected service on a VM connected to an isolated VLAN with access to the Internet. We connected the second VM, that executes the network sniffing tool (tcpdump), to another VLAN with port mirroring from the first VLAN. Finally, the workstation, connected to another isolated VLAN, had access to the server managing the VMs, the VM with the infected service, and the network traces.

While malware could use anti-virtualization techniques [51, 215], to the best of our knowledge, none of our samples used such techniques.[95] We executed the rest of the experiments on bare metal on

[95] This is consistent with the study of Cozzi et al. [75] that showed that in the 10 548 Linux malware they studied, only 0.24 % of them tried to detect if they were in a virtualized environment.

a computer with an AMD PRO A12-8830B R7 at 2.5 GHz, 12 GiB of RAM, and a 128 GB Intel SSD 600p Series.

Throughout the experiments, we tested our implementation on different types of services: web servers (nginx [206] and Apache [12]), database (mariadb [187]), work queue (beanstalkd [24]), message queue (mosquitto [103]), and git hosting services (gitea [120]).

7.2 SECURITY EVALUATION

In Section 7.2.1, we first evaluate how effective our responses and recovery are at withstanding an intrusion and reinfection. Then, we focus on how effective is our *selection* of responses in Section 7.2.2.

7.2.1 *Responses Effectiveness*

Our first experiments focus on how effective our responses against distinct types of intrusions are. We are not interested, per se, in the vulnerabilities that the attackers can exploit, but on how to stop attackers from performing malicious actions after they have infected a service.

The following list describes the malware[96] and attacks we used:

LINUX.BITCOINMINER Malware that connects to a mining pool using attackers-controlled credentials and mines cryptocurrency by using the resources of the system [267].

LINUX.REX.1 Malware that joins a Peer-to-peer (P2P) botnet to receive instructions to scan systems for vulnerabilities to replicate itself, elevate privileges by scanning for credentials on the machine, participate in a DDoS attack, or send spam [95].

HAKAI Malware that receives instructions from a C&C server to launch DDoS attacks, and to infect other systems by brute forcing credentials or exploiting vulnerabilities in routers [96, 207].

LINUX.ENCODER.1 Encryption ransomware that encrypts files commonly found on Linux servers (e.g., configuration files, or HTML files), and other media-related files (e.g., JPG, or MP3), while ensuring that the system can boot so that the administrator can see the ransom note [94].

GOAHEAD EXPLOIT Exploit that gives remote code execution to an attacker on all versions of the GoAhead [126] embedded web server prior to `3.6.5` [126].

Our work does not focus on detecting intrusions but on how to recover from and withstand them. Hence, we selected a diverse set of malware and attacks that covered different malicious behaviors, without selecting multiple malware with the same malicious behavior.

[96] See Appendix A for the hashes of the malware samples.

For each experiment, we start a vulnerable service, we checkpoint its state, we infect it, and we wait for the payload to execute (e.g., encrypt files). Then, we apply our responses and we evaluate their effectiveness. We consider the restoration successful if the service is still functioning and its state corresponds to the one that has been checkpointed. Finally, we consider the responses effective if we cannot reinfect the service or if the payload cannot achieve its goals anymore.

Attack Scenario	Malicious Behavior	Per-service Response Policy
Linux.BitCoinMiner	Mine for cryptocurrency	Ban mining pool IPs
Linux.BitCoinMiner	Mine for cryptocurrency	Reduce CPU quota
Linux.Rex.1	Determine C&C server	Ban bootstrapping IPs
Hakai	Receive data from C&C	Ban C&C servers' IPs
Linux.Encoder.1	Encrypt data	Read-only file system
GoAhead exploit	Exfiltrate via network	Forbid connect syscall
GoAhead exploit	Data theft	Render paths inaccessible

Table 7.1: Summary of the experiments that evaluate the effectiveness of the responses against various malicious behaviors

The experiments are summarized in Table 7.1. In each experiment, as expected, our solution successfully restored the service after the intrusion to a previous safe state. In addition, as expected, each response was able to withstand a reinfection for its associated malicious behavior and only impacted the specific service and not the rest of the system.

7.2.2 *Cost-Sensitive Response Selection*

Our second set of experiments focus on how effective is our approach at selecting cost-sensitive responses. We chose Gitea, a self-hosted Git-repository hosting service,[97] as a use case for a service, because it requires a diverse set of privileges and it shows how our approach can be applied to a complex real-world service.

In our use case, we configured Gitea with the principle of least privileges. It means that restrictions which corresponds to responses with a cost assigned to none are initially applied to the service (e.g., Gitea can only listen on port 80 and 443 or Gitea have only access to the directories and files it needs). Even if the service follows the best practices and is properly protected, an intrusion can still do damages (e.g., compromise the integrity of the repositories or the database) and our approach handles such cases.

We consider an intrusion that compromised our Gitea service with the `Linux.Encoder.1` ransomware.[98] When executed, it encrypts all the git repositories and the database used by Gitea. Hence, we configured

[97] Gitea is considered as an open source clone of the services provided by GitHub [121].

[98] In our experiments, we used an exploit for Gitea version 1.4.0 [255].

the policy to set the cost of such a malicious behavior to high,[99] since it would render the site almost unusable:

$$mbcost(\text{"gitea"}, \text{"compromise-data-availability"}) = \text{"high"}$$

Since our focus is not on intrusion detection, we assume that the IDS detected the ransomware. This assumption is reasonable with, for example, several techniques to detect ransomware such as API call monitoring, file system activity monitoring, or the use of decoy resources [150].[100]

However, in practice, an IDS can generate false positives, or it can provide non-accurate values for the likelihood of the intrusion leading to a less adequate response.[101] Hence, we consider three cases to evaluate the response selection: the IDS detected the intrusion and accurately set the likelihood, the IDS detected the intrusion but not with an accurate likelihood, and the IDS generated a false positive.

In Table 7.2, we display a set of responses for the ransomware. We devised this set based on existing strategies to mitigate ransomware, such as CryptoDrop lockdown [77] or Windows controlled folder access [194]. None of these responses could have been applied proactively by the developer of Gitea, because it degrades the quality of service. We set their respective cost for the service and the estimated performance. Finally, as a risk matrix, we use the one previously described in Table 5.1.

#	Response	Cost (c_r)	Performance (p_r)
1	Disable open system call	Very High	Very High
2	Read-only file system except sessions folder	High	Very High
3	Paths of git repositories inaccessible	High	Moderate
4	Read-only paths of all git repositories	Moderate	Moderate
5	Read-only paths of important git repositories	Low	Low
6	Read-only file system	Critical	Very High

Table 7.2: Responses to withstand ransomware reinfection with their associated cost and performance for Gitea

Now let us consider the three cases previously mentioned. In the first case, the IDS detected the intrusion and considered the intrusion very likely. After computing the Pareto-optimal set, we have three possible responses left (2, 4, and 5). The risk computed is $risk(\text{"high"}, \text{"very likely"}) = \text{high}$. The response selection then prioritizes performance and selects the response 2 that sets the file system as read-only except the session folder. This protects all information stored by Gitea (git repositories and its database). The session folder remains writable since having this folder read-only would render the site unusable, thus it is a core function (see the cost critical in Table 7.2). Gitea is restored with all the encrypted files. The selected response prevents the attacker to reinfect the service since the exploit require

[99] One would have to assign a cost for other malicious behaviors, but for the sake of conciseness we only show the relevant ones.

[100] For more details, see the study of Kharraz et al. [150] on ransomware attacks and their suggestions to detect them.

[101] If the IDS does not provide confidence metrics, the organization or an administrator could define one by default that would need to be defined once.

write accesses. In terms of quality of service, users can connect to the service and clone repositories, but due to the response a new user cannot register and users cannot push to repositories. Hence, this response is adequate since the service cannot get reinfected, core functions are maintained, and other functions (e.g., users can log in) are available.

In the second case, the IDS detected the intrusion but considered the intrusion very unlikely while the attacker managed to infect the service. The risk computed is $risk(\text{"high"}, \text{"very unlikely"}) = \text{low}$. The response selection then prioritizes cost and selects the response 5 that sets a subset of git repositories (the most important ones for the organization) as read-only. With this response, the attacker managed to reinfect the service and the ransomware encrypted many repositories, but not the most important ones. In terms of quality of service, users can still access the protected repositories, but due to the intrusion users cannot login anymore and they cannot clone the encrypted repositories (i.e., Gitea shows an error to the user). Hence, the response is less adequate when the IDS provides an incorrect value for the likelihood of the intrusion, since the malware managed to encrypt many repositories, but the core functions of Gitea are maintained.

In the third case, the IDS detected an intrusion with the likelihood being very likely, but it is in fact a false positive. The risk computed is $risk(\text{"high"}, \text{"very likely"}) = \text{high}$. It is similar to the first case where the response selected is response 2 due to a high risk. However, in this case, there is no actual ransomware. In terms of quality of service, users still have access to the service, since they can access the site, they can log in, and they can clone all repositories. However, they cannot register, they cannot push modifications to the repositories, and they cannot add issues. It shows that even with false positives, our approach minimizes the impact on the quality of service. Once an analyst classified the alert as a false positive, the administrator can configure the service to leave the degraded mode by deactivating the response.[102]

7.3 PERFORMANCE EVALUATION

Having evaluated and discussed the effectiveness of our approach at withstanding intrusions, we now focus on the availability and overall performance impact of our solution.

7.3.1 *Availability Cost*

In this subsection, we detail the experiments that evaluate the availability cost for the checkpoint and restore procedures.

Each time we checkpoint a service, we freeze its processes. As a result, a user might notice a slower responsiveness from the service. Hence, we measured the time to checkpoint different services: Apache HTTP server (v2.4.33), nginx (v1.12.1), mariadb (v10.2.16), and beanstalkd (v1.10). We repeated the experiment 10 times for each service. In average the time to checkpoint was always below 300 ms. In Table 7.3, we show more detailed timings about the different operations executed during a checkpoint: initialize (i.e., to initialize structures, to create or open directories, and to freeze processes), snapshot the file system, serialize the service's metadata, and checkpoint the processes using CRIU. We see that the time to perform this last operation varies depending on the service while other operations are not dependent on the service. It is related to the resources used by the service (e.g., the number of processes, memory used, or files opened): more resources used means more time spent by CRIU to checkpoint them.

---

[102] In this case, technically speaking, we would deactivate the response by remounting the directories as writable in the namespace of the service. However, as mentioned in Section 8.3, other responses might be more difficult to deactivate.

Checkpoint Operation		Mean	Standard deviation	Standard error of the mean
Service-independent operations				
Initialize	(µs)	643.20	90.75	14.35
Checkpoint service metadata	(µs)	51.47	8.45	1.33
Snapshot file system	(ms)	98.95	1.38	2.19
Checkpoint processes (CRIU)				
httpd	(ms)	199.24	11.05	3.49
nginx	(ms)	51.59	3.99	1.26
mariadb	(ms)	171.77	8.52	2.69
beanstalkd	(ms)	16.25	1.37	0.43
Total				
httpd	(ms)	298.88		
nginx	(ms)	151.24		
mariadb	(ms)	271.41		
beanstalkd	(ms)	115.89		

Table 7.3: Time to perform the checkpoint operations of a service

In Figure 7.1, we illustrate the results of the availability cost that users could perceive by measuring the latencies of HTTP requests made to an nginx server. We generated 100 requests per second for 20 seconds with the HTTP load testing tool Vegeta [237]. During this time, we checkpointed nginx at approximately 5, 11, and 16 seconds. We repeated the experiment three times. The output gave us the latency of each request, and we applied a moving average filter with a window size of 5. All requests were successful (i.e., no errors or timeouts) and the maximum latency during a checkpoint was 286 ms.

Both results show that our checkpointing has a small, but acceptable availability cost. We do not lose any connection, we only increase the

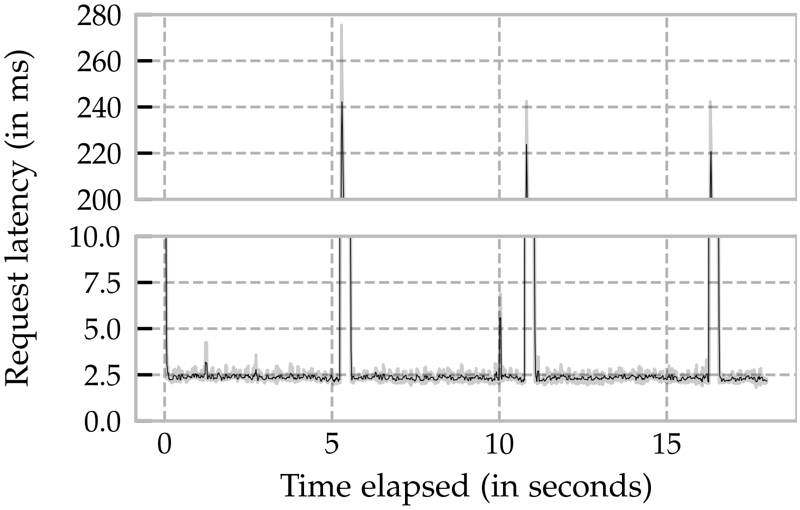


Figure 7.1: Impact of checkpoints on the latency of HTTP requests made to an nginx server (less is better)

requests' latency when the service is frozen. Since the latency increases only for a small period of time (maximum 300 ms), we consider such a cost acceptable. In comparison, SHELF [286] incurs a 7.6 % latency overhead for Apache during the whole execution of the system.

We also evaluated the time to restore the same services. In average, it took less than 325 ms. In Table 7.4, we show more detailed timings about the different operations executed during a restore: initialize (i.e., to initialize structures and to open directories), kill the processes, compare the snapshots of the file system, restore the service's metadata, and restore the processes using CRIU. When restoring, the time to kill the processes is service-dependent due to the different processes used and their number. We also see that the operation related to the comparison of the snapshots prior to restoring infected files takes a significant portion of the time.[103]

In contrast to the checkpoint, the restore procedure loses all network connections since we kill the processes before restoring them.[104] The experiments, however, show that the time to restore a service is small (less than 325 ms). For example, in comparison, CRIU-MR [277] took 2.8 s in average to complete their restoration process.

7.3.2 *Monitoring Cost*

As detailed in Section 6.2.3, our solution logs the path of any file modified by a monitored service. This monitoring, however, incurs an overhead for every process executing on the system—even for the non-monitored services.[105] There is also an additional overhead for monitored services that perform write accesses due to the audit event generated by the kernel and then processed by our daemon.

[103] Note that in this experiment, we performed a checkpoint then directly afterward we restored the service, it means the service did not modify files during the procedure. Hence, we only evaluated the time to compare snapshots and not the time it takes to effectively restore files. The time it takes to restore the files would vary depending on their number and their size.

[104] It could be possible to implement a method that retains packets in a buffer during the operation to avoid losing them, as implemented by CRIU-MR [277].

[105] For example, checking whether a process is part of the monitored services adds additional operations in the kernel.

Restore Operation		Mean	Standard deviation	Standard error of the mean
Kill processes				
httpd	(ms)	16.39	2.52	1.13
nginx	(ms)	19.24	3.69	1.65
mariadb	(ms)	28.48	2.16	0.97
beanstalkd	(ms)	10.85	1.19	0.53
Service-independent operations				
Initialize	(µs)	209.40	32.07	7.17
Compare Snapshots	(ms)	148.23	32.01	7.16
Restore service metadata	(µs)	212.75	36.23	8.10
Restore processes (CRIU)				
httpd	(ms)	132.42	6.09	2.72
nginx	(ms)	59.88	4.88	2.18
mariadb	(ms)	147.07	2.59	1.16
beanstalkd	(ms)	36.63	2.87	1.28
Total				
httpd	(ms)	299.29		
nginx	(ms)	227.79		
mariadb	(ms)	324.22		
beanstalkd	(ms)	196.16		

Table 7.4: Time to perform the restore operations of a service

Therefore, we evaluated the runtime overhead due to the monitoring by running synthetic and real-world workload benchmarks, from the Phoronix test suite [164], for three different cases:

1. no monitoring is present (**baseline**),
2. monitoring rule enabled, but the service running the benchmarks **is not monitored** (no audit events are triggered),
3. monitoring rule enabled and the service **is monitored** (audit events are triggered).

7.3.2.1 *Synthetic Benchmarks*

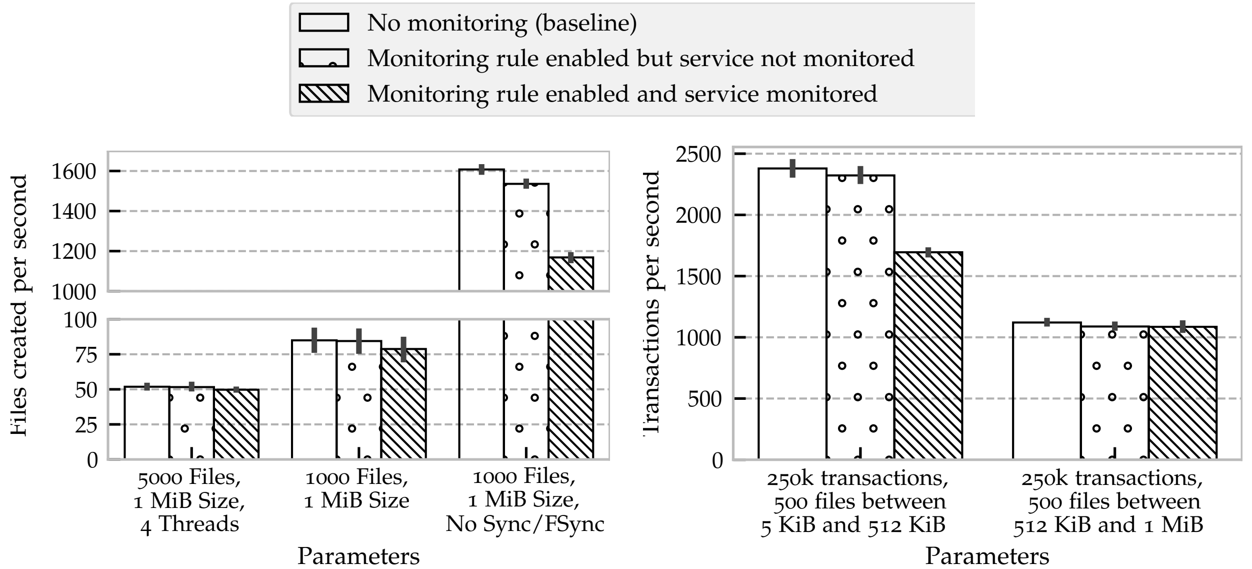
We ran synthetic I/O benchmarks that stress the system by performing many open, read, and write system calls:

COMPILEBENCH It emulates disk I/O operations related to the compilation of a kernel tree, reading the tree, or its creation [190].

FS-MARK It creates files and directories, at a given rate and size, either synchronously or asynchronously [280].

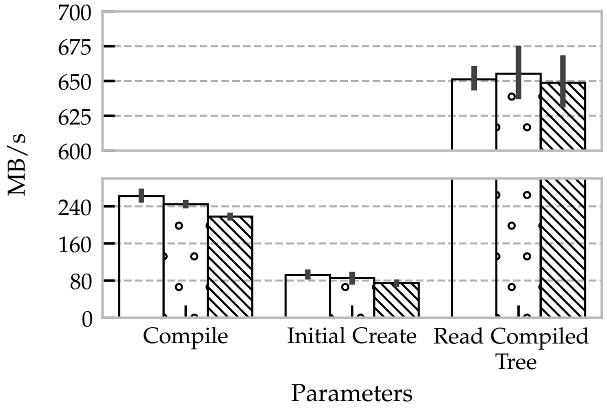
POSTMARK It emulates an email server by performing a given number of transactions that create, read, append to, and delete files of varying sizes [147].

The results of the `read compiled tree` test of the `compilebench` benchmark (Figure 7.2c) confirmed that the overhead is only due to `open` system calls with write access mode. This test only reads files, and we do not observe any noticeable overhead (less than 1 %, within the margin of error).



(a) Files created per second with the fs-mark benchmark (more is better)

(b) Transactions per second with the Postmark benchmark (more is better)



(c) MB/s score with the Compilebench benchmark (more is better)

Figure 7.2: Results of synthetic benchmarks to measure the overhead of the monitoring

We now focus on the results of the `fs-mark` and `Postmark` benchmarks, respectively illustrated in Figure 7.2a and Figure 7.2b. In both experiments, we notice a small overhead when the service is not monitored (between 0.6 % and 4.5 %). With `fs-mark` (Figure 7.2a), when writing 1000 files synchronously, we observe a 7.3 % overhead. In comparison, when the files are written asynchronously, there is a 27.3 % overhead. With `Postmark` (Figure 7.2b), we observe that the overhead is quite important (28.7 %) when it writes many small files (between 5 KiB and 512 KiB) but remains low (3.1 %) with bigger files (between 512 KiB and 1 MiB).

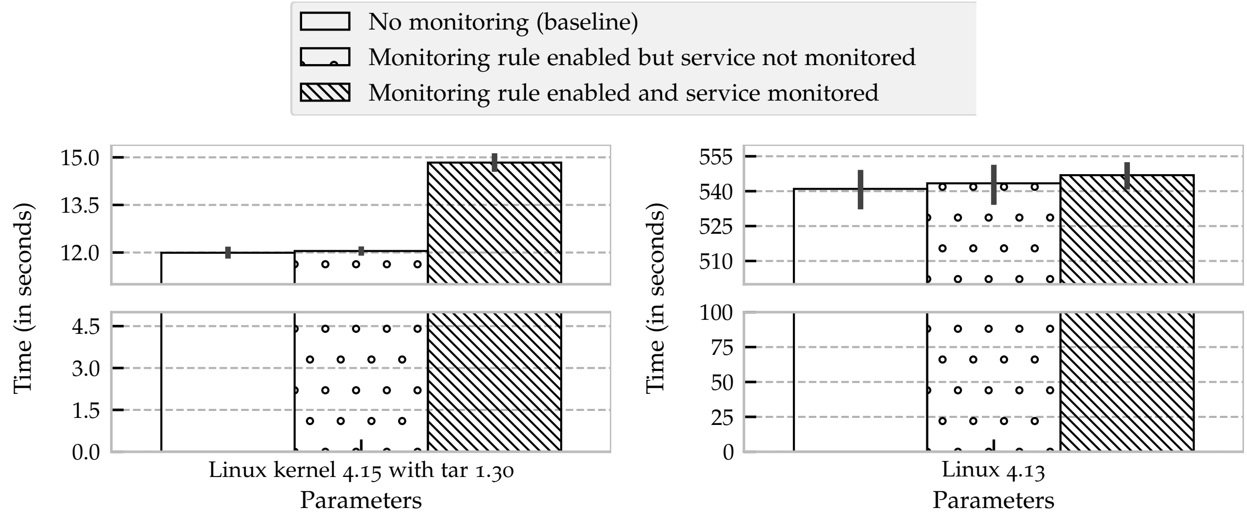
In summary, these synthetic benchmarks show that the worst case for our monitoring is when a monitored service writes many small files asynchronously in burst.

7.3.2.2 *Real-world Workload Benchmarks*

To have a different perspective than the synthetic benchmarks, we chose two benchmarks that use real-world workloads:

BUILD-LINUX-KERNEL It measures the time to compile the Linux kernel.[106]

UNPACK-LINUX It measures the time to extract the archive of the Linux kernel source code.



(a) Time (in seconds) to extract the archive (.tar.gz) of the Linux kernel source code (less is better)

(b) Time (in seconds) to build the Linux kernel (less is better)

Figure 7.3: Results of real-world workload benchmarks to measure the overhead of the monitoring

We illustrate the results in Figure 7.3. When the service is monitored, the overhead is only significant with `unpack-linux` (Figure 7.3a) where we observe a 23.7 % overhead. It concurs with our results from the synthetic benchmarks: writing many small files asynchronously incurs a significant overhead when the service is monitored (the time to decompress a file in this benchmark is negligible). With `build-linux-kernel` (Figure 7.3b), we observe a small overhead (1.1 %) even when the service is monitored (the time to compile the source code masks the overhead of the monitoring).

[106] While `build-linux-kernel` is CPU bound, it also performs many system calls, such as opening files to store the output of the compilation.

In comparison, SHELF [286] has a 65 % overhead when extracting the archive of the Linux kernel source code, and an 8 % overhead when building this kernel.

In conclusion, both the synthetic and non-synthetic benchmarks show that our solution is more suitable for workloads that do not write many small files asynchronously in burst. For instance, our approach would be best suited to protect services such as web, databases, or video encoding services.

7.3.3 *Storage Space Overhead*

Checkpointing services requires storage space to save the checkpoints. To evaluate the disk usage overhead, we checkpointed the same four services used in Section 7.3.1. Each checkpoint took respectively 26.2 MiB, 7.5 MiB, 136.0 MiB, and 130.1 KiB of storage space. The memory pages dumps took at least 95.3 % of the size of their checkpoint. Hence, if a service uses more memory under load (e.g., Apache), its checkpoint would take more storage space.

7.4 STABILITY OF DEGRADED SERVICES

We tested our solution on diverse types of services: web servers (nginx, Apache), databases (mariadb), work queues (beanstalkd), message queues (mosquitto), or git hosting services (gitea). In terms of restoration, none of the services crashed when restored with a policy that removed privileges that they required (i.e., when they are in a degraded mode). The reason is twofold.

First, we provided a policy that specified the responses with a critical cost. Therefore, our solution never selected a response that removes a privilege needed by a core function. Second, the services checked for errors when performing various operations. For example, if a service needed a privilege that we removed, it tried to perform the operation and failed, but only logged an error and did not crash. If we generalize our results, it means our solution will not make other services (that we did not test) crash if they properly check for error cases. This practice is common, and it is often highlighted by the compiler or static analysis tools when this is not the case.[107]

7.5 SUMMARY

Our evaluation shows that our approach is applicable to a diverse set of services and it was able to withstand different types of intrusions. The services do not crash when they are in a degraded mode. The checkpointing does not lose any network connection, or other information. We do, however, freeze the service, thus inducing an increase in requests' latency (as high as 300 ms). The restoration procedure does

[107] For example, the CERT C Coding Standard recommends not to ignore values returned by functions, and it provides a list of tools that automatically detect any violation of this recommendation [46]. It also recommends "implement[ing] a consistent and comprehensive error-handling policy" [45] with an example of a tool to help detect when this is not the case.

lose the network connections of the service, but the time to restore the service is still under 325 ms. Finally, our evaluation of the monitoring cost shows that our solution is suitable for CPU-intensive services, and I/O-intensive services, except for rare cases where services create many small files asynchronously. In the latter case, the overhead can be as high as 28.7 %.

8

CONCLUDING REMARKS

Having demonstrated the effectiveness of the responses, and showed the performance impact of our prototype, we now discuss the limitations of our approach, we give a quick summary of the comparison of our approach against the related work, and we conclude Part II while providing some potential areas to investigate in the future.

8.1 DISCUSSION AND LIMITATIONS

We discuss non-exhaustively limitations, areas that would need further work, or alternative choices for our solution.

FALSE POSITIVES Since our approach relies on an IDS, we also inherit the limitations of this IDS. It is possible that we start the recovery and response procedures due to a false positive from the IDS. In this case, it will negatively impact the service's availability and its functions, despite thwarting no threat. Our approach, however, minimizes this risk by considering the likelihood of the intrusion for the selection of cost-sensitive responses and by ensuring that core functions are maintained.

CRIU LIMITATIONS At the moment, CRIU cannot support all types of applications, since it has issues when handling external resources or graphical applications. For example, if a process has opened a device to have direct access to some hardware, checkpointing the state of the process may not be possible (except if it uses virtual or pseudo devices not corresponding to any physical devices). This technical limitation is because CRIU cannot be sure that when it restores a process, the physical device has the same state as when CRIU checkpointed the process. Since our implementation relies on CRIU, we inherit its limitations. Therefore, at the moment, our implementation is better suited for system services that do not have a graphical part and do not require direct access to some hardware.

ALTERNATIVE CHECKPOINTING An alternative to our transparent checkpointing with CRIU would be to implement a cooperative checkpointing approach where the service itself saves its state and uses it to restore itself after the IDS detected an intrusion. On one hand, such an approach would require changes in each service by adding a procedure to dump their state. For example, Android applications have such methods [10]. On the other hand, we would be able to apply more fine-grained mitigations, because the restoration procedure would have semantic information and knowledge of the data structures used by each service. For example, as a more fine-grained mitigation, we could change the implementation of a protocol.

SERVICE DEPENDENCIES In our work, at the moment, we only use the service dependency graph provided by the service manager (e.g., systemd) to recover and checkpoint dependent services together. We could also use this same graph to provide more precise response selection by taking into account the dependency between services, their relative importance, and to propagate the impact a malicious behavior can have. It could be used as a weight (in addition to the risk) to select optimal responses. Similar, but network-based, approaches have been heavily studied in the past [152, 241, 266].[108]

STATE INCONSISTENCIES Let us consider a service that does not follow properly the principle of least privilege with unnecessary write access to various files on the system. This service gets compromised and an attacker compromise files on the system that other services depends on. If we restore the files after the intrusion is detected, it could result in state inconsistencies in the services that depends on these files. Indeed, since we do not have information about which services use or depend on these files, we cannot restore their processes as well. It is the result of the trade-off we initially made where we do not monitor every event on the system to limit the performance overhead. More work would be needed to maintain the low overhead while improving the reliability in such cases. For example, we could use the information available during the installation of a service[109] to know a subset of the files it depends on, or we could ask maintainers to list the directories or files it relies on.

MODELS INPUT For our cost-sensitive response selection, we first need to associate an intrusion to a set of malicious behaviors, and the course of action to stop these behaviors. While standards exist to share threat information [21] and malicious behaviors [155, 200, 201] exhibited by malware, or attackers in general, we were not able to find open sources that provided them directly for the samples we used. This issue might be related to the fact that, to the best of our knowledge, no industry solution would exploit such information in an automated fashion. In our experiments, we extracted information about malicious behaviors from textual descriptions [94–96, 207, 267] and reused the existing standards to describe such malicious behaviors [155, 200, 201]. Likewise, we extracted information about responses to counter such malicious behaviors from textual descriptions [77, 94–96, 194, 207, 267].

[108] The host-based approach from Balepin et al. [20] requires administrators to give dependency information between applications and the resources they use on the system. While it may be more precise than service dependencies, it is a tedious, error prone process, and it introduces a maintenance burden.

[109] For example, Linux distributions rely on packages to install services. Each package contains the files to install on the system. In addition, it may contain instructions to create directories or files that the service requires. We could use both information to improve the reliability of the recovery in the case we discussed.

One may, nonetheless, assume that if approaches similar to ours, or intrusion response systems in general, are becoming more prevalent and used in production, threat intelligence sources will eventually provide such data.

GENERIC RESPONSES If we do not have precise information about the intrusion, but only a generic behavior or category associated to it (one of the top elements in the malicious behaviors hierarchy), we could automatically consider generic responses. For example, with ransomware we know that responses that either render the file system read-only or only specific directories will work. Such generic responses might help mitigate the lack of precise information.

8.2 COMPARISON WITH RELATED WORK

Our approach addresses various issues that we identified from the literature.[110] We summarize in Table 13.1 a comparison of our intrusion survivability approach against previous related work.

	Approach									
Characteristics	Our approach	SHELF [286]	CRIU-MR [277]	Taser [122]	Retro [153]	Shan et al. [242]	Foo et al. [113]	Balepin et al. [20]	Shameli-Sendi et al. [241]	Gehani and Kedem [119]
Recover from intrusions	●	●	●	●	●	●	○	◒	○	○
Withstand (re)infections	●	○	○	○	○	○	●	●	●	●
Maintain availability	●	●	●	○	○	○	◒	◒	◒	●
Host-based approach	●	●	●	●	●	●	○	●	○	●
Transparent	●	●	●	●	●	●	●	●	●	○
Fine-grained responses	●	○	○	○	○	○	○	○	○	●
Use service dependency graphs	◒	○	○	○	○	○	○	●	●	○
Quantitative risk assessment	○	○	○	○	○	○	○	●	●	●
Qualitative risk assessment	●	○	○	○	○	○	○	○	○	○

Table 8.1: Summary of the comparison between our intrusion survivability approach and the related work

Legend

● Has the property
○ Does not have the property
◒ Partially

To the best of our knowledge, our approach is the first to combine the ability to recover from intrusions with the ability to withstand potential reinfections after the system has been restored. Balepin et al. [20] had the ability to recover a file from a backup as a response,

[110] See Section 3.3 for more details about the state of the art related to our work.

but they assume that the IDS knows precisely which file has been compromised (and did not described how they were taking a backup of the files nor the performance impact). In practice, we might not have an accurate view of the illegitimate actions that have been carried out by the attacker—we can overestimate them or miss some of them.[111] In our approach, we make a trade-off between precision and performance by restoring all the files modified by a compromised service.[112] In addition, Balepin et al. [20] do not maintain the consistency between the state of the processes and the files they depend on. In our approach, to reduce the possibility of inconsistencies, we first freeze the processes. Then, we take simultaneously a checkpoint of both the file system and the state of the processes.

The fact that we can maintain the availability of the services and their core functions despite the presence of an adversary means that the OS can survive intrusions. Previous work on intrusion response [20, 113, 240] considered the cost of a response on the availability only in their models. Some of their responses, however, would significantly impact the availability without significant benefit for the security. For example, Shameli-Sendi et al. [240] has responses that reboot completely a system or restart a compromised service. In our approach, in addition to taking into account the cost of a response, our ability to restore the state of a service to a previous safe state minimize the availability impact.[113] Finally, in comparison to previous work, since we apply per-service responses we minimize the impact of a response on the rest of the system.

8.3 CONCLUSION AND FUTURE WORK

The work presented in this part of the dissertation introduces an intrusion survivability approach for commodity OSs. In contrast to other intrusion recovery approaches, our solution is not limited to restoring files or processes, but it also applies responses to withstand a potential reinfection. Such responses enforce per-service privilege restrictions and resource quotas to ensure that the rest of the system is not directly impacted. In addition, we only restore the files modified by the infected service to limit the restoration time. We devised a way to select cost-sensitive responses that do not disable core functions of services. We specified the requirements for our approach and proposed an architecture satisfying its requirements. Finally, we developed and evaluated a prototype for Linux-based systems by modifying systemd, Linux audit, CRIU, and the Linux kernel. Our results show that our prototype withstands known Linux attacks. Our prototype only induces a small overhead, except with I/O-intensive services that create many small files asynchronously in burst.

The initial idea behind this work has been presented at RESSI in 2018 [59], then the work was finally published and presented at

[111] We saw in Section 3.3.2.3 some intrusion recovery approaches that tried to address these issues—by limiting the number of false positives and false negatives—while introducing performance issues.

[112] We assume that we might not know all the illegitimate actions that were done.

[113] Restoring to a previous state also allows us to implement responses that would be more difficult otherwise (e.g., additional system call filters).

ACSAC in December 2019 [58]. For future work, we would like to investigate the following areas:

RESPONSE DEACTIVATION At some point the response we applied that degrade the service's state should be disabled (e.g., if a patch fixing a vulnerability has been applied).[114] We would like to implement such ability, but it might raise some challenges. For instance, for some responses on our Linux-based prototype, such as system call filters, it would be difficult to deactivate them without doing a checkpoint directly followed by a restore.[115] System call filters cannot be disabled once set, so we would need to recreate the current state of the service while removing some system call filters in the image of the processes.

DYNAMIC RESPONSE ADAPTATION At the moment we rely on experts to assess the performance of a response. This assessment, however, might be biased or wrong. We would like to add the ability of our approach to adapt the selection of responses based on the history of their success or failure [239, 252]. Our approach would still by default rely on expert assessments,[116] but it would use an additional weight to select a response if there is a success rate available for a response.

HARDWARE-BASED ISOLATION AND ENFORCEMENT We trust the kernel to isolate our components from the attacker. In addition, we trust the kernel to enforce our policies and responses. Unfortunately, as mentioned in Section 3.3.1, kernel vulnerabilities exist, have a long life span, and could be used to bypass our solution. We would like to investigate if and how we could change our architecture to execute some of our components in a hardware-based isolated environment without introducing a semantic gap or a maintenance burden.

[114] Shameli-Sendi et al. [239] noticed that this capability is rarely present among intrusion response systems.

[115] Much like CRIU-MR [277].

[116] This history might rarely be available for a response if we lack sensors to evaluate its success, and we also do not have data at the beginning.

Part III

DETECTING INTRUSIONS AT THE FIRMWARE LEVEL

9

INTRODUCING A SMM BEHAVIOR MONITORING APPROACH

In the previous part, we described our approach on how to survive intrusions at the OS-level. Before being able to survive, however, one must be able to detect an intrusion. As mentioned in Section 1.1.3, there are many solutions to detect intrusions at the OS-level. On the other hand, at the firmware level, there are fewer solutions available to detect intrusions or attacks targeting the low-level components of a platform. Therefore, in this part, we focus on intrusion detection at the firmware level.

Throughout this part, we use the BIOS as a use case. Indeed, it is the low-level software component of the platform that acts as the boot firmware, it is one of the most privileged software components, and it is increasingly targeted by attackers; as explained in Section 1.1.3.

Current platforms focus mostly on boot time integrity of the BIOS and the subsequent software components. The runtime part of the BIOS—that provides services to the OS and the platform in general—and more importantly the privileged services, have received less interest from the community. Unfortunately, while the attacks targeting the BIOS and the SMM were initially just proof of concepts [23, 33, 64, 99, 100, 211, 212, 223, 282–284], real-world attackers [118, 176, 226] have started to exploit vulnerable systems.

In this part, we describe a solution to improve the current situation, specifically on the detection side. The rest of this chapter is structured as follows. In Section 9.1, we recall why the security of the privileged services running in SMM is important, and why current platforms are most likely vulnerable. Then, in Section 9.2, we introduce our contributions. In Section 9.3, we give an overview of our approach and the requirements we have. Finally, in Section 9.4, we provide the threat model and the assumptions that we make for the subsequent chapters.

9.1 MOTIVATION

As explained in Section 2.1.5, the SMM is the last bastion of firmware security. It is the only mode that can write to the flash storage of the BIOS, and its execution is invisible to the OS, thus a perfect place to hide malware [106, 118, 176, 226]. In addition, it allows the attacker to perform actions that cannot be realized with kernel privileges. For example, the attacker could remain persistent on the platform or modify security policies (e.g., disable secure boot).

In practice, vendors typically use third-party code to build their firmware. It makes code review and vulnerability management more difficult, since they do not necessarily follow the same coding style or best practices. Hence, if a vulnerability is present in third-party code, it might be more difficult to find and to fix.

Even if a vulnerability related to the SMM has been found, reported, and patched in the BIOS's source code, all affected platforms needs to update their BIOS. In practice, however, the BIOS is not updated frequently [162]. This may be due to two reasons.

First, there is a fear that updating the BIOS may render the platform unusable[117] if a bug occurs. In comparison to OS updates where if a bug occurs one can simply reinstall the OS, a corrupted BIOS renders the platform unusable. While such a fear may have been legitimate in the past, current platforms can safely update their BIOS, but such a belief still remains.[118] Even in the unlikely case of a corrupted BIOS, platform vendors offer solutions to automatically reflash the BIOS without needing to send back the device.[119]

Second, the adoption of an automatic vendor-agnostic BIOS update mechanism remains slow. Indeed, in the past, updating the firmware of a device either required an inconvenient manual operation and to use vendor-specific software for each platform. Recently, thanks to the UEFI specifications, the update mechanism is more standardized with the UEFI Capsules [272]. In addition, recent efforts from Windows [192] or the LVFS [184] project for Linux provide an automatic update mechanism using the UEFI Capsules. Both the adoption from the vendor side (they have to upload their firmware voluntarily) and from the OS side (they need to enable and use such mechanisms), however, are slow. It results in platforms with outdated and vulnerable BIOSs.

In summary, if a vulnerability is found in a BIOS it can remain unfixed for a long period until an update is applied. Therefore, we can assume that an attacker will find a vulnerability.

In this part of the dissertation, we present an intrusion detection approach that address this issue by detecting intrusions that exploit those vulnerabilities. Such an approach allows platforms to be resilient against attackers despite the vulnerabilities and it reduces the risk of unpatched BIOSs.

9.2 CONTRIBUTIONS

Our work focuses on designing an event-based monitor for detecting intrusions that modify the expected behavior of the SMM code at runtime. While monitoring the behavior of SMM is our primary goal, ensuring the integrity of the monitor itself is critical to prevent an attacker from evading detection. Thus, we isolate the monitor from the monitored component (i.e., the target) by using a co-processor.

[117] Often referred as a "bricked" system or platform.

[118] It has a low probability of bricking the platform, but the impact is high. Hence, even to this day, the fear of updating still persists—for instance one can search "bios update risk" in a search engine and find various people asking in forums about this risk.

[119] For example, HP Sure Start offers this capability [130].

A common issue affecting hardware-based approaches that rely on an isolated monitor is the semantic gap between the monitor and the target [49, 142]. This issue occurs when the monitor only has a partial view of the target state. For example, if the monitor gets a snapshot of the physical memory without knowing virtual to physical mapping (e.g., CR3 register value on x86) it cannot reconstruct accurately the memory layout of the target. Our monitor addresses this issue by leveraging a communication channel that allows the target to send any information required to bridge this semantic gap. We enforce the communication of information relevant to the detection methods via an instrumentation phase. In addition, we ensure that the attacker cannot forge messages without first being detected.

Our detection approach relies on a model of the expected behavior of the monitored component, where any significant deviation from this behavior is flagged as illegal. We chose an anomaly-based approach as we aim not only to detect the exploitation of known vulnerabilities, but also of unknown or unreported (zero-day) vulnerabilities.

This approach is generic since it can rely on different detection methods i.e., different models of the legitimate behavior. It can be applied to monitor different types of low-level software such as SMM or ARM TrustZone secure world [13], which have the following properties: expose primitives called infrequently by upper layers and perform minimal computation per primitive. While generic, our approach introduces multiple challenges (e.g., the overhead involved by the communication, the provenance of the messages, or the integrity of the monitor and the code added by the instrumentation phase). In this work, we focus on the detection of attacks targeting the SMM code as a use case and show how we tackled these challenges. To illustrate the feasibility of our approach, we implement two detection methods: a CFI policy and a policy that enforces the integrity of CPU registers that control the execution context (e.g., CR3 and SMBASE).

Our contributions are the following:

- We propose a new approach using an event-based monitor targeting low-level software (Section 9.3).
- We study the applicability of our approach using two detection methods (CFI and execution context integrity) to detect intrusions and attacks against SMM runtime firmware code (Chapter 10).
- We develop a prototype implementing our approach (Chapter 11).
- We evaluate our approach in terms of detection capability and performance overhead on real-world firmware widely used in the industry (Chapter 12).

9.3 APPROACH OVERVIEW AND REQUIREMENTS

In this section, we describe the generic concepts and requirements of our event-based behavior monitoring approach. As explained previously, such concepts could be used to monitor different targets and could rely on different detection methods. We detail in Chapter 10 one possible implementation of this approach to detect runtime attacks on SMM code using CFI and other invariants.

Our approach, illustrated in Figure 9.1, relies on three key components, which we detail in the following subsections: a co-processor, a communication channel, and an instrumentation step. The dedicated co-processor isolates the monitor from the target. The target uses the communication channel to give more precise information about its behavior to the monitor. The instrumentation step enforces the communication.

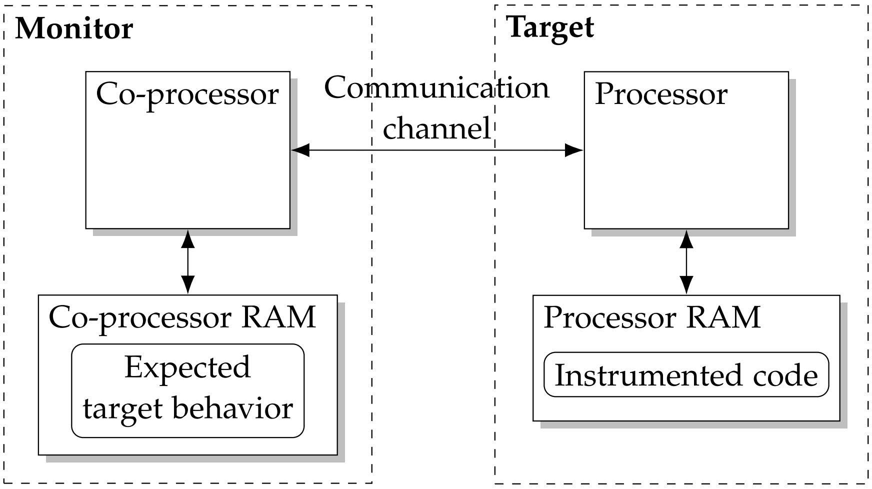


Figure 9.1: High-level overview of our co-processor-based monitoring approach

9.3.1 *Co-Processor*

The integrity of the monitor is crucial, because it is a trusted component that we rely on to detect intrusions in our system. The monitor could also be used to start remediation strategies and restore the system to a safe state. If the attacker compromised our monitor, we could not trust the detection nor the remediation.

When the target and the monitor share the same resources (e.g., CPU or memory), it gives the attacker a wide attack surface. Thus, it is necessary to isolate the monitor from the target. Modern CPUs provide hardware isolation features (e.g., SMM or ARM TrustZone [13]) to help reduce the attack surface of privileged code. However, if one wants to monitor the code executed in these environments, the monitor itself cannot benefit from these isolation features.

In our approach, we use a co-processor to execute the monitor's logic. This co-processor has its own execution environment and memory. Thus, the attacker cannot directly access this dedicated memory even if the target has been compromised. The attacker could only influence the behavior of the monitor via the communication channel, which becomes the only remaining attack surface. The simplicity of such an interface, however, makes it harder to find vulnerabilities and to attack the monitor. Such a design reduces its attack surface. Nevertheless, using a safe programming language for the monitor and a careful code review should be considered.

In the following subsection, we discuss the requirements for our communication channel.

9.3.2 *Communication with the Monitor*

Since we isolate the monitor from the target, the monitor cannot retrieve entirely the execution context of the target. Thus, there is a semantic gap between the current behavior of the target and what the monitor, executed on the co-processor, can infer about this behavior [49, 142].

We introduce a communication channel between the monitor and the target. It allows the target to send messages to the monitor. Different types of information could be sent using this communication channel such as the content of a variable in memory, the content of a register, or the address of a variable. The nature of such information depends on the detection approaches implemented on the monitor, providing flexibility in our approach.

This flexibility is an important aspect, because the class of vulnerabilities exploited over time evolves. If the attackers cannot exploit a class of vulnerability due to preventive security mechanisms or due to the monitoring, they might switch to other vulnerabilities currently not handled by the monitoring or the mitigations.

The communication channel is the only remaining attack vector against the monitor. Thus, how the monitor processes the messages and how the target sends them are an important part of the security of the approach. To this end, we require the following properties:

(CC1) MESSAGE INTEGRITY If a message is sent to the monitor, it cannot be removed or modified by code executed on the CPU. Otherwise, an attacker could compromise the target and then hide the intrusion by modifying or deleting the messages before they are processed by the monitor.

(CC2) CHRONOLOGICAL ORDER Messages are retrieved by the monitor in the order of their emission. Otherwise, an attacker could rearrange the order to evade the detection.

(CC3) **EXCLUSIVE ACCESS** The instrumented code and the monitor have exclusive access to the communication channel. Other software components of the platform cannot access it. Otherwise, an attacker could forge messages faking a legitimate behavior.

(CC4) **LOW LATENCY** Sending a message should be fast (e.g., submicrosecond), because low-level components need to minimize the time spent performing their task to avoid impacting higher-level components and the user experience.

9.3.3 *Instrumentation of the Target*

We enforce the communication from the target to the monitor by adding the communication code during an instrumentation step. We can perform the instrumentation step during the compilation or by rewriting the executable binary code.

If an attacker tampers with the instrumentation, the monitor would get inaccurate context of the behavior of the target, which can be exploited by the attacker to evade the detection. Thus, the integrity of the instrumentation (i.e., the communication code of the target) is crucial. To this end, we require the following properties:

(I1) **BOOT TIME INTEGRITY** The code and data at boot time are genuine and cannot be tampered with by the attacker.

(I2) **RUNTIME CODE INTEGRITY** The code cannot be modified by the attacker at runtime.

9.4 THREAT MODEL AND ASSUMPTIONS

In this section, we describe the threat model that we consider for the subsequent chapters of Part III. We focus our threat model for SMM code executed on x86 platforms. Nevertheless, similar threat models could be defined for other contexts such as code executed on ARM TrustZone.

As mentioned in the introduction of this chapter, our first assumption is that the attacker can find and exploit a vulnerability in SMM, more specifically a memory corruption in an SMI handler. Avoiding such vulnerabilities requires a strong discipline (e.g., following secure coding practices and reviewing the code). Even with the help of various tools (e.g., static or dynamic analysis), however, vulnerabilities can still be introduced. In Section 2.1.5, we describe attacks and vulnerabilities affecting the SMM—summarized in Table 2.1. We explicitly focus on the vulnerabilities that could be present in SMI handlers.

To ensure the integrity of our solution, we also make assumptions regarding the state of the platform at boot time, and the configuration of the platform at runtime. We assume that the code during the boot process is legitimate and that no attack is performed during that phase until the SMRAM is locked. Such an assumption is reasonable with the use of existing security mechanisms for recent firmware such as cryptographically signed BIOS updates [68, 225], measured boot [268], and verified boot with an immutable hardware root of trust [130, 230].

These mechanisms provide us with code and data integrity at boot time (I1, a requirement stated in Section 9.3.3). In addition, since recent firmware use page tables [281] in SMM we can enable write protection [289, 290] of the SMM code. We also expect that the platform has been configured properly in regard to the hardware registers that set up the SMRAM.[120] Therefore, we can assume code integrity at runtime (I2).

Another key assumption is that the attacker cannot send messages in lieu of SMM without being detected. First, by design, messages cannot be sent by other components than the CPU and among the messages sent by the CPU only those sent in SMM are processed by the monitor.[121] Second, we assume that there is no vulnerability in SMM code that can be exploited by an attacker to forge messages without altering the control flow. Since any attempt to alter the control flow results in the emission of a message describing an invalid control flow,[122] the attacker cannot forge messages without first being detected.

Finally, we do not consider an attacker trying to impede the availability of the system (denial of service) by flooding the communication channel. Attackers already have sufficiently high privileges to perform a denial of service, they do not need to target the SMM or our communication to achieve such a goal—they can already shut down or disable most of the components of the system.

We model an attacker with the following capabilities:

- Complete control over the OS or the hypervisor, meaning that the attacker already found vulnerabilities that elevate its privileges to kernel-level or hypervisor-level,
- Complete control over the memory, except the SMRAM, which is hardware-protected,
- Cannot exploit hardware vulnerabilities (e.g., cache poisoning attacks [99, 283] or bypassing SMRAM protection),
- Can trigger as many SMIs as necessary,
- Can exploit a memory corruption issue in an SMI handler.

[120] We can use the platform security assessment framework called chipsec [61] to detect misconfigured platforms that expose the SMRAM for example.

[121] See Section 11.2.2.

[122] See Section 10.1.

10

DETECTION METHODS AND MODELS

In this chapter, we discuss the detection methods and the models that we use to detect an intrusion in SMM. The first method, discussed in Section 10.1, enforces a CFI policy. The second method, described in Section 10.2, ensures the integrity of hardware registers that defines the execution context of the SMM code. These are the two methods that we consider the most relevant based on the state-of-art attacks that we detailed in Section 2.1.5. Finally, in Section 10.3, we describe how we ensure the integrity of the models used to enforce these policies.

10.1 TYPE-BASED CONTROL FLOW INTEGRITY

We enforce a CFI policy, because it is suited to detect attacks on low-level vulnerabilities that often appears in code written in C.[123] Recent platforms enable page table protections to enforce code integrity and non-executable data in SMM. Hence, with a non-writable executable code an attacker is unable to change the target of a *direct* branch encoded in the instruction performing a control-transfer. Likewise, with non-executable data, the attacker is unable to inject code to subsequently execute it. However, attacks that targets *indirect* branches are still possible. The target of an indirect branch is encoded in a register often populated by the address of a function or the return address of a function stored in memory before executing the indirect branch. Since this memory is not read-only—allowing legitimate modifications of the targets—it could be possible for an attacker to exploit a vulnerability that can modify a function pointer or a return address stored in memory and change the target of an indirect branch. To detect such attacks, we enforce a CFI policy on forward edges (indirect calls) and on backward edges (returns) of the CFG.

10.1.1 *Overview and Motivation*

We enforce a type-based CFI [208, 216, 264] policy on forward edges. It ensures that the address used in an indirect call matches the address of a function having an expected type signature known at compile time. For example, the call site `s->func(s, 1, "abc")` is an indirect call where the function pointer `func` has the following type signature: `int (*func)(struct foo*, int, char *)`. Our approach ensures that the address of `func` used at that call site always points to a function having the same signature.

---

[123] To the best of our knowledge, all SMM code is currently written in C (and a small part in assembly), thus we do not take into consideration other languages such as C++.

Our approach over-approximates the set of expected pointers with all functions with the same type signature. In practice, type-based CFI gives small equivalence classes [36] where one equivalence class contains all the possible targets for one call site. Other approaches use points-to analysis that try to determine all the targets that a pointer can point to. This type of analysis can sometimes give precise results (i.e., the complete set of pointers). However, in practice, as shown by Evans et al. [108], such analysis often fails to give the accurate set of pointers. Therefore, to avoid false positives, they over-approximate resulting in large equivalent classes such as all the available functions in the program.

In addition, in comparison to other type-based CFI approaches that were designed for more general-purpose software, our approach is designed for a more constrained environment. For example, we do not have to handle the dynamic loading of libraries (DLLs) in SMM.

We have divided our approach in two phases: compile time and runtime. At compile time, we analyze the SMM code to gather the type signatures, and we provide them to the monitor. We instrument the code to send the target of an indirect call site (known at runtime) to the monitor and to send the return address from the stack at the prologue and epilogue of each function. The analysis and the instrumentation are automatic, and they only require an additional step during the compilation of the BIOS. At runtime, the monitor checks that each indirect call and return from a function is legitimate.

10.1.2 *Illustrating Examples*

We now give two examples to illustrate the exploitation of vulnerabilities where an attacker hijacks the control flow by calling an illegitimate function from an indirect call site. We first give an example voluntarily simplified for an illustrative purpose. Then, we give an example based on a vulnerability found in a real SMI handler.

Listing 10.1 shows an intentional, simulated, but simple to understand, vulnerable (non-SMM) code. The function `vulnerable` (line 13) reads a number (as bytes) from the standard input and use it to access the `functions` array that contains 4 function pointers. Then, it makes an indirect call by using the function pointer retrieved from the array. The code checks that the number is within the bounds of the array, but a vulnerability allows an attacker to overwrite the first function pointer in the array—more precisely the last 6 bytes. The `read` call at line 17 reads 8 bytes while the variable `i` is only 2 bytes long. An attacker can forge an input so that it overwrites the last 6 bytes of the first pointer to hijack the control flow, and e.g., call the function named `secret`. We illustrate the state of the stack before the attacker overwrites the first pointer in Figure 10.1. By overwriting the pointer,

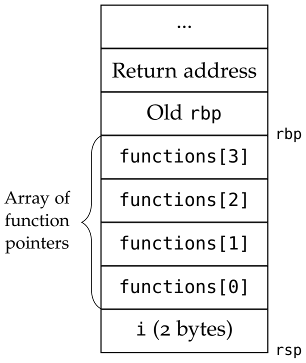


Figure 10.1: Simplified view of the stack frame of the function before the attacker overwrites `functions[0]`

the attacker changed the target of the indirect call to an illegitimate function that did not have the correct type.

```
1  static void secret(void) {
2      system("/bin/sh");
3  }
4
5  static void func1(int a) {
6      puts("func1");
7  }
8
9  static void func2(int a) {
10     puts("func2");
11 }
12
13 static int vulnerable(void) {
14     void (*functions[4])(int) = { func1, func2, func2, func1 };
15     short i;
16
17     read(fileno(stdin), &i, sizeof(long));
18     if (i < 0 || i > 3) {
19         fprintf(stderr, "invalid function number\n");
20         return -1;
21     }
22
23     functions[i](0x1234);
24
25     return 0;
26 }
```

Listing 10.1: Example of a simulated non-SMM vulnerable code

Our type-based CFI policy detects this by ensuring that the function pointer used at an indirect call site always points to a function having the same signature as the indirect call. In this example, our solution performs the check at line 23 of Listing 10.1, and an attacker may only call the functions `func1` and `func2`. Any other function pointer, whether it is a pointer to the first gadget of a ROP-chain or any other function in the code (e.g., `secret` or `vulnerable`), will trigger an alert.

```
EFI_STATUS sub_AD3AFA54(EFI_HANDLE SmmImageHandle, VOID *CommunicationBuffer, UINTN *SourceSize) {
    VOID *v3;
    VOID *v4;

    v3 = *(VOID **)(CommunicationBuffer + 0x20);
    v4 = CommunicationBuffer;
    if (v3) {
        *(v3 + 0x8)(*(VOID **)v3, &dword_AD002290, CommunicationBuffer + 0x18);
        *(VOID **)(v4 + 0x20) = 0;
    }

    return 0;
}
```

Source: Vulnerability discovered by Oleksiuk [212]

Listing 10.2: Example of a vulnerable function from a real SMI handler based on decompiled code

Listing 10.2 illustrates the case of a vulnerability in a real SMI handler [212].[124] While the vulnerability is simple, the decompiled code makes it more difficult to understand. At line 8, we have `*(v3 + 0x8)` that fetches a function pointer from a structure. The structure is part of the communication buffer structure that is controlled by the attacker (`v3`). Consequently, at line 8, the code performs an indirect call using a function pointer controlled by the attacker—giving the attacker the ability to hijack the control flow in SMM. Here the attacker can either call an illegitimate function, or more likely build a ROP chain and call the first gadget from this indirect call site.

10.1.3 *Code Analysis and Instrumentation*

We now explain how we analyze the SMM source code to build the models needed by the monitor to check each indirect call.

For each indirect call site, we assign a unique identifier (CSID), we create a mapping between their CSID and the type signature of the function called, and we add this type into a set of types called indirectly (SIND). We instrument each indirect call site to send the CSID and the branch target address to the monitor before executing the indirect call.

Then, for each function whose type signature is in SIND, we build at compile time a mapping between the function offset in memory and its type. This mapping gives us all the functions that could be called indirectly with their type signature and offset in memory.

Call Site ID	Type
1561	`i8(i32)`
4852	`i32(i8)`
...	...

Function Offset	Type
`0x0000b804`	`i8(i32)`
`0x0000ca04`	`i32()`
...	...

Figure 10.2: Example of the mappings that the source code analysis outputs

At the end of the build process, our analysis outputs two pieces of information: (1) a mapping between a CSID and its expected type; and (2) a mapping between an *offset* and the type of the function at that location. Figure 10.2 illustrates the mappings that are generated by our analysis. The indirect call with the ID 1561 expects a function that takes an integer (`i32`) and outputs a character or byte (`i8`). The function at offset `0x0000b804` is one possible target.

These mappings are then provided to the monitor. Such information, however, is not enough for the monitor to check the indirect calls. With these mappings, it only has the functions offset and not their final address in memory which are determined at runtime. The monitor needs the base address at which they are located, for each of these functions. We provide this information to the monitor by instrumenting

[124] Oleksiuk dumped the SMRAM from a machine and he analyzed the dump to find functions registered as an SMI handler. Then, he decompiled some of them and found this vulnerability.

the firmware code to send the address during the initialization phase (before the SMRAM is locked). This way, at boot time, the monitor computes the final mapping by adding the offset to the corresponding base address resulting in: a mapping between a CSID and its expected type; and a mapping between the address of a function and its type.[125]

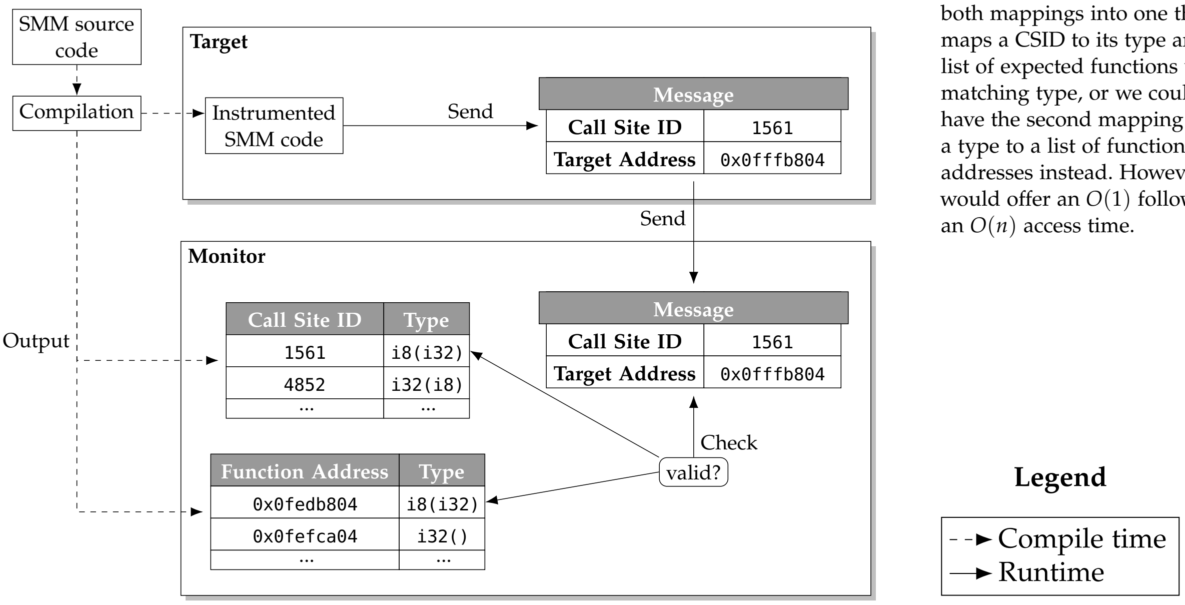

SMM source code
Compilation
Output
Target
Instrumented SMM code
Send
Message
Call Site ID
1561
Target Address
0x0fffb804
Send
Monitor
Call Site ID
Type
1561
i8(i32)
4852
i32(i8)
...
...
Message
Call Site ID
1561
Target Address
0x0fffb804
Check
valid?
Function Address
Type
0x0fedb804
i8(i32)
0x0fefca04
i32()
...
...
Legend
Compile time
Runtime

Figure 10.3: How the monitor detects illegitimate indirect calls

Thanks to the mappings and the instrumentation that sends messages, as illustrated in Figure 10.3, the monitor can verify that the target address (function pointer) received in a message has the expected type according to the call site ID from the same message. The attacker can control the target address, but not the call site ID. Therefore, if the attacker uses an address that points to a ROP chain, the monitor detects it, because this address it not in the second mapping. If the attacker uses a function address pointing to a function with an invalid type, the monitor detects it by checking the mappings.

10.1.4 *Shadow Call Stack*

Since CFI can be bypassed if it only checks forward edges, we implement a shadow call stack to check backward edges. A shadow call stack compares the value of the return address at the prologue and at the epilogue of the function. The CPU stores the return address in the stack at the prologue of a function, and when arriving at the epilogue, there is no legitimate reason for this value to be different. The role of the shadow call stack is to create a copy of the call stack (the succession of return address on the stack) and to verify independently whether the value changed.

---

[125] These mappings offer $O(1)$ access time. We could merge both mappings into one that maps a CSID to its type and its list of expected functions with a matching type, or we could also have the second mapping match a type to a list of function addresses instead. However, both would offer an $O(1)$ followed by an $O(n)$ access time.

We implement our shadow call stack by instrumenting the SMM code so that it sends one message at the prologue and epilogue of each function. The message at the prologue of a function sends a *push* and the one at the epilogue sends a *pop*. Each message contains the current value of the return address on the stack. The monitor uses the messages to create its own copy of the call stack. In addition, after the monitor receives a *pop* message, it checks that the return address received matches the one on top of its copy of the stack—as illustrated in Figure 10.4.

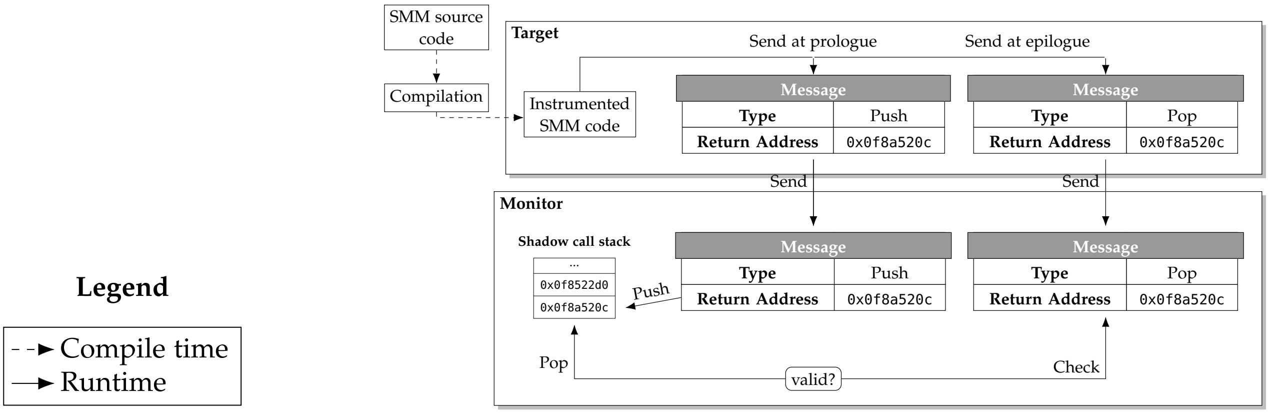

SMM source code
Compilation
Target
Instrumented SMM code
Send at prologue
Send at epilogue
Message
Type
Push
Return Address
0x0f8a520c
Message
Type
Pop
Return Address
0x0f8a520c
Send
Send
Monitor
Shadow call stack
...
0x0f8522d0
0x0f8a520c
Push
Message
Type
Push
Return Address
0x0f8a520c
Message
Type
Pop
Return Address
0x0f8a520c
Pop
valid?
Check
Legend
Compile time
Runtime

Figure 10.4: How the monitor detects illegitimate returns using a shadow call stack

In comparison to other shadow call stack implementations, due to the SMM environment and constraints, we do not have to deal with exceptions, longjmp, or even multi-threading [222].

10.2 EXECUTION CONTEXT INTEGRITY

In addition to a CFI policy, the monitor ensures the integrity of relevant x86 CPU registers in SMM. It stores expected values in its memory at boot time and verifies the values sent by the target at runtime.

When entering SMM, the main CPU stores its context in the save state area and restores it when exiting [139]. For example, the value of the SMBASE register is stored in this save state area. The processor uses the SMBASE register every time an SMI is triggered to jump to the SMM entry point, and when exiting the SMM it restores the value of the SMBASE register from the value stored in the save state area. It is not possible to directly modify the SMBASE register, but it is possible for an SMI handler to modify the SMBASE value stored in the save state area. In that case, the SMBASE register will be restored with this modified value when the processor leaves the SMM. Thus, the next time an SMI is triggered, the processor will use the new value of SMBASE. This behavior is genuine at boot time to relocate the

SMRAM to another location in RAM. At runtime, however, there is no valid reason to do this. If an attacker manages to change the SMBASE, it results in arbitrary code execution when the next SMI is triggered.

To detect this attack, our approach checks that the SMBASE value does not change between SMIs at runtime. At boot time, when the SMM and SMRAM are set up (during the DXE phase), we instrument the code to send the final value (after the relocation) of the SMBASE to the monitor. At runtime, we send the current value of the SMBASE just before the `rsm` instruction that returns from SMM. The monitor can then detect if the SMM has been compromised and if an attacker modified the SMBASE.

In addition, MMU-related registers, like CR3 (i.e., an x86 register holding the physical address of the page directory), are interesting targets for attackers [143]. We need to protect their integrity, since recent firmware use page tables [281, 288, 290]. The CPU resets these registers at the beginning of each SMI with a value stored in memory. Such a value is not supposed to change at runtime. If an attacker succeeds in modifying this value stored in memory, then the corresponding register is under the control of the attacker at the beginning of the next SMI.

Similarly, to detect this attack, we instrument the SMM code to register at boot time the expected value of CR3 and to send at runtime the value of CR3 (stored in memory, not the register) before the `rsm` instruction. The monitor can then detect if an attacker tries to perform an attack similar to the one described by Jang et al. [143].

10.3 ISOLATION OF THE MODELS

Since inlined-based approaches are within the same execution environment than the attacker, they often rely on information hiding to protect the data structures they rely on for their approach. For example, inline-based approaches that implement a shadow call stack need to hide the shadow stack in the same address space as the attacker. They rely on the assumption that attackers will not be able to find the shadow stack [37]. However, since the shadow stack is in the same address space, and since the instrumentation writes to this shadow stack, attackers also have write access to it. Therefore, if attackers can find other vulnerabilities—that differ from the ones allowing to hijack the control flow—they could potentially leak that information or corrupt the data structures without being detected.

Likewise, if attackers can modify the data structures or models that we use to detect an illegitimate behavior, they could bypass the detection. Our approach solves this issue by isolating the detection logic, the models of the behavior, and the data structures (e.g., shadow call stack and indirect call mappings) in the dedicated memory of the monitor. This isolation provides a more robust CFI solution against attackers in comparison to inlined-based CFI approaches.

11

ARCHITECTURE AND IMPLEMENTATION

The design of our solution is illustrated in Figure 11.1. In this figure, straight arrows represent the steps taken during runtime and dashed arrows the steps taken during the instrumentation phase (compilation time). We describe our architecture and our implementation in more details in the following sections.

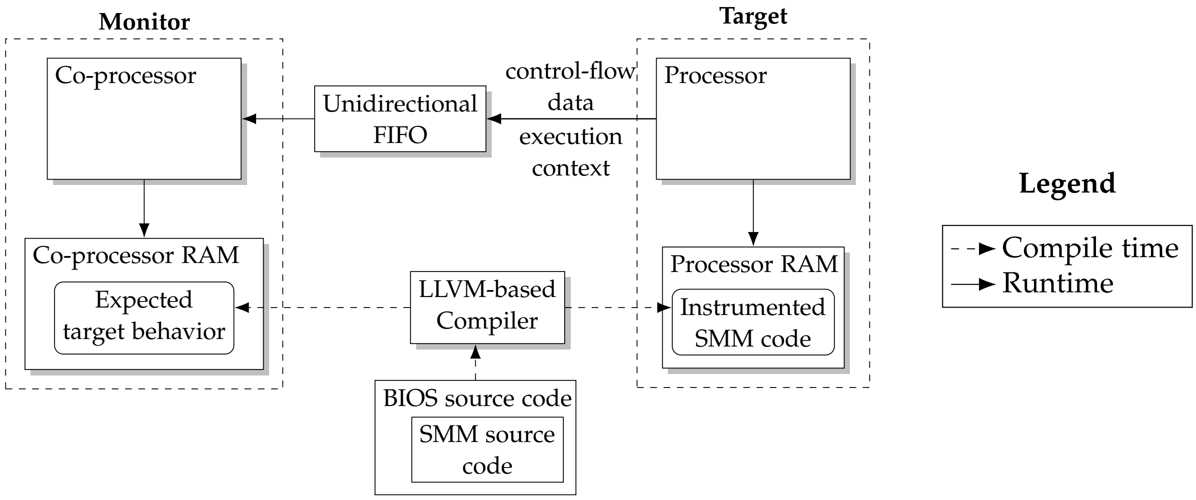

Monitor
Target
Co-processor
Unidirectional FIFO
control-flow data
execution context
Processor
Co-processor RAM
Expected target behavior
LLVM-based Compiler
Processor RAM
Instrumented SMM code
BIOS source code
SMM source code
Legend
Compile time
Runtime

Figure 11.1: High-level overview of our architecture that monitors the SMM

11.1 CO-PROCESSOR AND MONITOR

Our goal is to extend existing platforms without requiring a change of the main processor, and to limit the changes we make in regard to the performance of the platform and its cost. Therefore, the choice of the co-processor for our architecture has several constraints: enough processing power to process the messages, energy efficient, and low-cost.

For example, existing platforms are shipped with co-processors respecting these constraints, such as the AMD Secure Processor—also known as the Platform Security Processor (PSP) [5]—and the Apple Secure Enclave Processor (SEP) [186]. Both are used as a security processor to perform sensitive tasks and handle sensitive data (e.g., cryptographic keys). In those solutions, the main CPU cannot directly access the memory of the co-processor. The CPU can only ask the co-processor to perform security-sensitive tasks via a communication channel. In our case, the main processor does not request any service provided by the co-processor, but it uses the communication channel to send behavior related information.

The PSP is an ARM Cortex A5 and the SEP is an ARM Cortex A7. Such processors are similar, they are both 32-bit ARMv7 with in-order execution and 8-stage pipeline. The main difference is that the A5 is single-issue and the A7 is partial dual-issue.

In our architecture, we chose a similar design, and we propose to use an ARM Cortex A5 [14] co-processor to execute our monitor. It gives us the isolation needed and enough processing power to process the messages for our use case. In practice, we do not necessarily need to add another co-processor to the architecture of the platform if an existing one is present that can be reused—such as the AMD Secure Processor.

For the monitor, as mentioned before, we should reduce the risk of exploitable vulnerabilities that could be used by an attacker to bypass our solution. We focused on memory corruption vulnerabilities by implementing our monitor with approximately 1300 lines of Rust [191], a safe system programming language.

11.2 COMMUNICATION CHANNEL

In this section, we look at how existing co-processors communicate with the main CPU and explain why they do not fit our requirements. Then, we describe how we design our communication mechanism to fulfill the properties we defined in Section 9.3.2.

11.2.1 *Existing Mechanisms*

A major characteristic of the communication channel is its performance, especially its latency, since each message sent by the instrumented code impacts the overall latency of SMI handlers. The Intel BITS defined the acceptable latency of an SMI handler to 150 µs [135]. Delgado and Karavanic [88] showed that, if the latency exceeds this threshold, it causes a degradation of performance (I/O throughput or CPU time) or user experience (e.g., severe drop in frame rates in game engines).

Both the PSP and the SEP use mailbox communication channels to send and receive messages with the main CPU [4, 186]. Mailboxes work as follows. One processor writes to a mailbox register, which triggers an interrupt in a second CPU. Upon receiving the interrupt, the second CPU executes code that fetches the value in the mailbox, processes the message, and then writes a response.

We could use such a mechanism to fulfill our security properties (CC1 and CC2) by making the SMM code wait until the co-processor acknowledged the message—an attacker should then be unable to violate the integrity of the pending messages. Shelton [244] studied the latency of mailboxes on Linux and measured on average a 7500 cycles latency. For example, with a 2 GHz clock this gives 3.75 µs per message, which does not fulfill the low-latency requirement (CC4).

Since the mechanism used by existing co-processors, like the PSP or the SEP, does not allow low latency communication while fulfilling our security requirements, we propose to use a specific hardware component to that end.

11.2.2 *Restricted FIFO*

We propose to add a restricted First In First Out (FIFO) queue between the main CPU and our co-processor. This FIFO is an additional hardware component connected to the main CPU and the co-processor, because we want to re-use existing processors without modifying them.

The goal of the FIFO is to store the messages sent by the target awaiting to be processed by the co-processor. The FIFO only allows the main CPU to push messages and our co-processor to pop them. The FIFO receives messages fragmented in packets. Only our FIFO handles the storage of the messages, if the queue is full it does not wrap over (to avoid overwriting previous messages), and the attackers do not have access to its memory, thus they cannot violate the integrity of the messages after they have been sent.[126] We consider single-threaded access to the FIFO, since only one core handles the SMI, while other cores must wait [139]. [127]

We are using a co-processor with less processing power than the main CPU and the monitor usually processes messages at a lower rate than their production. Thus, the FIFO could overflow or reach its limit. Such a case would happen if the monitored component would be continuously executing, which is not the case with SMM code. Most of the time the main CPU will execute code in kernel mode or user mode, which are not monitored and hence do not send any message. An SMI, on the other hand, will create a burst of messages when triggered. Hence, the only case where the FIFO could overflow is if an attacker deliberately triggered SMIs at very high rate, which would be detected as an attack.[128]

Our design requires a fast interconnect between the main CPU executing the monitored component and the FIFO. The precise interconnect depends on the CPU manufacturer. In the x86 world two major interconnects exist: QuickPath Interconnect (QPI) [134] from Intel and HyperTransport [127] from AMD.[129]

These interconnects are used for inter-core or inter-processor communication and are specifically designed for low latency. For example, CPU manufacturers are using them to maintain cache coherency. Furthermore, they have been leveraged to perform CPU-to-device communication [117, 178, 179]. The co-processor could be connected to the

[126] As mentioned in Section 9.4, we also assume code integrity thanks to various memory protections which ensure that the messages cannot be tampered with before being sent.

[127] At the beginning of each SMI, there is a synchronization code ensuring that only one core executes in SMM. This implies that we do not instrument the code responsible for the synchronization between the cores. This code does not interact with any attacker-controlled data and cannot be influenced by the attacker; hence we trust it.

[128] It would be similar to a DOS attack. Nevertheless, as we mentioned previously in Section 9.4, the attacker already has the privileges to shut down the machine without needing to compromise the SMM.

[129] At the time of this work, these were the interconnects used by Intel and AMD. Recent architectures now use newer versions called Intel Ultra Patch Interconnect [133] and AMD Infinity Fabric [172].

FIFO using these interconnects (using glue logic) or an interconnect with similar performance.[130]

Our monitored component has a mapping between a physical address and the hardware component (i.e., the FIFO) allowing it to send packets via the interconnect. Routing tables are used by interconnects. Such routing tables are configured via a software interface (with kernel privileges) to decide where the packets are sent. Thus, as explained by Song et al. [249], it would be possible for an attacker to modify the routing tables to prevent the delivery of the messages to the FIFO. Such an attack would be the premise of an attack against a vulnerable SMI handler. Therefore, at the beginning of each SMI, we enforce the mapping by overwriting the routing table in the SMM code to prevent such an attack.

In addition, the FIFO filters the messages by checking the SMIACT# signal of the CPU specifying whether the main CPU is in SMM or not [127, 139]. Hence, the monitor only processes messages sent in SMM and prevents an attacker from sending messages when the target is not executing (e.g., an attacker sending messages in kernel mode).

Finally, an important part to consider when designing the communication channel is whether it is synchronous or asynchronous. An asynchronous channel can send a message without waiting its acknowledgment—which reduces the overhead of the communication for the target. However, it increases the delay between an intrusion and its detection. A synchronous channel, on the other hand, allows the monitor to stop the target at the first invalid behavior detected. However, it requires waiting for the acknowledgments and to add more instrumentation code to handle the acknowledgment. This increases the code size, and it increases the overall performance cost. Our approach works with a synchronous or asynchronous channel, but in the context of the SMM, we chose to use an asynchronous channel for performance reasons.

To summarize, this design fulfills the message integrity property (CC1), since the target can only push messages to the restricted FIFO. Moreover, if the queue is full it does not wrap over, and the target enforces the routing table mapping. It fulfills the chronological order property (CC2), because it is a FIFO and there is no concurrent access to it while in SMM. In addition, it fulfills the exclusive access property (CC3), since we filter messages to ensure they only come from the SMM, the integrity of the instrumentation code is ensured with the use of page tables with write-protection enabled, and the attacker cannot forge messages without first being detected. Finally, we fulfill the last property (CC4) by using a low latency interconnect between the main CPU and the FIFO and having an asynchronous communication channel.

[130] For example, ARM platforms can use the AMBA [15] interconnect.

11.3 INSTRUMENTATION

The instrumentation step of our implementation that modifies the SMM code is twofold: (1) an instrumentation to send CFI related information; and (2) an instrumentation to send information regarding x86 specific variables and registers.

We rely on LLVM 3.9 [165], a compilation framework widely used in the industry and the research community, to instrument the SMM code. LLVM is following the architecture of other compilers where a frontend reads the source code and translates it into an Intermediate Representation (IR). The LLVM IR is a strongly-typed low-level programming language similar to assembly, but it is generic in order to abstract architecture details (e.g., it uses an infinite number of registers). After the translation to the IR, multiple passes either gather information from the IR via static analysis, or optimize the IR for size or running time. After the passes are done, the backend translates the IR to target specific machine code (e.g., x86). The backend can also incorporate multiple passes specific to the architecture and its constraints to optimize even more the code by relying on architecture features.

We implement two LLVM passes with approximately 600 lines of C++ code. We execute our passes at link time. The first pass enforces the forward-edge CFI (i.e., indirect calls always branch to valid targets). It is performed on the LLVM IR since it provides us with all the type information that we need. The second pass enforces the backward-edge CFI (i.e., a shadow call stack). It is done in the backend since it is architecture specific, and more importantly we want to ensure that it will not be optimized away or placed outside the prologue or epilogue.

As mentioned previously for the type-based CFI policy, the monitor needs the base address of the functions. We instrument the SMM code manually to send at boot time this information by modifying the functions that load SMM modules in memory during the DXE phase.

We also instrument the SMM code to send some values related to x86 CPU registers. These values, such as SMBASE or the saved value of CR3, could be modified by an attacker to take control of the SMM or evade detection. This part is done manually, since it requires knowledge of the source code to know which variable to send and when.[131] The amount of code required, however, is low and it can be reused for many implementations. Indeed, vendors usually rely on a common framework for their BIOS, such as EDK II [263].

[131] An idea would be to automate part of this work by scanning the source code to find any usage of the relevant registers, such as CR3.

12

EVALUATION AND RESULTS

We evaluated our approach on two real-world implementations of code running in SMM. We first conducted a security evaluation of our approach, as described in Section 12.2, to answer the following questions:

1. Can we detect the exploitation of vulnerabilities in SMI handlers?
2. Do we encounter false positives?

Then, we evaluated the performance of our approach, as detailed in Section 12.3, to answer the following questions:

3. What is the performance impact of the instrumentation and the communication on the SMI handlers?
4. How much time does it take for the co-processor to process the messages sent by the SMI handlers?
5. What is the impact of the instrumentation on the firmware size?

12.1 EXPERIMENTAL SETUP

We used a simulation-based prototype in order to have enough flexibility in exploring the hardware architecture, in a manner that would have been difficult to achieve using real hardware, such as FPGA-based solutions.[132] A simulation allows us to simulate an interconnect and to simulate the delay it takes for the main CPU to send one packet to the restricted FIFO.

We used EDK II [263] and coreboot [259], two real-world implementations of code running in SMM. EDK II is an open source UEFI-compliant firmware used as the foundation for most vendor-based firmware. Coreboot is an open source firmware performing hardware initialization before executing a payload (e.g., legacy BIOS or UEFI-compliant firmware). We built this firmware using our LLVM toolchain, and we only instrumented the SMM related code.

12.1.1 *Simulator and Emulator*

We both used a simulator and an emulator to validate our approach. The main goal of emulators is to be as feature-compatible as possible. However, they are not cycle-accurate and does not try to model accurately the performance of x86 or ARM platforms. Simulators, on the other hand, try to model accurately the performance of the platforms

[132] At the time of the experiments, to the best of our knowledge, there was no off-the-shelf FPGA-based solutions with direct access to HyperTransport or Intel QPI commercially available.

they simulate, but often do not implement all their features (e.g., no possibility to lock the SMRAM). Therefore, we use an emulator to have all the SMM features for the security evaluation, and a simulator to model accurately the performance of our implementation.

For the security evaluation, we used the QEMU 2.5.1 [25] emulator and we modified it to emulate our communication channel.

We used the gem5 [26] cycle-accurate simulator to estimate the performance impact both on the main CPU by modeling an x86 system, and on the co-processor by modeling an ARM Cortex A5. We modified gem5 to simulate our FIFO communication channel. It allowed us to specify the delay (in nanoseconds) it takes to send or receive information from it.[133]

12.1.2 Simulated Communication Channel Delay

We relied on previous studies on interconnects [62, 178] to estimate the delay of the communication channel. Litz et al. [178] encountered a latency between 36 to 64 cycles to send one packet with HyperTransport on a CPU-FPGA platform.[134] Even with a small clock rate, for example 500 MHz, we can expect a latency of around 72 to 128 ns, close to an uncached memory access. Choi et al. [62] have similar results with QPI-based platforms.[135]

Hence, we simulated a delay of 128 ns to send one packet. This corresponds to the worst-case scenario to send one packet. Since the reference latency we have for AMD HyperTransport and Intel QPI are for FPGA prototypes, lower latencies are expected with an ASIC implementation. Furthermore, since we use a point-to-point connection, and since only one core of the main CPU is running while in SMM and sending packets, we did not consider a fluctuation of the latency or any additional delays.

Finally, we simulate the same interconnect and delay between the main CPU and the FIFO, and between the co-processor and the FIFO.[136]

12.1.3 SMI Handlers

To evaluate the performance of our prototype, we need to test its impact on the various types of SMI handlers. Some are platform-specific and interact with specific hardware interfaces with platform-specific, but they are often proprietary. While others are more generic, are reused by multiple implemantations, and they are part of open-source implementations.

For our performance evaluation, we used SMI handlers from EDK II and coreboot. EDK II does not implement any hardware initialization nor vendor-related SMI handlers. At the time of writing, most of the SMI handlers available in EDK II at runtime are dependent on

[133] We give the parameters used for gem5 in Appendix B.

[134] Litz et al. [178] designed an FPGA card with the HTX3 interface, which is needed for point-to-point communication with HyperTransport. Xilinx used to sell such products, but they are now discontinued.

[135] Choi et al. [62] had access to a QPI-based CPU-FPGA platform thanks to a collaboration between Intel and academics at that time.

[136] In a real prototype, one may need to use a glue logic for the ARM architecture.

standard hardware components that cannot be easily simulated (e.g., a storage device that follows the Opal specifications [269] for encryption or a TPM chip).

In our evaluation, we used the VariableSmm SMI handlers from EDK II. They manage variables within the SMM [291] thanks to four different handlers: GetVariable, SetVariable, QueryVariableInfo and GetNextVariableName (GNVN).

Since coreboot provides hardware initialization and vendor-related SMI handlers, we use them for our evaluation. In addition, these handlers communicate with devices that can be simulated with gem5. A majority of these handlers, however, are simpler compared to the VariableSmm SMI handlers. We used three SMI handlers for the Intel ICH4 i82801gx[137] and two for the AMD Agesa Hudson southbridge.[138] These SMI handlers process hardware events such as pressing the power button (PM1), General Purpose Events (GPE), Advanced Power Management Control (APMC) events, or Total Cost of Ownership (TCO) events.

For the performance evaluation with gem5, in the case of EDK II, we modified EmulatorPkg (usually used to simulate an UEFI environment) where we added support for the SMM. It gave us an environment similar to the execution in SMM to execute the SMI handlers. It does not fully model the SMM, but this part of the evaluation does not focus on the functionality but on the performance. Moreover, the missing features (e.g., no possibility to lock the SMRAM) do not impact the performance results in any way. In the case of coreboot, we extracted the 5 SMI handlers and compiled the source code of the SMI handlers and not the whole firmware, since coreboot did not support the use of clang (LLVM) as a compiler.

12.2 SECURITY EVALUATION

There is no public data set of vulnerable SMM code, in contrast to userland applications. Attacks targeting the SMM are highly specific to the architecture and to the proprietary code of the platform. This code is therefore not publicly available and would not execute on our experimental setup, thus it cannot be used to test our solution.

Consequently, we have implemented SMI handlers with vulnerabilities similar to previously disclosed ones[139] affecting real-world firmware. We reproduced attacks exploiting the following vulnerabilities giving arbitrary execution:

- A buffer overflow in a SMI handler allowing an attacker to modify the return address stored on the stack [145];
- An arbitrary write allowing an attacker to modify a function pointer used in an indirect call [211];

[137] For example, present on motherboards from Apple, ASUS, GIGABYTE, Intel or Lenovo.

[138] For example, present on motherboards from AMD, ASUS, HP, Lenovo or MSI.

[139] See Section 2.1.5 for more details.

- An arbitrary write allowing an attacker to modify the SMBASE [223];
- An insecure indirect call where the function pointer is retrieved from a data structure controlled by the attacker [212].

Vulnerability	Attack Target	Security Advisories	Detected
Buffer Overflow	Return address	CVE-2013-3582 [78]	Yes
Arbitrary write	Function pointer	CVE-2016-8103 [81]	Yes
Arbitrary write	SMBASE	LEN-4710 [170]	Yes
Insecure call	Function pointer	LEN-8324 [171]	Yes

Table 12.1: Effectiveness of our approach against state-of-the-art attacks

As shown in Table 12.1, the monitor detected all these attacks as soon as it received and processed the messages, since these attacks modify the control-flow of the SMM code (i.e., its behavior). We did not encounter false positives, which is expected since we use a conservative strategy regarding indirect calls. Also, while bad software engineering practices using function type cast could introduce false positives, we did not encounter such cases in the code we evaluated, as no function cast was present.

Finally, our CFI implementation performs a sound analysis to recover the potential targets of an indirect call. Therefore, the analysis is not complete, and it would be possible for an attacker to redirect the control flow to a function that should have never been called, but that has the expected type signature (a type collision). Nonetheless, we argue that a type-based CFI increases the difficulty for the attacker, since the only available targets for an indirect call are a subset of the existing functions within the SMRAM with the right type signature. Our analysis with EDK II gave 158 equivalence classes of size 1, 24 of size 2, 42 of size 3, 2 of size 5, 1 of size 9, and 1 of size 13. As mentioned by Burow et al. [36], a high number of small equivalence classes provides a precise CFG. A way to improve the precision of the CFG would be to combine our static analysis (providing some context-sensitivity with the type information), with a points-to analysis, such as the work from Lattner et al. [166]. Points-to analyses can sometimes give the complete set of the functions being called at an indirect call site. An idea would be that if we know that the points-to analysis gave a complete set, the monitor uses this information to validate an indirect call, otherwise it uses the over-approximation of the type signature.

12.3 PERFORMANCE EVALUATION

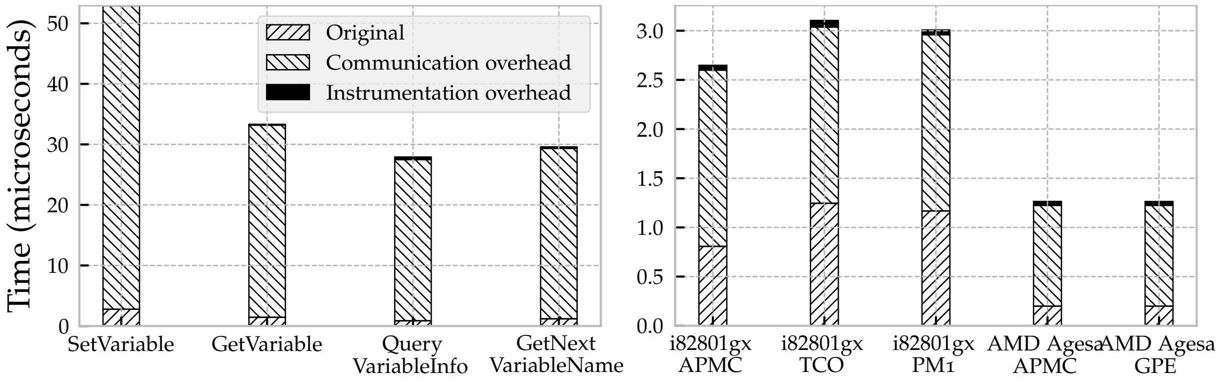
As explained in Section 11.2.1, the time spent in SMM has to be limited (threshold of 150 µs) [88, 135]. On that account, we evaluated the running time overhead of our solution on SMI handlers for the main CPU. We also evaluated the time it takes for the co-processor to process the messages sent by different SMI handlers. Thus, we can estimate the time between an intrusion, its detection, and its remediation.

Finally, the size of firmware code is limited by the amount of flash (e.g., 8MB or 16MB). Thus, we evaluated the size of the firmware before and after our instrumentation.

12.3.1 *Runtime Overhead*

The additional SMM code added with our instrumentation introduces two costs: the raw communication delay between the main CPU and the hardware FIFO; and the instrumentation overhead. The former is related to the time it takes the main CPU to push the packets to the FIFO. The latter is due to multiple factors, such as fetching and executing new instructions or storing intermediate values resulting in register spilling (e.g., the return address of a function fetched from the stack).

We performed 100 executions of each SMI handler we selected for our evaluation.[140] For each SMI handler, we measured the time it takes for the original handler to execute, the cost of the communication, and the additional instrumentation overhead. The results we obtained are illustrated by Figure 12.1.



(a) EDK II VariableSmm (b) coreboot

Figure 12.1: Time (in microseconds) to execute each SMI handler (averaged over 100 executions) with the original time, and the overhead divided between the communication overhead due to pushing packets to the FIFO and the instrumentation overhead

We see that even with a low latency of 128 ns for the communication channel, there is a high overhead due to the communication. The

[140] See Section 12.1.3.

instrumentation overhead, on the other hand, is negligible in comparison. For the SMI handlers from EDK II, the high overhead is also due to the number of messages related to the shadow stack (see Table 12.2), while the number of messages for indirect calls or the integrity of the relevant CPU registers (SMBASE and CR3) are negligible. For the SMI handlers from coreboot, we see a lower overhead since the code is less complex, calls fewer functions, and perform less indirect calls. Nevertheless, for each SMI handler, even with the overhead of our solution, we observe that the time spent in SMM is below the 150 µs threshold [135]. It ensures that the impact on the performance of the system is low and not noticeable for the user.

	Number of packets sent			
SMI Handler	Shadow stack (SS)	Indirect call (IC)	SMBASE & CR3 (SC)	Total number of packets
EDK II				
VariableSmm				
SetVariable	384	4	4	392
GetVariable	240	4	4	248
QueryVariableInfo	299	4	4	208
GetNextVariableName	212	4	4	220
coreboot				
Intel i82801gx				
APMC/TCO/PM1	8	2	4	14
AMD Agesa Hudson				
APMC/GPE	4	0	4	8

Table 12.2: Number of packets sent during the execution of one SMI handler (Number of packets per message type: SS=2, IC=2, SC=4)

12.3.2 *Co-Processor Performance*

We measured the time it takes for the monitor to process all the messages generated by one execution of each SMI handler. We made an average of 1000 executions. Results are illustrated in Figure 12.2.

For each SMI handler there is at least a factor of 4 between the time it takes for the target to execute the instrumented SMI handler and the time it takes for the co-processor to process all the messages that have been sent by the instrumented SMI handler. For example, we see in Figure 12.1 that it takes around 52 µs to execute the SetVariable SMI handler, and in Figure 12.2 that it takes around 230 µs to process all the messages. This means that there is a delay between an intrusion

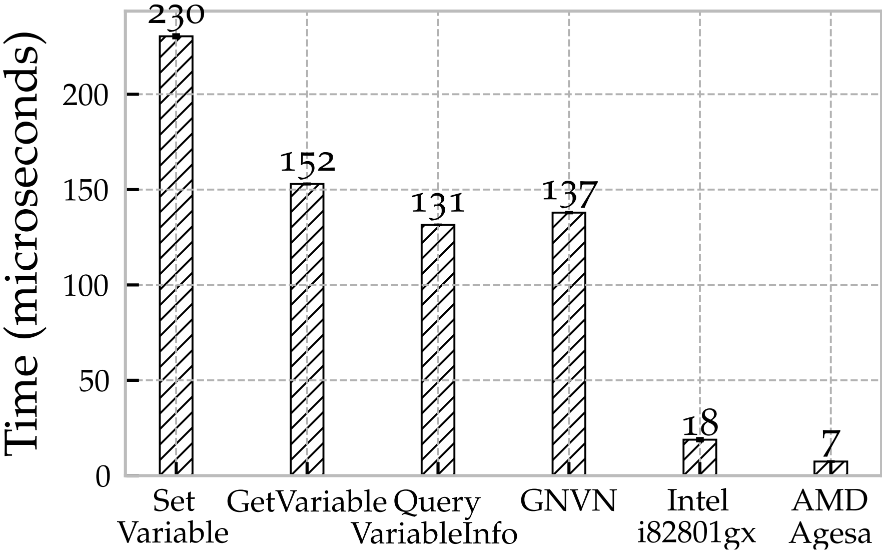


Figure 12.2: Time (in microseconds) to process all the messages sent by one execution of each SMI handler for the co-processor

and its detection, but such a delay will be less than a millisecond. Hence, the co-processor could start a remediation action within one millisecond after an intrusion occurred.

In our threat model, the attacker already has kernel privileges before attacking the SMM code. Our objective is not to detect intrusions that could have been done solely with kernel privileges, such as leaking confidential data. We consider that the final objective of the attacker is to remain persistent in the system even in the case of a reboot. In this case, a remediation action would not prevent the intrusion and does not have to be taken immediately.

12.3.3 *Firmware Size*

For EDK II, our instrumentation added 17408 bytes to the firmware code. However, firmware is compressed before being stored in the flash and only a subset of the firmware is related to the SMM. We measured a 0.6 % increase in size of the compressed firmware. Thus, our instrumentation incurs an acceptable overhead in terms of size for the firmware.

For coreboot, our instrumentation added 568 bytes for the AMD Agesa Hudson SMI handlers and 3448 bytes for the Intel i82801gx SMI handlers. However, at the time of the experiments, we were not able to measure the whole firmware size when building coreboot with our LLVM toolchain, since coreboot did not support clang as a compiler.[141] We built separately the SMI handlers from coreboot toolchain for our evaluation, but compiling the whole firmware (not just the SMM related code) was not possible.

[141] We could measure the size of coreboot compiled with gcc, but the size varies when using clang or gcc.

12.4 SUMMARY

In this chapter, we described our experimental setup that relies on a simulator and an emulator. We evaluated the ability of our prototype to detect attacks against the SMM. All attacks were detected, and no false positives were encountered. The analysis gave a precise CFG with a high number of small equivalence classes. We also evaluated the runtime overhead for the SMI handlers (due to the instrumentation and communication), and none of them crossed the 150 µs threshold (despite a high overhead). We then evaluated the time the co-processor takes to process messages. It gave us a maximum of 230 µs—allowing us to expect a time to detection less than 1 ms in general. Finally, we evaluated the firmware size overhead when compiled with our solution. Since our solution only instruments the SMM code, we observed a low overhead (0.6 %). These current results are encouraging, since they show that our solution can detect intrusions at the SMM level, while not impacting the availability of the system.

13

CONCLUDING REMARKS

In this chapter, we discuss the limitations of our detection approach, we give a quick summary of the comparison of our approach against the related work, and we conclude Part III while providing some potential areas to investigate in the future.

13.1 DISCUSSION AND LIMITATIONS

Having demonstrated the effectiveness of the detection against state-of-the-art attacks and showed that the performance impact of our solution still kept the SMI handlers below the 150 µs threshold, we now discuss some limitations of the approach or the evaluation, and areas that would need further work.

TYPE COLLISIONS A year after the publication of this work, Farkhani et al. [109] published a study on the effectiveness of type-based CFI in userland applications (e.g., nginx and Exim). They showed that the main threat for such an approach is type collision—illegitimate functions having the same type as legitimate functions for an indirect call. In practice, the target functions—that an attacker wants to call when exploiting the vulnerability—rarely have the same type as the indirect call site. They showed, however, that it is possible to find type collisions with functions that call the intended target function of an attacker. For example, the attacker wants to call the function `foo`, but it does not have the same type as the indirect call site. The function `bar` does have the same type and also calls `foo`. Using this technique, they found multiple chains allowing arbitrary code executions in userland applications with multiple nested functions calls eventually leading to the `execve` function. This may make us question the effectiveness of type-based CFI. In the case of the SMM, however, we do not have functions similar to `execve`. Nevertheless, it would be interesting to perform a similar study on various SMM code implementations to understand whether type collisions are also prevalent[142] and if this technique could be used to achieve arbitrary code execution in SMM (e.g., by calling functions that first disable code integrity or non-executable data protections).

SHADOW CALL STACK As mentioned previously, a recent document from Intel [136] suggests that their future processors will be shipped with a CFI technology in hardware and available for

[142] A preliminary analysis using the size of the equivalence classes (see Section 12.2) tells us that type collisions are rare in the code that we used in the evaluation.

the SMM. If this technology does appear, it would be possible to reduce our overhead by using it, since our current main performance overhead is related to the handling of the shadow stack. It is important to note that we would still instrument the indirect calls, because the tracking of indirect branches with the Intel solution is too coarse-grained Indeed, Intel solution allows an indirect call to target any function, while our solution reduces the set of targets to any function with a legitimate type. Finally, we would send messages to ensure that the hardware CFI is enabled and a message when an exception occurs due to a violation of the shadow call stack. Such an implementation would allow us to implement other detection methods since our overhead would be lower. Another idea to reduce the current overhead would be to evaluate the feasibility of reducing the number of packets (e.g., if we use an unused bit in the target address to distinguish whether this is a pop or a push; we save one packet). Finally, we could also investigate approaches that do not instrument functions if we can prove that they cannot corrupt the stack [216].

SIMULATION-BASED PROTOTYPE In the evaluation of our approach, we used a simulation-based prototype since we were not able to find suitable boards to implement an FPGA prototype for the communication channel (no commercially available FPGA boards with direct access to the low latency interconnect with Intel or AMD x86 processors). In addition, it gave us more flexibility to test and explore the hardware architecture. Simulations, however, do not offer the same characteristics as real systems. The performance results could thus be considered unsound. To mitigate this threat, we used the state-of-the-art simulator gem5[143] to model as accurately as possible the systems we were using, and we based our timing model for the communication channel on various studies.

[143] Butko et al. [38] evaluated that gem5 gave a performance prediction with a 20 % error on average.

13.2 COMPARISON WITH RELATED WORK

Our approach addresses various issues that we identified from the literature.[144] We summarize in Table 13.1 a comparison of our approach against previous work that we consider the most relevant.

[144] See Section 3.2 for more details about the state of the art related to our work.

To the best of our knowledge, our approach is the first that provides a flexible (i.e., that allows us to implement various policies or detection methods) anomaly-based SMM monitoring while addressing issues such as the semantic gap or transient attacks. For example, if we look at the three approaches showed in Table 13.1 that can monitor the SMM, they are either knowledge-based [34], have a semantic gap issue [34, 180], cannot detect transient attacks [34], or lack flexibility since they can only enforce one specific policy [136].

	Approach						
Characteristics	Our approach	Copilot [220]	DeepWatch [34]	HyperSentry [18]	KI-Mon [167]	MGuard [180]	Future Hardware-based CFI [136]
Monitor the SMM	●	❍	●	❍	❍	●	●
Code integrity	■	●	●	●	●	●	■
Control Flow Integrity	●	❍	❍	❍	❍	❍	●
SMBASE and CR3 integrity	●	❍	❍	❍	❍	❍	❍
Flexible	●	●	●	●	●	●	❍
No semantic gap issue	●	❍	❍	●	❍	❍	●
Detect transient attacks	●	❍	❍	❍	●	●	●
Detect unknown attacks	●	●	❍	●	●	●	●
No new or modified hardware	❍	❍	●	●	❍	❍	❍

Table 13.1: Summary of the comparison between our SMM behavior monitoring approach and the related work

Legend

● Has the property
❍ Does not have the property
■ Assumed

In our approach, we add a new hardware component—the restricted FIFO—to allow the main processor to send information. This new component, however, does not need to change in the future if the threat or the types of vulnerabilities evolve over time. For example, in comparison to other approaches that enforce CFI [136, 169] by modifying the processor or relying on debugging features, our approach is more flexible since we only need to update the instrumentation and the monitor's logic to implement another detection method.

Finally, in comparison to other work, we assumed code integrity by relying on memory protection from the MMU. An idea to investigate would be to combine our approach with others that detects if the code has been compromised (e.g., if the page table has been tampered with)—strengthening the robustness of the detection method.

13.3 CONCLUSION AND FUTURE WORK

In this part of the dissertation, we proposed a new event-based approach for low-level software using three key components: a co-processor to isolate the monitor, a communication channel to reduce the semantic gap, and an instrumentation of the software to enforce the communication of behavior related information. We show that we can use this approach to detect intrusions targeting SMM services by maintaining a CFI policy and by ensuring the integrity of the execution context (CR3 and SMBASE). Nevertheless, it is flexible, and it can implement other detection methods. Unlike other approaches, we solve the challenges of the semantic gap and the transient attacks while remaining flexible.

We implemented our approach by instrumenting and monitoring real-world firmware, and by simulating the co-processor executing our monitor. The results show that we detect state-of-the-art attacks against the SMM, while remaining below the 150 µs threshold, thus avoiding any noticeable impact on the user.

This work has been published and presented at ACSAC in December 2017 [60]. For future work, we would like to investigate the following areas:

RECOVERING FROM THE INTRUSION Current state-of-the-art solutions help to restore (at boot time) the BIOS—stored in the flash—in a safe state in case it has been compromised or corrupted unintentionally. However, to the best of our knowledge, no solution exists to recover the state of the SMM at runtime without impacting the user experience, in case it gets compromised.

MONITORING OTHER TARGETS Our approach could be used to monitor other low-level targets such as the ARM TrustZone secure world [13]—since it offers a similar environment than SMM (e.g., a non-secure bit to know whether the CPU is in the secure world or not, like the SMIACT# signal). We would like to investigate whether this is feasible and to know if there are potential barriers or new challenges to the generalization of the approach.

IMPLEMENTING OTHER DETECTION METHODS Previous work focused on ensuring code integrity and data integrity for runtime firmware. In our work, we focused on ensuring the integrity of the control-flow and the integrity of registers that affect the execution context. Chen et al. [50] demonstrated that non-control data attacks are realistic threats against real-world programs. Therefore, we would like to investigate whether approaches such as data flow integrity [44] could be implemented in our context.

Part IV

EPILOGUE

14

CONCLUSION

> *Let's go exploring!*
>
> — **Calvin**
> Calvin and Hobbes, Bill Watterson

We began this dissertation by describing three problems that today's platforms face:

1. preventive security is not sufficient,
2. commodity OSs can detect but cannot survive intrusions,
3. and low-level components are increasingly targeted by attackers.

The first problem is well-known, and decades of research try to address this issue. In our work, however, we noticed that despite such a well-known fact, the second and the third problem—affecting today's platforms such as servers, laptops, or smartphones—are either not addressed in the literature or the state-of-the-art solutions are limited (e.g., loss of availability or coarse-grained responses). These gaps motivated our research and eventually led to this dissertation.

In the rest of this chapter, we first provide a summary of the contributions that support the two claims that we exposed in Section 1.2. Then, we list some potential areas to investigate for future work.

14.1 SUMMARY OF THE CONTRIBUTIONS SUPPORTING OUR CLAIMS

In Part II, we demonstrated that intrusion survivability is a viable approach for commodity OSs. The goal of the approach is to recover the compromised service to a previous state and then to put it in a degraded mode to withstand future reinfection. It minimizes the availability cost and allows the system to survive the intrusions while waiting for a patch to be applied fixing the vulnerabilities used by the attackers. We introduced a cost-sensitive response selection process to help select optimal responses when an intrusion is detected. We designed an architecture to orchestrate the selection, recovery, and the application of the responses that degrade the state of the compromised service. We implemented a Linux-based prototype and used it to evaluate our approach. The results support our claim that commodity OSs can survive intrusions by showing that:

- responses allow the service to withstand a reinfection or stop it,

- suitable responses are selected based on the available information that we have on the intrusion,
- the availability cost is small,
- the monitoring cost is small (except in the rare case of a service that performs many small write accesses to the file system in burst),
- and that the approach is applicable to various types of services.

In Part III, we demonstrated that we can detect intrusions at the firmware level with the highly privileged SMM as a use case. Low-level software components—such as the SMM—are increasingly targeted and our approach helps to detect intrusions that compromise them. We introduced an event-based and co-processor-based behavior monitoring approach specifically targeted at low-level software components. Our approach addressed limitations from the literature (e.g., the semantic gap or transient attacks) while remaining flexible. We detailed two detection methods and the architecture of our solution. We implemented a simulation-based prototype and used it to evaluate our approach. The results support our claim that a hardware-based approach is suitable to detect intrusions at the SMM level by showing that:

- state-of-the-art attacks against the SMM are detected,
- no false positives were encountered,
- the instrumentation of the SMI handlers does not make them exceed the 150 µs threshold,
- the co-processor can detect an intrusion in less than a millisecond,
- and the firmware size did not increase significantly.

These contributions and results, however, are only the starting point to build platforms that can withstand, survive, or be resilient against intrusions. There is still a long way to go to apply these ideas to real world systems at scale. We now discuss the next steps.

14.2 PERSPECTIVES

We focused our work on the aspects that we considered the most relevant in consideration of the problems we listed. Moreover, we only evaluated certain aspects of our approaches (e.g., security, effectiveness, or performance). However, during our work we identified areas that require further work, or other aspects that we should evaluate. We already listed some directions for future work—that are specific to each contribution—in the conclusion of each part (Section 8.3 and Section 13.3). We first summarize them here, then we discuss more long-term perspectives that are related to the general idea behind this thesis.

14.2.1 *Extend the Approaches and their Evaluation*

For the work described in Part II related to the intrusion survivability at the OS level, we would like to investigate the following areas for future work:

- Evaluating the feasibility of deactivating responses.
- Maintaining a history of the success and failures of the responses against a specific malicious behavior to mitigate the bias and errors that experts can introduce when assessing the performance of a response.
- Using hardware-based isolation mechanisms to improve the security of our architecture and increase our trust in the enforcement of the responses.

For the work described in Part III related to the intrusion detection at the firmware level, we would like to investigate the following areas for future work:

- Performing runtime SMM recovery after an intrusion has been detected—without impacting the availability of the platform—to evaluate whether our approach can successfully recover from an intrusion in SMM.
- Monitoring other low-level targets (e.g., ARM TrustZone secure world) to evaluate the generalization of the approach.
- Implementing other detection methods—such as data flow integrity [44]—to evaluate the flexibility of the approach.

Here, we only listed non-exhaustively directions that could extend our current approaches. We now take a look at less immediate steps related to our work.

14.2.2 *Surviving and Adapting Intrusions*

Let us now consider more long-term and less immediate directions. They do not necessarily require that the previously listed directions are addressed. We consider the following perspectives less immediate since they would require more work, and we do not have a clear view of how to tackle the challenges involved.

The first open question is how—based on the ability to recover at the SMM-level—can we make the SMM able to survive intrusions?

At the beginning of this dissertation, we explained that low-level components—such as the SMM—did not have sufficient intrusion detection capabilities, while OSs did not have intrusion survivability capabilities. We introduced approaches to fill these gaps, but we are now at a point where the SMM is only able to detect without surviving. While being able to do recovery at the SMM-level could be possible technically, we wonder whether it would be possible to put SMI handlers in a degraded mode. First, we do not know how feasible it would be to reduce the privileges of code executing in SMM at the granularity of one SMI handler.[145] Second, even if we had such an ability, an SMI handler performs specific tasks with most of the time only one specific function. It is not sure whether it would be possible to restrict an SMI handler without rendering its only function unavailable.

The second open question is how to *automatically* adapt the system so that it is not vulnerable anymore to the vulnerabilities used by the initial intrusion—allowing us to deactivate the responses that degrade the services. Intrusions may be due to a software flaw (e.g., a memory corruption or a race condition), thus we should patch the software. Or it may be due to a misconfiguration, thus we should reconfigure the service properly. The difficulty lies in both the ability to *automatically* determine what and where the vulnerability is (e.g., a software flaw or a misconfiguration), but also in determining how to patch it.[146] Finally, an important aspect is how can we guarantee that the automatic fix indeed makes the system not vulnerable anymore and not just one particular exploitation of the vulnerability.

[145] An area to investigate would be the use of virtualization in SMM by looking at SMI Transfer Monitors (STMs) [288].

[146] A fix could also be changing the implementation of a protocol.

APPENDICES

A

MALWARE SAMPLES

Malware	SHA-256
Linux.BitCoinMiner	`690aea53dae908c9afa933d60f467a17ec5f72463988eb5af5956c6cb301455b`
Linux.Rex.1	`762a4f2bf5ea4ff72fce674da1adf29f0b9357be18de4cd992d79198c56bb514`
Linux.Encoder.1	`18884936d002839833a537921eb7ebdb073fa8a153bfeba587457b07b74fb3b2`
Hakai	`58a5197e1c438ca43ffc3739160fd147c445012ba14b3358caac1dc8ffff8c9f`

Table A.1: Malware used in our experiments with the SHA-256 hash of the samples

In Table A.1, we list the malware samples used in our experiments alongside their respective SHA-256 hash.

B

GEM5 PARAMETERS

Parameter		x86	ARM Cortex A5
CPU Type		DerivO3Cpu	timing[i]
Clock		2 GHz	500 MHz
Restricted FIFO latency		128 ns	128 ns
Cache line size		32 B	32 B
L1 I	Size	32 kB	16 kB[ii]
	Associativity	2	2
L1 D	Size	64 kB	16 kB [ii]
	Associativity	2	4
L2	Size	2 MB	512 kB
	Associativity	8	8
DRAM	Type	DDR3_1600	LPDDR3_1600_x32[iii]
	Size	1024 MB	10 MB

[i] We use the timing model since the A5 is a single-issue in-order CPU and our evaluation mainly depends on load/store operations. [ii] The cache size has a range of options: 4 kB, 8 kB, 16 kB, 32 kB or 64 kB. [iii] Educated guess, based on the fact that this is a standard for low power consumption memory.

Table B.1: Parameters used with gem5 for the x86 and the ARM simulation

In Table B.1, we show the different parameters we used to configure gem5 for the simulation of the main CPU and the co-processor. We used the default parameters of the out-of-order x86 simulation, except the CPU clock, which we set to a higher frequency. For the ARM Cortex simulation, we derived the parameters from the ARM technical reference manual [14].

RÉSUMÉ SUBSTANTIEL EN FRANÇAIS

Le crime organisé, l'espionnage industriel, les attaquants opportunistes, ou encore les acteurs étatiques, font tous partie des menaces potentielles[147] pour la sécurité du système d'information d'une organisation ou d'un individu. Des décennies de recherche et de progrès en sécurité informatique nous ont permis de mettre en place de nombreux mécanismes de sécurité préventifs. Parmi ces mécanismes, nous pouvons citer comme exemples la cryptographie, le contrôle d'accès, ou les pare-feux.

Pourquoi, malgré ces mécanismes, des intrusions surviennent-elles ? Pourquoi ne nous est-il pas possible de construire des systèmes parfaitement sécurisés ? Plusieurs facteurs entrent en jeu et doivent être pris en compte afin de répondre à ces questions. Avant d'exposer la thèse soutenue par ce manuscrit, nous expliquons ces facteurs et les différents problèmes auxquels nos systèmes d'information font face de nos jours.

INTRODUCTION

Les mécanismes de sécurité préventifs visent à empêcher un attaquant de violer des propriétés de sécurité. Les trois propriétés principales sont la confidentialité (seules les entités autorisées peuvent accéder à l'information), l'intégrité (seules les entités autorisées peuvent modifier l'information), et la disponibilité (les entités autorisées doivent avoir accès à l'information quand elles le souhaitent).[148]

Insuffisance des mécanismes de sécurité préventifs

Bien que nos systèmes soient conçus avec des mécanismes cherchant à garantir les propriétés susnommées, les personnes en charge de la conception de systèmes peuvent faire des erreurs, sont contraints par des budgets (en temps et en argent), sont biaisés, ou limités par des politiques internes au sein de leur organisation. Ainsi, certaines protections ne seront pas mises en place ou seront vulnérables. Cela signifie qu'en pratique, nos systèmes d'information sont conçus pour prévenir la violation des propriétés de sécurité jusqu'à un certain point.[149] De plus, nos systèmes sont exposés que ce soit via Ethernet, Wi-Fi, Bluetooth, ou le réseau de téléphonie mobile. Les attaquants peuvent donc attaquer nos systèmes en continu jusqu'à ce qu'ils soient compromis, tout en prenant peu de risque, car l'attaque est réalisée à distance et est difficile à attribuer.

[147] Pour plus d'informations, voir le rapport annuel de l'agence européenne chargée de la sécurité des réseaux et de l'information (abrégée ENISA en anglais) [188] étudiant l'évolution des menaces ciblant les systèmes d'information.

[148] Le terme entité est à prendre au sens large et fait référence à des utilisateurs, des processus, ou des systèmes par exemple.

[149] C'est pourquoi il est important d'élaborer des modèles d'attaquant (*threat model* en anglais) afin de savoir précisément contre quoi on souhaite se protéger.

Plus précisément, des intrusions peuvent survenir par la combinaison de plusieurs facteurs techniques et économiques. Les facteurs techniques rendent un système vulnérable comme, par exemple, une mauvaise configuration du système, un système qui n'est pas maintenu à jour, ou une vulnérabilité d'un composant (logiciel ou matériel). Les attaquants peuvent exploiter ces vulnérabilités pour violer la politique de sécurité du système (c.-à-d., réaliser une intrusion) afin d'atteindre leurs buts (p. ex., voler des données confidentielles). Les raisons économiques sont les différentes motivations qui guident les décisions des organisations ou des individus — attaquants ou défenseurs.[150] Par exemple, le questionnement d'un attaquant pourrait être : les bénéfices d'une intrusion sont-ils supérieurs aux coûts engendrés ?

En prenant en compte ces facteurs, nous en concluons que nous devons supposer qu'à terme une intrusion aura lieu. C'est pourquoi nous devons non seulement construire des systèmes capables de prévenir des intrusions, mais aussi capables de *détecter* et *survivre* à ces intrusions.

Incapacité des systèmes d'exploitation à survivre aux intrusions

L'idée d'un système capable de détecter automatiquement des intrusions, abrégé IDS en anglais (pour *Intrusion Detection System*), date des années 1980.[151] De nos jours les IDSs sont présents dans la plupart des systèmes d'exploitation, abrégés OSs en anglais (pour *Operating Systems*), par exemple sous la forme d'un antivirus,[152] ou sont implantés au sein du réseau [217]. Cependant, comme leur nom l'indique, les IDSs se concentrent uniquement sur la détection et ne permettent pas aux OSs de survivre ou résister à une intrusion.

Malheureusement, bien que le concept de tolérance aux intrusions date lui aussi des années 1980 [116], la plupart des travaux de recherche se sont concentrés sur les systèmes critiques, distribués, ou sur la résilience d'un réseau. Ainsi, la capacité des OSs généralistes à survivre à une intrusion — après que celle-ci a été détectée — est limitée.

Des solutions existent afin de restaurer le système dans un état sain [122, 153, 197, 277, 286] ou limiter l'impact d'une intrusion [20, 113, 119, 239, 241], mais ces solutions sont limitées. Par exemple, elles peuvent engendrer une perte de disponibilité, impliquer un fort coût en performance, et nécessiter des modifications dans les applications, ou simplement ne pas être capables de survivre face aux intrusions après une restauration. Nous arrivons donc à la conclusion que les solutions de l'état de l'art, provenant du monde académique ou industriel, ne permettent pas à un système d'exploitation généraliste de survivre aux intrusions.

[150] Voir les travaux de Anderson [8] et Anderson et Moore [9] qui étudient ces aspects économiques.

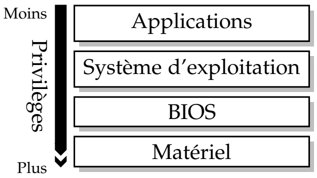
[151] Voir les travaux de Anderson [7] et Denning [90].

[152] Voir les travaux de Morin et Mé [203] qui ont étudié les similarités et différences entre un IDS et un antivirus.

Ciblage accru des composants de bas niveau

Néanmoins, la sécurité des OSs et de leurs applications s'est grandement améliorée depuis les 1980 ou 1990. Ces améliorations rendent la tâche des attaquants plus difficile. Il est désormais plus compliqué d'infecter un système, ou une application, sans être repéré ou sans laisser de traces. C'est pourquoi certains attaquants — plus sophistiqués et ayant plus de ressources — s'attaquent aux composants de bas niveau.

Plus spécifiquement, un de ces composants, le BIOS (abréviation de *Basic Input/Output System* en anglais) est une cible particulièrement intéressante. Il s'agit du composant logiciel responsable de la configuration du matériel au démarrage de l'ordinateur, juste après avoir appuyé sur le bouton démarrage. En effet, en raison de son accès direct au matériel, et comme il s'agit d'un composant qui s'exécute en premier sur la plateforme, le BIOS est hautement privilégié. Il est aussi responsable de la mise en place d'un des modes d'exécution les plus privilégiés de l'architecture x86 : le *System Management Mode* (SMM). Garantir l'intégrité du BIOS et du code s'exécutant en SMM est donc primordial pour maintenir l'intégrité du reste de la plateforme. Si un attaquant arrive à compromettre le BIOS, il peut compromettre n'importe quel composant qui le suit, tel que le système d'exploitation, et donc rendre contournable toute solution de sécurité implémentée au niveau de l'OS.[153]



Couches d'abstraction

Par exemple, en 2018, le groupe nommé APT28, Fancy Bear, ou Sednit,[154] a utilisé ce type d'attaque [226]. Cela lui permettait d'avoir un logiciel malveillant furtif implanté dans le BIOS qui était capable de survivre face à une réinstallation de l'OS ou un remplacement du disque dur.

Malheureusement, bien que de nombreuses solutions existent pour garantir l'intégrité du BIOS et des composants qui le suivent au démarrage [68, 130, 225, 230, 268, 272], peu de travaux se sont intéressés à la sécurité du SMM. Soit ils reposent sur des principes de sécurité préventive qui — comme nous l'avons expliqué précédemment — ne suffisent pas, soit ils offrent des capacités de détection, mais celles-ci sont limitées (p. ex., faussé sémantique, impossibilité de détecter des attaques inconnues, ou rigidité de l'approche de détection). Nous en concluons que les composants de bas niveau de nos plateformes sont, pour le moment, incapables de détecter si une intrusion a eu lieu. Ce constat est particulièrement inquiétant, car ces composants ne sont pas nécessairement mis à jour[155] et la durée de vie ou d'utilisation d'une plateforme avec ces composants se compte en années.

---

[153] Au meilleur de notre connaissance, Duflot et co-auteurs [97, 98] furent les premiers à considérer le SMM comme moyen de contourner les sécurités du système d'exploitation.

[154] Ils sont soupçonnés d'être les responsables de l'attaque sur le Comité national démocrate américain [3] et la chaîne de télévision française TV5 Monde [273].

[155] Ainsi toutes les vulnérabilités qui sont présentes et connues ne seront pas corrigées.

Thèse soutenue par ce manuscrit

Ce manuscrit soutient que les plateformes peuvent être construites afin qu'elles détectent des intrusions au niveau du BIOS et survivent automatiquement aux intrusions au niveau du système d'exploitation sans impacter significativement la qualité de service de l'utilisateur. Dans un premier temps, nous démontrons qu'une approche de survivabilité aux intrusions est viable et praticable pour des OSs généralistes. Dans un second temps, nous démontrons qu'il est possible de détecter des intrusions au niveau du BIOS avec une solution basée sur du matériel.

CONTRIBUTIONS

Nos travaux contribuent à l'amélioration de la survivabilité aux intrusions au niveau de la couche du système d'exploitation et des applications, et à l'amélioration de la détection d'intrusion au niveau de la couche du BIOS. Ces travaux ont fait l'objet de publications à ACSAC en 2017 [60], RESSI en 2018 [59], et ACSAC en 2019 [58].

Survivabilité des systèmes d'exploitation généralistes aux intrusions

Notre première contribution répond à la question suivante : comment concevoir un OS de façon que ses services puissent survivre aux intrusions tout en maintenant leur disponibilité ?

L'approche que nous proposons se distingue de l'état de l'art en trois points. Nous *combinons* la restauration de l'état des fichiers et des processus d'un service avec la capacité d'appliquer des réponses qui limitent l'impact de l'attaquant dans le cas d'une nouvelle compromission. Nous restaurons et appliquons des réponses *par service* qui vont impacter seulement le service qui a été compromis et n'impacteront pas le reste du système. Nous mettons le service qui a été compromis dans un *état dégradé* après avoir restauré son état (fichiers et processus). Pour ce faire, nous retirons des privilèges qui sont normalement utilisés par le service (p. ex., la capacité d'écrire dans un dossier ou un fichier).

L'état dégradé est volontaire. Lorsqu'une intrusion est détectée, nous n'avons pas d'information précise concernant les vulnérabilités exploitées afin de les corriger, ou nous n'avons pas de correctif disponible. L'état dégradé permet au service et au système de survivre à l'intrusion pour deux raisons. Premièrement, cela empêche l'attaquant, soit de réinfecter le service, soit d'effectuer des actions malveillantes (il n'a, en effet, plus les privilèges nécessaires). Deuxièmement, l'état dégradé maximise le nombre de fonctionnalités disponibles au sein du service, cela permettant d'attendre le ou les correctifs nécessaires.

Afin de maximiser le nombre de fonctionnalités disponibles, nous avons mis en place un modèle permettant de calculer les réponses optimales à appliquer. Ce modèle vise à limiter le coût des réponses — en termes de disponibilité — tout en maximisant leur efficacité face à l'intrusion. Bien que ce type de modèle — basé sur le coût — ne soit pas nouveau, notre manière de le calculer l'est. En effet, les méthodes précédentes se basent principalement sur des graphes d'attaques ou de vulnérabilités [113, 151, 239, 241].

Dans notre approche, nous utilisons la notion de comportement malveillant. Nous associons un comportement malveillant[156] à un coût, les réponses possibles pour contrer ce comportement à un coût, et nous calculons un risque en utilisant une matrice de risque qui prend en compte la probabilité de l'intrusion et le coût du comportement malveillant. Nous utilisons ensuite une méthode d'optimisation multi-objectifs afin de déterminer la réponse optimale. Nous déterminons la priorité des objectifs à l'aide d'un poids. Le poids est le niveau de risque ; plus le risque est fort et plus la solution sera choisie afin de favoriser l'efficacité de la réponse. À l'inverse, plus le risque est faible et plus la solution sera choisie afin de minimiser le coût de la réponse.

Nous avons développé un prototype de notre approche de survivabilité sur un système basé sur le noyau Linux.[157] Nous avons utilisé ce prototype pour nos expériences. Grâce à celles-ci, nous avons tout d'abord pu déterminer que notre approche est capable de résister à différents types d'intrusions et que les services qui sont dans un état dégradé fonctionnent toujours. Ensuite, nous avons pu déterminer que notre modèle permet de sélectionner des réponses adaptées à l'intrusion et au service. Enfin, nous avons observé que le coût en performance est négligeable excepté dans le cas de services qui effectuent de nombreuses écritures de petite taille en rafale sur le système de fichiers.

Détection d'intrusion au niveau du SMM

Notre seconde contribution répond à la question suivante : comment détecter des intrusions au sein de composants de bas niveau tel que le SMM ?

L'approche que nous proposons repose sur quatre éléments : un coprocesseur, un canal de communication restreint, une instrumentation du code cible, et un modèle du comportement attendu de la cible. Le coprocesseur — avec sa mémoire dédiée — permet d'isoler le moniteur dans un environnement d'exécution distinct de la cible. Ainsi, si l'attaquant compromet la cible, il n'a pas d'accès direct à la mémoire du moniteur, ni n'a de contrôle direct sur le moniteur. Cependant, cette isolation introduit un faussé sémantique — il nous est difficile d'avoir accès à l'état de la cible (p. ex., ses registres ou ce qu'elle exécute). Le canal de communication résout ce problème en permettant l'envoi d'information liées au comportement actuel de la cible à notre moniteur. Cependant, celui-ci doit garantir l'intégrité des messages quand ils ont été envoyés et nécessite une faible latence pour ne pas impacter significativement la cible qui envoie ces messages. Nous imposons cette transmission d'information en instrumentant le code de la cible. Cette instrumentation permet ainsi d'éviter de modifier le processeur au niveau du matériel. Enfin, le modèle qui permet de détecter si une intrusion a eu lieu — en déterminant si le comportement observé n'est pas celui attendu — est généré après analyse de la cible.

---

[156] Le projet *Malware Attribute Enumeration and Characterization (MAEC)* [155] propose une liste des comportements malveillants les plus répandus sous forme de hiérarchie [201].

[157] Pour ce faire, nous nous sommes basés sur des projets tels que systemd [253], CRIU [76], snapper [247], ou encore Linux audit [141].

Comme cas d'utilisation, nous avons choisi d'étudier et d'évaluer notre approche avec le code du BIOS s'exécutant en SMM. Le SMM est un mode d'exécution très privilégié des processeurs x86 [139]. De nos jours, il est utilisé pour exécuter des services privilégiés[158] du BIOS — durant l'exécution du système d'exploitation — dans un environnement de confiance. En effet, le code qui s'exécute en SMM est stocké dans une zone protégée de la mémoire appelée SMRAM. Seul le code qui s'exécute en SMM peut accéder au SMRAM — le système d'exploitation ne le peut donc pas. De plus, seul le code qui s'exécute en SMM peut écrire dans la flash du BIOS. Pour passer en mode SMM, une interruption nommée SMI doit être déclenchée, qui ensuite bascule sur l'exécution du gestionnaire d'interruption associé. Malheureusement, comme le reste du système est en pause durant la gestion de cette interruption, le gestionnaire ne doit pas prendre plus de 150 µs pour traiter l'interruption [135].

Pour détecter des intrusions, nous utilisons deux modèles. Le premier modèle repose sur le graphe de flot de contrôle (abrégé CFG en anglais pour *Control-Flow Graph*). Nous vérifions que chaque appel indirect cible une fonction avec un type valide (celui de l'appel indirect) et nous vérifions que chaque retour de fonction utilise l'adresse préalablement stockée sur la pile. Pour la vérification des types, nous analysons le code source pour extraire les types des appels indirects et des fonctions, puis nous instrumentons le code pour envoyer un message à chaque appel indirect contenant un identifiant unique associé à l'appel et l'adresse utilisée. Pour les retours de fonction, nous instrumentons le code pour envoyer un message au prologue et à l'épilogue de chaque fonction. Ainsi, le moniteur peut recréer une pile permettant de vérifier les adresses (*shadow call stack* en anglais) et vérifier que chaque appel indirect est valide. Le second modèle repose sur des invariants liés à des registres qui permettent de modifier le contexte d'exécution. Dans notre cas, nous nous sommes intéressés à CR3 (qui est l'adresse physique du quatrième niveau des pages utilisées pour la traduction d'adresses virtuelles) et à SMBASE (qui est utilisé pour calculer le point d'entrée en SMM).[159] Les valeurs de ces registres sont temporairement stockées en mémoire et peuvent être modifiées par un attaquant. Nous vérifions à chaque SMI que les valeurs des registres n'ont pas été modifiées, car il n'y a aucune

[158] Par exemple, éteindre l'ordinateur dans le cas d'une surchauffe du système, la mise à jour du BIOS dans la flash et sa protection, ou encore la gestion sécurisée de variables UEFI [139, 291].

[159] Des attaques ont été montrées exploitant ces registres [143, 223].

raison légitime qu'elles le soient après la configuration du système au démarrage.

Pour le coprocesseur, nous nous reposons sur un ARM Cortex A5 [14].[160] Il nous permet d'isoler le moniteur et a suffisamment de puissance de calcul pour traiter les messages.

Nous avons étudié les mécanismes existants de communication entre deux processeurs et aucun ne répondait à nos exigences en termes de sécurité et de performance.[161] Nous introduisons donc un nouveau composant qui fonctionne comme une file d'attente de type « premier entré, premier sorti », abrégé FIFO en anglais (pour *First In First Out*), mais restreinte. En effet, le FIFO n'acceptera que l'entrée de messages de la part de la cible et interdira de retirer ou de modifier leur contenu — garantissant ainsi leur intégrité. Pour répondre à la problématique de la latence, nous utilisons un bus à faible latence pour connecter le processeur de la cible au FIFO, ainsi que pour le coprocesseur. Intel QPI et AMD HyperTransport sont deux exemples de bus à faible latence.[162]

Afin d'instrumenter le code SMM, nous avons développé deux modules d'instrumentation en utilisant l'infrastructure LLVM [165]. Le premier analyse le code qui s'exécute en SMM et l'instrumente pour la politique visant à garantir l'intégrité des appels indirects. Le second instrumente le code pour garantir l'intégrité de l'adresse de retour stockée sur la pile. Nous instrumentons ensuite manuellement le code pour envoyer la valeur stockée en mémoire de CR3 et SMBASE au moniteur.

Pour évaluer notre approche, nous avons utilisé un émulateur et un simulateur.[163] Grâce aux différentes expériences menées, nous avons pu déterminer que notre approche est capable de détecter des attaques de l'état de l'art tout en respectant la contrainte du seuil des 150 µs. Le moniteur mettait au maximum 230 µs pour traiter les messages reçus durant l'exécution d'un SMI. Enfin la taille du BIOS n'a augmenté que de 0.6 % en raison de l'instrumentation d'une partie du code.

TRAVAUX FUTURS

Ces travaux nous ont permis d'établir les premières bases d'une plateforme capable de détecter et survivre aux intrusions. Cependant, durant nos recherches, nous avons concentré notre analyse et nos expériences sur des points spécifiques. D'autres aspects restent donc à étudier.

Dans un premier temps, nous envisageons d'évaluer des aspects supplémentaires de nos approches. Par exemple, évaluer la capacité de nos approches à se généraliser à d'autres cas d'utilisation tels que l'observation du comportement du code s'exécutant en ARM TrustZone *secure world* [13] — un environnement proche du SMM — et appliquer l'approche de survivabilité à Windows. Il serait aussi

[160] D'autres architectures nécessitant un coprocesseur de sécurité reposent sur ce type de processeur [5, 186].

[161] Par exemple, le mécanisme de communication par boîte aux lettres (ou *mailbox* en anglais) aurait nécessité d'attendre un accusé de réception de la part du moniteur avant de continuer l'exécution pour garantir l'intégrité du message. Malheureusement, cette attente aurait donné en moyenne une latence de 7500 cycles [244].

[162] Ils sont généralement utilisés pour connecter des processeurs entre eux ou un processeur avec le *chipset*.

[163] Au meilleur de notre connaissance, au moment où les expériences ont été réalisées, il n'existait aucune carte FPGA ayant accès au bus AMD HyperTransport ou Intel QPI disponible à la vente. Certaines furent disponible par le passé, mais elles ont été retirées de la vente.

intéressant d'améliorer certains aspects de nos approches. Par exemple, nous souhaitons ajouter d'autres modèles de détection pour le SMM, et améliorer l'isolation des composants utilisés pour la survivabilité du système (p. ex., les composants en charge de la restauration et de l'application des réponses).[164]

Dans un second temps, nous souhaitons nous intéresser à ce qui suit la capacité de survivre à une intrusion : la capacité d'adaptation. C'est-à-dire, comment faire en sorte que la plateforme soit en mesure de ne plus être vulnérable afin de retirer les réponses qui ont été appliquées ? En effet, les tâches d'analyse de l'intrusion, de détermination des vulnérabilités exploitées, de développement, et d'application d'un correctif s'effectuent actuellement manuellement. Nous souhaitons rendre automatiques ces tâches afin qu'à terme nous puissions corriger automatiquement le système et retirer automatiquement les réponses mises en place.

[164] Pour le moment, nous nous reposons entièrement sur le noyau du système d'exploitation pour les isoler. Cependant, les noyaux sont connus pour avoir des vulnérabilités avec une longue durée de vie pouvant aller jusqu'à 5 ans en moyenne [65, 72].

BIBLIOGRAPHY

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. "Control-flow integrity". In: *Proceedings of the 12th ACM conference on Computer and Communications Security* (Alexandria, VA, USA). CCS '05. ACM, 2005, pp. 340–353. DOI: `10.1145/1102120.1102165` (cit. on pp. 30, 31, 38, 39).

[2] Lillian Ablon and Andy Bogart. *Zero Days, Thousands of Nights: The life and Times of Zero-Day Vulnerabilities and Their Exploits*. RAND Corporation, 2017. DOI: `10.7249/RR1751` (cit. on p. 51).

[3] Dmitri Alperovitch. *Bears in the Midst: Intrusion into the Democratic National Committee*. CrowdStrike. June 15, 2016. URL: `https://www.crowdstrike.com/blog/bears-midst-intrusion-democratic-national-committee/` (visited on 08/05/2019) (cit. on pp. 6, 151).

[4] AMD. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 16h Models 30h-3Fh Processors*. Advanced Micro Devices, Inc. Mar. 2016 (cit. on p. 120).

[5] AMD TATS BIOS Development Group. "AMD Security and Server innovation". UEFI PlugFest. Mar. 2013 (cit. on pp. 119, 155).

[6] James P. Anderson. *Computer Security Technology Planning Study*. Tech. rep. ESD-TR-73-51. Volume 1. Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), Oct. 1972 (cit. on p. 30).

[7] James P. Anderson. *Computer Security Threat Monitoring and Surveillance*. Tech. rep. James P. Anderson Co., Fort Washington, PA. Apr. 1980 (cit. on pp. 5, 26, 150).

[8] Ross Anderson. "Why information security is hard – an economic perspective". In: *17th Annual Computer Security Applications Conference*. IEEE. Dec. 2001, pp. 358–365. DOI: `10.1109/ACSAC.2001.991552` (cit. on pp. 5, 150).

[9] Ross Anderson and Tyler Moore. "Information security: where computer science, economics and psychology meet". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 367.1898 (2009), pp. 2717–2727. DOI: `10.1098/rsta.2009.0027` (cit. on pp. 5, 150).

[10] Android Developers. *Understand the Activity Lifecycle*. 2019. URL: `https://developer.android.com/guide/components/activities/activity-lifecycle` (visited on 10/09/2019) (cit. on p. 95).

[11] Louis Anthony Tony Cox. "What's wrong with risk matrices?" In: *Risk Analysis* 28.2 (2008), pp. 497–512. DOI: `10.1111/j.1539-6924.2008.01030.x` (cit. on p. 70).

[12] *Apache HTTP Server*. URL: `https://httpd.apache.org/` (visited on 08/05/2019) (cit. on p. 84).

[13] ARM. *ARM Security Technology: Building a Secure System using TrustZone Technology*. ARM. Apr. 2009 (cit. on pp. 14, 105, 106, 136, 155).

[14] ARM. *ARM Cortex-A5 Technical Reference Manual*. ARM. Jan. 2016 (cit. on pp. 120, 147, 155).

[15] ARM. *AMBA Specifications*. 2017. URL: `https://www.arm.com/products/system-ip/amba-specifications` (visited on 05/05/2019) (cit. on p. 122).

[16] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004), pp. 11–33. DOI: `10.1109/TDSC.2004.2` (cit. on pp. 28, 29, 42).

[17] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. "Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. ACM, 2014, pp. 90–102. DOI: `10.1145/2660267.2660350` (cit. on pp. 44, 62).

[18] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. "HyperSentry: enabling stealthy in-context measurement of hypervisor integrity". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security* (Chicago, IL, USA). CCS '10. ACM, Oct. 2010, pp. 38–49. DOI: `10.1145/1866307.1866313` (cit. on pp. 32, 33, 135).

[19] Andrei Bacs, Remco Vermeulen, Asia Slowinska, and Herbert Bos. "System-level support for intrusion recovery". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 2012, pp. 144–163. ISBN: 978-3-642-37300-8 (cit. on pp. 43, 46, 47).

[20] Ivan Balepin, Sergei Maltsev, Jeff Rowe, and Karl Levitt. "Using Specification-Based Intrusion Detection for Automated Response". In: *Recent Advances in Intrusion Detection*. 2003, pp. 136–154. DOI: `10.1007/978-3-540-45248-5_8` (cit. on pp. 6, 49–51, 53, 54, 96–98, 150).

[21] Sean Barnum. "Standardizing cyber threat intelligence information with the Structured Threat Information eXpression (STIX)". Feb. 2014 (cit. on pp. 69, 77, 96).

[22] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. "A Survey on Hypervisor-Based Monitoring: Approaches, Applications, and Evolutions". In: *ACM Computing Surveys* 48.1 (Aug. 2015), 10:1–10:33. DOI: `10.1145/2775111` (cit. on p. 31).

[23] Oleksandr Bazhaniuk, Yuriy Bulygin, Andrew Furtak, Mikhail Gorobets, John Loucaides, Alexander Matrosov, and Mickey Shkatov. "A new class of vulnerabilities in SMI handlers" (Vancouver, B.C., Canada). CanSecWest. 2015 (cit. on pp. 17, 18, 103).

[24] *beanstalkd*. URL: `https://kr.github.io/beanstalkd/` (visited on 08/05/2019) (cit. on p. 84).

[25] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA, USA). ATC '05. USENIX Association, 2005, pp. 41–46 (cit. on p. 126).

[26] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. "The gem5 simulator". In: *ACM SIGARCH Computer Architecture News* 39.2 (2011), pp. 1–7. DOI: `10.1145/2024716.2024718` (cit. on p. 126).

[27] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. "Jump-oriented programming: a new class of code-reuse attack". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (Hong Kong, China). ASIACCS '11. ACM, 2011, pp. 30–40. DOI: `10.1145/1966913.1966919` (cit. on p. 38).

[28] Deborah Bodeau and Richard Graubart. *Cyber Resiliency Engineering Framework*. Tech. rep. MITRE, Sept. 2011 (cit. on p. 28).

[29] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. "On the Importance of Checking Cryptographic Protocols for Faults". In: *Proceedings of the 16th Annual International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT'97. Konstanz, Germany, 1997, pp. 37–51. DOI: `10.1007/3-540-69053-0_4` (cit. on p. 63).

[30] Herbert Bos. *Operating Systems & Virtualization Security*. Tech. rep. Knowledge Area. July 2019 (cit. on p. 22).

[31] Brian M. Bowen, Pratap V. Prabhu, Vasileios P. Kemerlis, Stelios Sidiroglou, Angelos D. Keromytis, and Salvatore J. Stolfo. "BotSwindler: Tamper Resistant Injection of Believable Decoys in VM-Based Hosts for Crimeware Detection". In: *Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection*. RAID '10. Ottawa, Ontario, Canada, 2010, pp. 118–137. DOI: `10.1007/978-3-642-15512-3_7` (cit. on p. 64).

[32] Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina. "Exploit Programming: From Buffer Overflows to "Weird Machines" and Theory of Computation". In: *USENIX ;login:* 36.6 (Dec. 2011) (cit. on p. 38).

[33] Yuriy Bulygin, Oleksandr Bazhaniuk, Andrew Furtak, John Loucaides, and Mikhail Gorobets. "BARing the System: New vulnerabilities in Coreboot & UEFI based systems". REcon Brussels. 2017 (cit. on pp. 17, 18, 103).

[34] Yuriy Bulygin and David Samyde. "Chipset based approach to detect virtualization malware". Black Hat USA. 2008 (cit. on pp. 31, 32, 35, 41, 134, 135).

[35] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. "Control-Flow Integrity: Precision, Security, and Performance". In: *ACM Comput. Surv.* 50.1 (Apr. 2017), 16:1–16:33. DOI: `10.1145/3054924` (cit. on p. 39).

[36] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. "Control-flow integrity: Precision, security, and performance". In: *ACM Computing Surveys (CSUR)* 50.1 (2017), p. 16. DOI: `10.1145/3054924` (cit. on pp. 112, 128).

[37] Nathan Burow, Xinping Zhang, and Mathias Payer. "SoK: Shining Light on Shadow Stacks". In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2019, pp. 985–999. DOI: `10.1109/SP.2019.00076` (cit. on p. 117).

[38] Anastasiia Butko, Florent Bruguier, Abdoulaye Gamatié, Gilles Sassatelli, David Novo, Lionel Torres, and Michel Robert. "Full-System Simulation of big. LITTLE Multicore Architecture for Performance and Energy Exploration". In: *Proceedings of the 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE Computer Society, Sept. 2016, pp. 201–208. DOI: `10.1109/MCSoC.2016.20` (cit. on p. 134).

[39] James Butler and Greg Hoglund. "Hidden Processes: The Implication for Intrusion Detection". Black Hat USA. 2004 (cit. on p. 37).

[40] Canonical, Novell, and Immunix. *AppArmor*. URL: `https://gitlab.com/apparmor` (visited on 08/05/2019) (cit. on p. 22).

[41] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. "Control-flow bending: On the effectiveness of control-flow integrity". In: *Proceedings of the 24th USENIX Security Symposium* (Washington, D.C., USA). SEC'15. USENIX Association, 2015, pp. 161–176 (cit. on p. 40).

[42] Curtis A. Carver Jr. "Adaptive agent-based intrusion response". PhD thesis. Texas A&M University, 2001 (cit. on p. 49).

[43] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. "On the Difficulty of Software-based Attestation of Embedded Devices". In: *Proceedings of the 16th ACM conference on Computer and communications security* (Chicago, Illinois, USA). CCS '09. ACM, 2009, pp. 400–409. DOI: `10.1145/1653662.1653711` (cit. on p. 36).

[44] Miguel Castro, Manuel Costa, and Tim Harris. "Securing Software by Enforcing Data-flow Integrity". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. USENIX Association, 2006, pp. 147–160 (cit. on pp. 136, 141).

[45] CERT C Coding Standard. *ERR00-C. Adopt and implement a consistent and comprehensive error-handling policy*. Aug. 30, 2019. URL: `https://wiki.sei.cmu.edu/confluence/display/c/ERR00-C.+Adopt+and+implement+a+consistent+and+comprehensive+error-handling+policy` (visited on 10/09/2019) (cit. on p. 93).

[46] CERT C Coding Standard. *EXP12-C. Do not ignore values returned by functions*. Aug. 30, 2019. URL: `https://wiki.sei.cmu.edu/confluence/display/c/EXP12-C.+Do+not+ignore+values+returned+by+functions` (visited on 10/09/2019) (cit. on p. 93).

[47] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. "Return-oriented programming without returns". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security* (Chicago, IL, USA). CCS '10. ACM, Oct. 2010, pp. 559–572. DOI: `10.1145/1866307.1866370` (cit. on p. 38).

[48] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. "Linux kernel vulnerabilities: State-of-the-art defenses and open problems". In: *Proceedings of the Second Asia-Pacific Workshop on Systems*. ACM. 2011, 5:1–5:5. DOI: `10.1145/2103799.2103805` (cit. on p. 20).

[49] Peter M. Chen and Brian D. Noble. "When Virtual Is Better Than Real". In: *Proceedings Eighth Workshop on Hot Topics in Operating Systems*. May 2001, pp. 133–138. DOI: `10.1109/HOTOS.2001.990073` (cit. on pp. 31, 105, 107).

[50] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. "Non-Control-Data Attacks Are Realistic Threats". In: *Proceedings of the 14th USENIX Security Symposium*. USENIX Association, 2005 (cit. on p. 136).

[51] Xu Chen, Jon Andersen, Z. Morley Mao, Michael Bailey, and Jose Nazario. "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware". In: *38th IEEE/IFIP International Conference On Dependable Systems and Networks*. June 2008, pp. 177–186. DOI: `10.1109/DSN.2008.4630086` (cit. on p. 83).

[52] Ronny Chevalier. *bus-unit-util: fix parsing of IPAddress{Allow,Deny}*. systemd git repository. Sept. 21, 2018. URL: `https://github.com/systemd/systemd/commit/afc1feaeba27c814134376374555cf6a1f4dc874` (visited on 10/07/2019) (cit. on p. 81).

[53] Ronny Chevalier. *audit: fix use-after-free in audit_add_watch*. July 18, 2018. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=baa2a4fdd525c8c4b0f704d20457195b29437839 (visited on 10/07/2019) (cit. on p. 81).

[54] Ronny Chevalier. *btrfs: do not wrongly report failure when doing btrfs comparison*. snapper git repository. Oct. 12, 2018. URL: https://github.com/openSUSE/snapper/pull/438 (visited on 10/07/2019) (cit. on p. 81).

[55] Ronny Chevalier. *lib/c: add const qualifier to criu_set_service_binary*. CRIU git repository. June 18, 2018. URL: https://github.com/checkpoint-restore/criu/commit/3ec82d4b6aa9badbc8f78da95f7375fd050b2953 (visited on 10/07/2019) (cit. on p. 81).

[56] Ronny Chevalier. *lib/c: add missing criu_local_set_service_binary signature*. CRIU git repository. June 18, 2018. URL: https://github.com/checkpoint-restore/criu/commit/d71fc8dc08325a2090acd901746de32c7e739b01 (visited on 10/07/2019) (cit. on p. 81).

[57] Ronny Chevalier. *shared: do not include ~when appending syscall filters property*. systemd git repository. June 18, 2018. URL: https://github.com/systemd/systemd/commit/98008caa94a7aca76297ecdca86a23f460386429 (visited on 10/07/2019) (cit. on p. 81).

[58] Ronny Chevalier, David Plaquin, Chris Dalton, and Guillaume Hiet. "Survivor: A Fine-Grained Intrusion Response and Recovery Approach for Commodity Operating Systems". In: *Proceedings of the 35th Annual Computer Security Applications Conference*. ACSAC'19. ACM, Dec. 2019. DOI: 10.1145/3359789.3359792 (cit. on pp. 8, 99, 152).

[59] Ronny Chevalier, David Plaquin, and Guillaume Hiet. "Intrusion Survivability for Commodity Operating Systems and Services: A Work in Progress". In: *Rendez-vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information*. RESSI'18. May 2018 (cit. on pp. 8, 98, 152).

[60] Ronny Chevalier, Maugan Villatel, David Plaquin, and Guillaume Hiet. "Co-processor-based Behavior Monitoring: Application to the Detection of Attacks Against the System Management Mode". In: *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACSAC'17. ACM, Dec. 2017, pp. 399–411. DOI: 10.1145/3134600.3134622 (cit. on pp. 8, 62, 136, 152).

[61] *CHIPSEC: Platform Security Assessment Framework*. URL: https://github.com/chipsec/chipsec (visited on 08/05/2019) (cit. on pp. 19, 109).

[62] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. "A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms". In: *Proceedings of the 53rd Annual Design Automation Conference* (Austin, TX, USA). DAC '16. ACM, 2016, 109:1–109:6. ISBN: 978-1-4503-4236-0. DOI: `10.1145/2897937.2897972` (cit. on p. 126).

[63] Daniel Cid. *OSSEC*. URL: `https://www.ossec.net/` (visited on 08/05/2019) (cit. on p. 35).

[64] core collapse. *ASUS Eee PC and other series: BIOS SMM privilege escalation vulnerabilities*. Aug. 2009. URL: `http://www.securityfocus.com/archive/1/505590` (visited on 08/05/2019) (cit. on pp. 17, 18, 103).

[65] Kees Cook. *Security bug lifetime*. Oct. 18, 2016. URL: `https://outflux.net/blog/archives/2016/10/18/security-bug-lifetime/` (visited on 10/04/2019) (cit. on pp. 43, 156).

[66] Kees Cook. "Status of the Kernel Self Protection Project". Linux Security Summit. Aug. 2016 (cit. on p. 43).

[67] Kees Cook. "The State of Kernel Self Protection". Linux Conf AU. Jan. 2018 (cit. on p. 43).

[68] David Cooper, William Polk, Andrew Regenscheid, and Murugiah Souppaya. *BIOS protection guidelines*. Tech. rep. NIST Special Publication 800-147. National Institute of Standards and Technology, Apr. 2011. DOI: `10.6028/NIST.SP.800-147` (cit. on pp. 12, 109, 151).

[69] Jonathan Corbet. "vmsplice(): the making of a local root exploit". In: *LWN* (Feb. 12, 2008) (cit. on p. 22).

[70] Jonathan Corbet. "Seccomp and sandboxing". In: *LWN* (May 13, 2009) (cit. on pp. 78, 80).

[71] Jonathan Corbet. "Checkpoint/restart: it's complicated". In: *LWN* (Nov. 9, 2010) (cit. on p. 44).

[72] Jonathan Corbet. "Kernel vulnerabilities: old or new?" In: *LWN* (Oct. 19, 2010) (cit. on pp. 43, 156).

[73] Jonathan Corbet. "Kernel security: beyond bug fixing". In: *LWN* (Oct. 28, 2015) (cit. on p. 43).

[74] Thomas Coudray, Arnaud Fontaine, and Pierre Chifflier. "Picon: Control Flow Integrity on LLVM IR". In: *Symposium sur la Sécurité des Technologies de l'Information et des Communications* (Rennes, France). SSTIC'15. 2015 (cit. on pp. 31, 39).

[75] Emanuele Cozzi, Mariano Graziano, Yannick Fratantonio, and Davide Balzarotti. "Understanding Linux Malware". In: *Proceedings of the 2018 IEEE Symposium on Security and Privacy*. May 2018, pp. 161–175. DOI: `10.1109/SP.2018.00054` (cit. on p. 83).

[76] *CRIU*. 2019. URL: `https://criu.org/` (visited on 08/05/2019) (cit. on pp. 45, 79, 153).

[77] CryptoDrop, LLC. *CryptoDrop*. 2019. URL: `https://www.cryptodrop.org/` (visited on 08/05/2019) (cit. on pp. 86, 96).

[78] *CVE-2013-3582*. May 2013. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-3582` (visited on 08/05/2019) (cit. on p. 128).

[79] *CVE-2016-5195*. 2016. URL: `https://nvd.nist.gov/vuln/detail/CVE-2016-5195` (visited on 08/05/2019) (cit. on p. 22).

[80] *CVE-2016-7117*. 2016. URL: `https://nvd.nist.gov/vuln/detail/CVE-2016-7117` (visited on 08/05/2019) (cit. on p. 22).

[81] *CVE-2016-8103*. Sept. 2016. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8103` (visited on 08/05/2019) (cit. on p. 128).

[82] *CVE-2016-8655*. 2016. URL: `https://nvd.nist.gov/vuln/detail/CVE-2016-8655` (visited on 08/05/2019) (cit. on p. 22).

[83] *D-Bus*. 2019. URL: `https://www.freedesktop.org/wiki/Software/dbus/` (visited on 08/05/2019) (cit. on p. 77).

[84] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. "HAFIX: Hardware-assisted flow integrity extension". In: *Proceedings of the 52nd Annual Design Automation Conference*. ACM. 2015, p. 74. DOI: `10.1145/2744769.2744847` (cit. on p. 34).

[85] Lucas Davi and Fabian Monrose. "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection". In: *Proceedings of the 23rd USENIX Security Symposium* (San Diego, CA, USA). USENIX Association, Aug. 2014, pp. 401–416 (cit. on p. 40).

[86] Hervé Debar, Marc Dacier, and Andreas Wespi. "Towards a taxonomy of intrusion-detection systems". In: *Computer Networks* 31.8 (1999), pp. 805–822. DOI: `10.1016/S1389-1286(98)00017-6` (cit. on pp. 27, 30).

[87] Hervé Debar, Marc Dacier, and Andreas Wespi. "A revised taxonomy for intrusion-detection systems". In: *Annales Des Télécommunications* 55.7 (July 1, 2000), pp. 361–378. DOI: `10.1007/BF02994844` (cit. on pp. 27, 32, 34).

[88] Brian Delgado and Karen L. Karavanic. "Performance implications of System Management Mode". In: *IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, 2013, pp. 163–173. DOI: `10.1109/IISWC.2013.6704682` (cit. on pp. 16, 120, 129).

[89] Yong Deng, Rehan Sadiq, Wen Jiang, and Solomon Tesfamariam. "Risk analysis in a linguistic environment: a fuzzy evidential reasoning-based approach". In: *Expert Systems with Applications* 38.12 (2011), pp. 15438–15446. DOI: `10.1016/j.eswa.2011.06.018` (cit. on p. 72).

[90] Dorothy E. Denning. "An Intrusion-Detection Model". In: *Proceedings of the 1986 IEEE Symposium on Security and Privacy* (Oakland, CA, USA). IEEE Computer Society, Apr. 1986, pp. 118–131. DOI: `10.1109/SP.1986.10010` (cit. on pp. 5, 26, 150).

[91] Jack B. Dennis and Earl C. Van Horn. "Programming Semantics for Multiprogrammed Computations". In: *Communications of the ACM* 9.3 (Mar. 1966), pp. 143–155. DOI: `10.1145/365230.365252` (cit. on p. 22).

[92] systemd developpers. *systemd.exec(5) – Execution environment configuration* (cit. on p. 23).

[93] systemd developpers. *systemd.unit(5) – Unit configuration* (cit. on p. 78).

[94] Dr. Web. *Linux.Encoder.1*. Nov. 2015. URL: `https://vms.drweb.com/virus/?i=7703983` (visited on 08/05/2019) (cit. on pp. 84, 96).

[95] Dr. Web. *Linux.Rex.1*. Aug. 2016. URL: `https://vms.drweb.com/virus/?i=8436299` (visited on 08/05/2019) (cit. on pp. 84, 96).

[96] Dr. Web. *Linux.BackDoor.Fgt.1430*. July 2018. URL: `https://vms.drweb.com/virus/?i=17573534` (visited on 08/05/2019) (cit. on pp. 84, 96).

[97] Loïc Duflot, Daniel Etiemble, and Olivier Grumelard. "Using CPU System Management Mode to Circumvent Operating System Security Functions" (Vancouver, B.C., Canada). CanSecWest. 2006 (cit. on pp. 16, 151).

[98] Loïc Duflot, Daniel Etiemble, and Olivier Grumelard. "Utiliser les fonctionnalités des cartes mères ou des processeurs pour contourner les mécanismes de sécurité des systèmes d'exploitation". In: *Symposium sur la Sécurité des Technologies de l'Information et des Communications* (Rennes, France). SSTIC'06. 2006 (cit. on pp. 16, 151).

[99] Loïc Duflot, Olivier Levillain, Benjamin Morin, and Olivier Grumelard. "Getting into the SMRAM: SMM Reloaded" (Vancouver, B.C., Canada). CanSecWest. 2009 (cit. on pp. 16–18, 103, 109).

[100] Loïc Duflot, Olivier Levillain, Benjamin Morin, and Olivier Grumelard. "System Management Mode Design and Security Issues" (Cologne, Germany). IT Defense. Feb. 3, 2010 (cit. on pp. 17, 103).

[101] Loïc Duflot, Yves-Alexis Perez, and Benjamin Morin. "What if you can't trust your network card?" In: *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*. RAID'11. 2011, pp. 378–397. DOI: `10.1007/978-3-642-23644-0_20` (cit. on pp. 34, 36).

[102] Loïc Duflot, Yves-Alexis Perez, Guillaume Valadon, and Olivier Levillain. "Can you still trust your network card" (Vancouver, B.C., Canada). CanSecWest. 2010 (cit. on p. 34).

[103] Eclipse Foundation. *Mosquitto*. 2019. URL: `https://mosquitto.org/` (visited on 08/05/2019) (cit. on p. 84).

[104] Jake Edge. "A seccomp overview". In: *LWN* (Sept. 2, 2015) (cit. on pp. 22, 80).

[105] Robert J. Ellison, David A. Fisher, Richard C. Linger, Howard F. Lipson, and Thomas Longstaff. *Survivable Network Systems: An emerging discipline*. Tech. rep. Software Engineering Institute, Carnegie Mellon University, Nov. 1997 (cit. on pp. 5, 29, 42).

[106] Shawn Embleton, Sherri Sparks, and Cliff C Zou. "SMM rootkit: a new breed of OS independent malware". In: *Security and Communication Networks* 6.12 (2013), pp. 1590–1605. DOI: `10.1002/sec.166` (cit. on p. 103).

[107] Úlfar Erlingsson. "The Inlined Reference Monitor Approach to Security Policy Enforcement". PhD thesis. 2004 (cit. on p. 30).

[108] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. "Control jujutsu: On the weaknesses of fine-grained control flow integrity". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, CO, USA). ACM, 2015, pp. 901–913. DOI: `10.1145/2810103.2813646` (cit. on pp. 40, 112).

[109] Reza Mirzazade Farkhani, Saman Jafari, Sajjad Arshad, William Robertson, Engin Kirda, and Hamed Okhravi. "On the Effectiveness of Type-based Control Flow Integrity". In: *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA). ACSAC '18. ACM, 2018, pp. 28–39. DOI: `10.1145/3274694.3274739` (cit. on p. 133).

[110] Eric Fisch. "Intrusion Damage Control and Assessment: A Taxonomy and implementation of automated responses to intrusive behavior". PhD thesis. Texas A&M University, 1996 (cit. on p. 49).

[111] Matt Fleming. "A thorough introduction to eBPF". In: *LWN* () (cit. on p. 23).

[112] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. "An Empirical Study on the Correctness of Formally Verified Distributed Systems". In: *Proceedings of the 12th European Conference on Computer Systems* (Belgrade, Serbia). EuroSys '17. 2017, pp. 328–343. DOI: `10.1145/3064176.3064183` (cit. on p. 4).

[113] Bingrui Foo, Yu-Sung Wu, Yu-Chun Mao, Saurabh Bagchi, and Eugene H. Spafford. "ADEPTS: Adaptive Intrusion Response Using Attack Graphs in an E-Commerce Environment". In: *Proceedings of the International Conference on Dependable Systems and Networks*. DSN '05. 2005, pp. 508–517. DOI: `10.1109/DSN.2005.17` (cit. on pp. 50, 51, 53, 59, 65, 97, 98, 150, 153).

[114] Michael Frantzen and Michael Shuey. "StackGhost: Hardware Facilitated Stack Protection". In: *USENIX Security Symposium*. 2001 (cit. on p. 34).

[115] Ivan Fratrić. *ROPGuard: Runtime prevention of return-oriented programming attacks*. Tech. rep. 2012 (cit. on pp. 39, 40).

[116] J. Fray, Y. Deswarte, and D. Powell. "Intrusion-Tolerance Using Fine-Grain Fragmentation-Scattering". In: *Proceedings of the 1986 IEEE Symposium on Security and Privacy* (Oakland, CA, USA). Apr. 1986. DOI: `10.1109/SP.1986.10019` (cit. on pp. 5, 150).

[117] Holger Fröning, Mondrian Nüssle, Heiner Litz, Christian Leber, and Ulrich Brüning. "On achieving high message rates". In: *Proceedings of the 13th International Symposium on Cluster, Cloud and Grid Computing (CCGrid)* (Los Alamitos, CA, USA). IEEE Computer Society, May 2013, pp. 498–505. DOI: `10.1109/CCGrid.2013.43` (cit. on p. 121).

[118] Sean Gallagher. "Your USB cable, the spy: Inside the NSA's catalog of surveillance magic". In: *Ars Technica* (Dec. 31, 2013) (cit. on pp. 6, 103).

[119] Ashish Gehani and Gershon Kedem. "RheoStat: Real-Time Risk Management". In: *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection*. RAID '04. 2004, pp. 296–314. DOI: `10.1007/978-3-540-30143-1_16` (cit. on pp. 52, 53, 55, 97, 150).

[120] *Gitea*. 2019. URL: `https://gitea.io/` (visited on 08/05/2019) (cit. on p. 84).

[121] GitHub, Inc. *GitHub*. 2019. URL: `https://github.com/` (visited on 08/05/2019) (cit. on p. 85).

[122] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. "The Taser Intrusion Recovery System". In: *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom). SOSP '05. 2005, pp. 163–176. ISBN: 1-59593-079-5. DOI: `10.1145/1095810.1095826` (cit. on pp. 6, 43, 46, 47, 59, 97, 150).

[123] Gustavo Daniel Gonzalez Granadillo. "Optimization of cost-based threat response for Security Information and Event Management (SIEM) systems". PhD thesis. Institut National des Télécommunications, Dec. 2013 (cit. on p. 49).

[124] grsecurity. *grsecurity*. URL: `https://grsecurity.net/` (visited on 08/05/2019) (cit. on p. 43).

[125] Tejun Heo. *Control Group v2*. Oct. 2015. URL: `https://www.kernel.org/doc/Documentation/cgroup-v2.txt` (visited on 08/05/2019) (cit. on pp. 22, 78, 80).

[126] Daniel Hodson. *Remote LD_PRELOAD Exploitation*. Dec. 18, 2017. URL: `https://www.elttam.com.au/blog/goahead/` (visited on 08/05/2019) (cit. on p. 84).

[127] Brian Holden, Don Anderson, Jay Trodden, and Maryanne Daves. *HyperTransport 3.1 Interconnect Technology*. MindShare Press, Sept. 2008. ISBN: 978-0-9770-8782-2 (cit. on pp. 121, 122).

[128] HP Inc. *HP Sure Start Gen3*. Tech. rep. HP Inc., Jan. 2017 (cit. on p. 32).

[129] HP Inc. *HP Sure Start with Runtime Intrusion Detection*. Tech. rep. HP Inc., May 2017 (cit. on p. 32).

[130] HP Inc. *HP Sure Start: Automatic Firmware Intrusion Detection and Repair*. Tech. rep. HP Inc., Jan. 2019 (cit. on pp. 13, 32, 62, 104, 109, 151).

[131] Francis Hsu, Hao Chen, Thomas Ristenpart, Jason Li, and Zhendong Su. "Back to the Future: A Framework for Automatic Malware Removal and System Repair". In: *Proceedings of the 22nd Annual Computer Security Applications Conference*. ACSAC '06. 2006, pp. 257–268. ISBN: 0-7695-2716-7. DOI: `10.1109/ACSAC.2006.16` (cit. on pp. 43, 46, 59).

[132] Intel. *Intel® 7 Series / C216 Chipset Family Platform Controller Hub (PCH)*. Tech. rep. Intel, June 2012 (cit. on p. 15).

[133] Intel. *Intel Xeon Scalable Platform*. 2017. URL: `https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-scalable-platform-brief.pdf` (visited on 10/11/2019) (cit. on p. 121).

[134] Intel Corporation. "Introduction to the Intel Quickpath Interconnect". Jan. 2009 (cit. on p. 121).

[135] Intel Corporation. *bits-365*. Mar. 2011. URL: `https://biosbits.org/news/bits-365/` (visited on 08/05/2019) (cit. on pp. 16, 120, 129, 130, 154).

[136] Intel Corporation. "Control-flow Enforcement Technology Specification". May 2019 (cit. on pp. 34, 39–41, 133–135).

[137] Intel Corporation. "Instruction Set Summary". In: *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Jan. 2019. Chap. 5 (cit. on p. 21).

[138] Intel Corporation. "Model-Specific Registers". In: *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Jan. 2019. Chap. 2 (cit. on pp. 17, 18).

[139] Intel Corporation. "System Management Mode". In: *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Jan. 2019. Chap. 34 (cit. on pp. 14, 18, 116, 121, 122, 154).

[140] IT-ISAC. *FAQ*. URL: `https://www.it-isac.org/faq` (visited on 10/04/2019) (cit. on pp. 69, 77).

[141] Mirek Jahoda, Ioanna Gkioka, Robert Krátký, Martin Prpič, Tomáš Čapek, Stephen Wadeley, Yoana Ruseva, and Miroslav Svoboda. "System Auditing". In: *Red Hat Enterprise Linux 7 Security Guide*. 2017. Chap. 6, pp. 185–204 (cit. on pp. 81, 153).

[142] Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E. Porter, and Radu Sion. "SoK: Introspections on Trust and the Semantic Gap". In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. SP '14. IEEE Computer Society, 2014, pp. 605–620. ISBN: 978-1-4799-4686-0. DOI: `10.1109/SP.2014.45` (cit. on pp. 31, 105, 107).

[143] Daehee Jang, Hojoon Lee, Minsu Kim, Daehyeok Kim, Daegyeong Kim, and Brent Byunghoon Kang. "ATRA: Address Translation Redirection Attack against Hardware-based External Monitors". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, AZ, USA). ACM, 2014, pp. 167–178. DOI: `10.1145/2660267.2660303` (cit. on pp. 31, 34, 117, 154).

[144] Anas Abou El Kalam, Salem Benferhat, Alexandre Miège, Rania El Baida, Frédéric Cuppens, Claire Saurel, Philippe Balbiani, Yves Deswarte, and Gilles Trouessin. "Organization Based Access Control". In: *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*. POLICY '03. IEEE Computer Society, 2003 (cit. on p. 53).

[145] Corey Kallenberg, John Butterworth, Xeno Kovah, and C Cornwell. "Defeating Signed BIOS Enforcement". EkoParty, Buenos Aires. 2013 (cit. on p. 127).

[146] Corey Kallenberg and Rafal Wojtczuk. "Speed racer: Exploiting an Intel flash protection race condition". 31st Chaos Communication Congress (31C3). 2014 (cit. on p. 17).

[147] Jeffrey Katcher. *Postmark: A new file system benchmark*. Tech. rep. 3022. Network Appliance, 1997 (cit. on p. 91).

[148] Michael Kerrisk. "CAP_SYS_ADMIN: the new root". In: *LWN* (Mar. 14, 2012) (cit. on p. 22).

[149] Michael Kerrisk. "Namespaces in operation, part 1: namespaces overview". In: *LWN* (Jan. 4, 2013) (cit. on pp. 22, 78, 80).

[150] Amin Kharraz, William Robertson, Davide Balzarotti, Leyla Bilge, and Engin Kirda. "Cutting the Gordian Knot: A Look Under the Hood of Ransomware Attacks". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2015, pp. 3–24. DOI: `10.1007/978-3-319-20550-2_1` (cit. on pp. 63, 86).

[151] Nizar Kheir, Nora Cuppens-Boulahia, Frédéric Cuppens, and Hervé Debar. "A Service Dependency Model for Cost-sensitive Intrusion Response". In: *Proceedings of the 15th European Conference on Research in Computer Security* (Athens, Greece). ESORICS'10. 2010, pp. 626–642. DOI: `10.1007/978-3-642-15497-3_38` (cit. on pp. 50, 51, 65, 153).

[152] Nizar Kheir, Hervé Debar, Nora Cuppens-Boulahia, Frédéric Cuppens, and Jouni Viinikka. "Cost Evaluation for Intrusion Response Using Dependency Graphs". In: *International Conference on Network and Service Security*. June 2009. ISBN: 978-2-9532-4431-1 (cit. on p. 96).

[153] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. "Intrusion Recovery Using Selective Re-execution". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada). OSDI'10. USENIX Association, 2010, pp. 89–104 (cit. on pp. 43, 46, 47, 59, 97, 150).

[154] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors". In: *Proceeding of the 41st Annual International Symposium on Computer Architecture* (Minneapolis, Minnesota, USA). ISCA '14. IEEE Press, 2014, pp. 361–372. ISBN: 978-1-4799-4394-4. DOI: `10.1109/ISCA.2014.6853210` (cit. on p. 63).

[155] Ivan Kirillov, Desiree Beck, Penny Chase, and Robert Martin. "Malware Attribute Enumeration and Characterization". 2011 (cit. on pp. 63, 65, 77, 96, 153).

[156] John C. Knight. "Safety Critical Systems: Challenges and Directions". In: *Proceedings of the 24th International Conference on Software Engineering*. ICSE '02. Orlando, Florida: ACM, 2002, pp. 547–550. DOI: `10.1145/581339.581406` (cit. on p. 42).

[157] John C. Knight and Elisabeth A. Strunk. "Achieving Critical System Survivability Through Software Architectures". In: *Architecting Dependable Systems II*. Ed. by Rogério de Lemos, Cristina Gacek, and Alexander Romanovsky. 2004, pp. 51–78. ISBN: 978-3-540-25939-8 (cit. on pp. 5, 42).

[158] John C. Knight, Elisabeth A Strunk, and Kevin J. Sullivan. "Towards a rigorous definition of information system survivability". In: *Proceedings of the 3rd DARPA Information Survivability Conference and Exposition*. Vol. 1. IEEE. 2003, pp. 78–89. DOI: `10.1109/DISCEX.2003.1194874` (cit. on p. 29).

[159] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019. DOI: `10.1109/SP.2019.00002` (cit. on p. 63).

[160] Barbara Kordy, Sjouke Mauw, Saša Radomirović, and Patrick Schweitzer. "Attack–defense trees". In: *Journal of Logic and Computation* 24.1 (2014), pp. 55–87. DOI: `10.1093/logcom/exs029` (cit. on p. 50).

[161] Barbara Kordy, Ludovic Piètre-Cambacédès, and Patrick Schweitzer. "DAG-based attack and defense modeling: Don't miss the forest for the attack trees". In: *Computer Science Review* 13-14 (2014), pp. 1–38. DOI: `10.1016/j.cosrev.2014.07.001` (cit. on p. 50).

[162] Xeno Kovah and Corey Kallenberg. "How Many Million BIOSes Would you Like to Infect?" (Vancouver, B.C., Canada). CanSecWest. 2015 (cit. on p. 104).

[163] Jean-Claude Laprie. "From dependability to resilience". In: *38th IEEE/IFIP International Conference On Dependable Systems and Networks*. 2008 (cit. on p. 28).

[164] Michael Larabel and Matthew Tippett. *Phoronix Test Suite*. URL: `https://www.phoronix-test-suite.com/` (visited on 08/05/2019) (cit. on p. 90).

[165] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (San Jose, CA, USA). CGO '04. IEEE Computer Society, Mar. 2004, pp. 75–88 (cit. on pp. 123, 155).

[166] Chris Lattner, Andrew Lenharth, and Vikram Adve. "Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World". In: *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California). PLDI '07. June 2007. DOI: `10.1145/1250734.1250766` (cit. on p. 128).

[167] Hojoon Lee, Hyungon Moon, Daehee Jang, Kihwan Kim, Jihoon Lee, Yunheung Paek, and Brent ByungHoon Kang. "KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object". In: *Proceedings of the 22th USENIX Security Symposium* (Washington, D.C., USA). USENIX Association, 2013, pp. 511–526 (cit. on pp. 34, 37, 135).

[168] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. "LogGC: garbage collecting audit log". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer &#38; communications security* (Berlin, Germany). CCS '13. ACM, 2013, pp. 1005–1016. DOI: `10.1145/2508859.2516731` (cit. on p. 48).

[169] Yongje Lee, Ingoo Heo, Dongil Hwang, Kyungmin Kim, and Yunheung Paek. "Towards a Practical Solution to Detect Code Reuse Attacks on ARM Mobile Devices". In: *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy* (Portland, OR, USA). HASP '15. ACM, 2015, 3:1–3:8. DOI: `10.1145/2768566.2768569` (cit. on pp. 34, 39, 135).

[170] *Lenovo Security Advisory: LEN-4710*. Sept. 2016. URL: `https://support.lenovo.com/us/en/product_security/len_4710` (visited on 08/05/2019) (cit. on p. 128).

[171] *Lenovo Security Advisory: LEN-8324*. Nov. 2016. URL: `https://support.lenovo.com/us/en/solutions/len-8324` (visited on 08/05/2019) (cit. on p. 128).

[172] Kevin Lepak, Gerry Talbot, Sean White, Noah Beck, Sam Naffziger, et al. "The Next Generation AMD Enterprise Server Product Architecture". In: *IEEE Hot Chips*. 2017 (cit. on p. 121).

[173] Thomas Letan. "Specifying and Verifying Hardware-based Security Enforcement Mechanisms". PhD thesis. CentraleSupélec, Oct. 2018 (cit. on p. 19).

[174] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. "SBAP: Software-based Attestation for Peripherals". In: *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*. TRUST'10. 2010, pp. 16–29. DOI: `10.1007/978-3-642-13869-0_2` (cit. on p. 36).

[175] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. "VIPER: verifying the integrity of PERipherals' firmware". In: *Proceedings of the 18th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA). CCS '11. ACM, 2011, pp. 3–16. DOI: `10.1145/2046707.2046711` (cit. on p. 36).

[176] Philippe Lin. *Hacking Team Uses UEFI BIOS Rootkit to Keep RCS 9 Agent in Target Systems*. TrendLabs Security Intelligence Blog. July 13, 2013. URL: `https://blog.trendmicro.com/trendlabs-security-intelligence/hacking-team-uses-uefi-bios-rootkit-to-keep-rcs-9-agent-in-target-systems/` (visited on 08/05/2019) (cit. on pp. 6, 103).

[177] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium (USENIX Security 18)* (Baltimore, MD). USENIX Association, 2018, pp. 973–990. ISBN: 978-1-931971-46-1 (cit. on p. 63).

[178] Heiner Litz, Holger Froening, Mondrian Nuessle, and Ulrich Bruening. "VELO: A novel communication engine for ultra-low latency message transfers". In: *Proceedings of the 37th International Conference on Parallel Processing*. IEEE Computer Society, 2008, pp. 238–245. DOI: `10.1109/ICPP.2008.85` (cit. on pp. 121, 126).

[179] Heiner Litz, Maximilian Thuermer, and Ulrich Bruening. "TCCluster: A Cluster Architecture Utilizing the Processor Host Interface as a Network Interconnect". In: *Proceedings of the International Conference on Cluster Computing (CLUSTER)*. IEEE Computer Society, 2010, pp. 9–18. DOI: `10.1109/CLUSTER.2010.37` (cit. on p. 121).

[180] Ziyi Liu, JongHyuk Lee, Junyuan Zeng, Yuanfeng Wen, Zhiqiang Lin, and Weidong Shi. "CPU Transparent Protection of OS Kernel and Hypervisor Integrity with Programmable DRAM". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel). ISCA '13. ACM, 2013, pp. 392–403. ISBN: 978-1-4503-2079-5. DOI: `10.1145/2485922.2485956` (cit. on pp. 34, 41, 134, 135).

[181] Ge Livian. *BIOS Threat is Showing up Again!* Sept. 2011. URL: `https://www.symantec.com/connect/blogs/bios-threat-showing-again` (visited on 09/06/2019) (cit. on p. 6).

[182] Tony C.S. Lo. "Implementing A Bluetooth Stack on UEFI". UEFI PlugFest. Oct. 2014 (cit. on p. 12).

[183] David J. Lu. "Watchdog Processors and Structural Integrity Checking". In: *IEEE Transactions on Computers* 31.7 (July 1982), pp. 681–685. DOI: `10.1109/TC.1982.1676066` (cit. on p. 38).

[184] *LVFS: Linux Vendor Firmware Service*. URL: `https://fwupd.org/` (visited on 08/05/2019) (cit. on p. 104).

[185] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. "ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting". In: *Proceedings of the 2016 Annual Network and Distributed System Security Symposium* (San Diego, CA). NDSS '16. Feb. 2016. DOI: `10.14722/ndss.2016.23350` (cit. on p. 48).

[186] Tarjei Mandt, Mathew Solnik, and David Wang. "Demystifying the Secure Enclave Processor". Black Hat Las Vegas. 2016 (cit. on pp. 119, 120, 155).

[187] *mariadb*. URL: `https://mariadb.org/` (visited on 08/05/2019) (cit. on p. 84).

[188] Louis Marinos and Marco Lourenço. *ENISA Threat Landscape Report 2018*. Tech. rep. European Union Agency for Network and Information Security (ENISA), Jan. 2019. DOI: `10.2824/622757` (cit. on pp. 3, 149).

[189] R. Timothy Marler and Jasbir S. Arora. "Survey of multi-objective optimization methods for engineering". In: *Structural and Multidisciplinary Optimization* 26.6 (Apr. 2004), pp. 369–395. DOI: `10.1007/s00158-003-0368-6` (cit. on pp. 71, 72).

[190] Chris Mason. *Compilebench*. 2008. URL: `https://oss.oracle.com/~mason/compilebench/` (visited on 08/05/2019) (cit. on p. 90).

[191] Nicholas D. Matsakis and Felix S. Klock II. "The Rust Language". In: *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology* (Portland, OR, USA). HILT '14. ACM, 2014, pp. 103–104 (cit. on p. 120).

[192] Microsoft. *Windows UEFI firmware update platform*. Apr. 20, 2017. URL: `https://docs.microsoft.com/en-us/windows-hardware/drivers/bringup/windows-uefi-firmware-update-platform` (visited on 08/05/2019) (cit. on p. 104).

[193] Microsoft. *Job Objects*. MSDN. May 31, 2018. URL: `https://msdn.microsoft.com/en-us/library/windows/desktop/ms684161(v=vs.85).aspx` (visited on 08/05/2019) (cit. on pp. 23, 78).

[194] Microsoft. *Protect important folders with controlled folder access*. Nov. 29, 2018. URL: `https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-exploit-guard/controlled-folders-exploit-guard?ocid=cx-blog-mmpc` (visited on 08/05/2019) (cit. on pp. 86, 96).

[195] Microsoft. *Restricted Tokens*. MSDN. May 31, 2018. URL: `https://msdn.microsoft.com/en-us/library/windows/desktop/aa379316(v=vs.85).aspx` (visited on 08/05/2019) (cit. on pp. 23, 78).

[196] Microsoft. *Windows Integrity Mechanism Design*. MSDN. July 5, 2018. URL: `https://msdn.microsoft.com/en-us/library/bb625963.aspx` (visited on 08/05/2019) (cit. on pp. 23, 78).

[197] Microsoft. *Windows System Restore*. May 31, 2018. URL: `https://docs.microsoft.com/en-us/windows/desktop/sr/system-restore-portal` (visited on 08/05/2019) (cit. on pp. 45, 150).

[198] Microsoft. *Windows Defender Antivirus*. July 23, 2019. URL: `https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-antivirus/windows-defender-antivirus-in-windows-10` (visited on 08/05/2019) (cit. on p. 44).

[199] MITRE. *ATT&CK*. 2019. URL: `https://attack.mitre.org/` (visited on 05/05/2019) (cit. on p. 65).

[200] MITRE. *Encyclopedia of Malware Attributes*. URL: `https://collaborate.mitre.org/ema/` (visited on 08/05/2019) (cit. on pp. 63, 77, 96).

[201] MITRE. *Malware Capabilities*. URL: `https://github.com/MAECProject/schemas/wiki/Malware-Capabilities` (visited on 08/05/2019) (cit. on pp. 65, 77, 96, 153).

[202] Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent Byunghoon Kang. "Vigilare: Toward Snoop-based Kernel Integrity Monitor". In: *Proceedings of the ACM Conference on Computer and Communications Security* (Vienna, Austria). CCS '12. ACM, 2012, pp. 28–37. ISBN: 978-1-4503-1651-4. DOI: `10.1145/2382196.2382202` (cit. on pp. 33, 34, 36).

[203] Benjamin Morin and Ludovic Mé. "Intrusion detection and virology: an analysis of differences, similarities and complementariness". In: *Journal in Computer Virology* 3.1 (Apr. 1, 2007), pp. 39–49. DOI: `10.1007/s11416-007-0036-2` (cit. on pp. 5, 150).

[204] Alexander Motzek, Gustavo Gonzalez-Granadillo, Hervé Debar, Joaquin Garcia-Alfaro, and Ralf Möller. "Selection of Pareto-efficient response plans based on financial and operational assessments". In: *EURASIP Journal on Information Security* 2017-07.1 (2017), p. 12. DOI: `10.1186/s13635-017-0063-6` (cit. on pp. 52, 71).

[205] Steve Neville. *Web site offline? New server-focused FAIRWARE Ransomware could be why*. Sept. 2016. URL: `https://blog.trendmicro.com/web-site-offline-new-server-focused-fairware-ransomware/` (visited on 08/05/2019) (cit. on p. 63).

[206] *nginx*. URL: `https://nginx.org/` (visited on 08/05/2019) (cit. on p. 84).

[207] Ruchna Nigam. *Unit 42 Finds New Mirai and Gafgyt IoT/Linux Botnet Campaigns*. July 2018. URL: `https://researchcenter.paloaltonetworks.com/2018/07/unit42-finds-new-mirai-gafgyt-iotlinux-botnet-campaigns/` (visited on 08/05/2019) (cit. on pp. 84, 96).

[208] Ben Niu and Gang Tan. "Modular control-flow integrity". In: *ACM SIGPLAN Notices* 49.6 (2014), pp. 577–587. DOI: `10.1145/2594291.2594295` (cit. on pp. 39, 40, 111).

[209] Katherine Noyes. "Docker: A 'Shipping Container' for Linux Code". In: *Linux.com* (Aug. 1, 2013) (cit. on p. 23).

[210] NSA and Red Hat. *SELinux*. URL: `https://selinuxproject.org/` (visited on 08/05/2019) (cit. on pp. 22, 76).

[211] Dmytro Oleksiuk. *Exploiting AMI Aptio firmware on example of Intel NUC*. Oct. 2016. URL: `http://blog.cr4.sh/2016/10/exploiting-ami-aptio-firmware.html` (visited on 08/05/2019) (cit. on pp. 17, 18, 103, 127).

[212] Dmytro Oleksiuk. *Exploring and exploiting Lenovo firmware secrets*. June 2016. URL: `http://blog.cr4.sh/2016/06/exploring-and-exploiting-lenovo.html` (visited on 08/05/2019) (cit. on pp. 17, 18, 103, 113, 114, 128).

[213] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. "MulVAL: A Logic-based Network Security Analyzer". In: *Proceedings of the 14th USENIX Security Symposium*. SEC'05. Baltimore, MD: USENIX Association, 2005 (cit. on p. 50).

[214] Roberto Paleari, Lorenzo Martignoni, Emanuele Passerini, Drew Davidson, Matt Fredrikson, Jonathon T. Giffin, and Somesh Jha. "Automatic Generation of Remediation Procedures for Malware Infections". In: *USENIX Security Symposium*. 2010 (cit. on p. 44).

[215] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. "A Fistful of Red-pills: How to Automatically Generate Procedures to Detect CPU Emulators". In: *Proceedings of the 3rd USENIX Conference on Offensive Technologies* (Montreal, Canada). WOOT'09. USENIX Association, 2009 (cit. on p. 83).

[216] PaX Team. "RAP: RIP ROP". H2HC. Oct. 2015 (cit. on pp. 39, 40, 111, 134).

[217] Vern Paxson. "Bro: a System for Detecting Network Intruders in Real-Time". In: *Computer Networks* 31.23-24 (1999), pp. 2435–2463 (cit. on pp. 5, 150).

[218] Jenny M. Pelner and James A. Pelner. *Minimal Intel Architecture Boot Loader*. Tech. rep. Intel, Jan. 2010 (cit. on pp. 9, 10).

[219] Nick L. Petroni Jr, Timothy Fraser, AAron Walters, and William A. Arbaugh. "An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data". In: *Proceedings of the 15th USENIX Security Symposium* (Vancouver, B.C., Canada). USENIX Association, 2006 (cit. on pp. 32, 37).

[220] Nick L. Petroni Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. "Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor". In: *Proceedings of the 13th USENIX Security Symposium* (San Diego, CA, USA). USENIX Association, Aug. 2004, pp. 179–194 (cit. on pp. 32, 35, 37, 135).

[221] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. "The Use of Name Spaces in Plan 9". In: *SIGOPS Operating Systems Review* 27.2 (Apr. 1993), pp. 72–76. DOI: `10.1145/155848.155861` (cit. on p. 22).

[222] Manish Prasad and Tzi-cker Chiueh. "A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks." In: *Proceedings of USENIX Annual Technical Conference*. June 2003, pp. 211–224 (cit. on p. 116).

[223] Bruno Pujos. *SMM unchecked pointer vulnerability*. May 2016. URL: `http://esec-lab.sogeti.com/posts/2016/05/30/smm-unchecked-pointer-vulnerability.html` (visited on 08/05/2019) (cit. on pp. 17, 18, 103, 128, 154).

[224] Ganesan Ramalingam. "The undecidability of aliasing". In: *ACM Trans. Program. Lang. Syst.* 16.5 (Sept. 1994), pp. 1467–1471. DOI: `10.1145/186025.186041` (cit. on p. 40).

[225] Andrew R. Regenscheid. *Platform Firmware Resiliency Guidelines*. Tech. rep. Special Publication 800-193. National Institute of Standards and Technology, Apr. 2018. DOI: `10.6028/NIST.SP.800-193` (cit. on pp. 12, 109, 151).

[226] ESET Researchers. *LoJax: First UEFI rootkit found in the wild, courtesy of the Sednit group*. Tech. rep. ESET, Sept. 2018 (cit. on pp. 6, 7, 17, 103, 151).

[227] Ohad Rodeh, Josef Bacik, and Chris Mason. "BTRFS: The Linux B-tree Filesystem". In: *ACM Transactions on Storage (TOS)* 9.3 (2013), p. 9. DOI: `10.1145/2501620.2501623` (cit. on p. 79).

[228] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. "Return-oriented programming: Systems, languages, and applications". In: *ACM Transactions on Information and System Security (TISSEC)* 15.1 (2012), p. 2. DOI: `10.1145/2133375.2133377` (cit. on p. 38).

[229] Ron Ross, Richard Graubart, Deborah Bodeau, and Rosalie McQuaid. *Cyber Resiliency Considerations for the Engineering of Trustworthy Secure Systems*. Tech. rep. Special Publication 800-160, Volume 2. National Institute of Standards and Technology, Mar. 2018 (cit. on p. 28).

[230] Xiaoyu Ruan. "Boot with Integrity, or Don't Boot". In: *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. Apress, Aug. 2014. Chap. 6, pp. 143–163. ISBN: 978-1-4302-6572-6. DOI: `10.1007/978-1-4302-6572-6_6` (cit. on pp. 13, 62, 109, 151).

[231] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Part 2*. 6th ed. Microsoft Press, 2012. ISBN: 978-0735665873 (cit. on pp. 21, 77).

[232] Joanna Rutkowska. "Beyond the CPU: Defeating hardware based RAM acquisition". Black Hat DC. Feb. 2007 (cit. on p. 33).

[233] Jerome H. Saltzer and Michael D. Schroeder. "The protection of information in computer systems". In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308. DOI: `10.1109/PROC.1975.9939` (cit. on p. 21).

[234] Fred B. Schneider. "Enforceable Security Policies". In: *ACM Trans. Inf. Syst. Secur.* 3.1 (Feb. 2000), pp. 30–50. DOI: `10.1145/353323.353382` (cit. on p. 30).

[235] Mark Seaborn and Thomas Dullien. *Exploiting the DRAM rowhammer bug to gain kernel privileges*. Mar. 2015. URL: `https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html` (visited on 08/05/2019) (cit. on p. 63).

[236] Security Working Group. *Draft Standard for Information Technology–POSIX–Part 1*. Withdrawn draft. IEEE, Oct. 1997 (cit. on p. 22).

[237] Tomás Senart. *Vegeta*. URL: `https://github.com/tsenart/vegeta` (visited on 08/05/2019) (cit. on p. 88).

[238] Arvind Seshadri, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. "SWATT: Software-based attestation for embedded devices". In: *IEEE Symposium on Security and Privacy*. IEEE. 2004, pp. 272–282. DOI: `10.1109/SECPRI.2004.1301329` (cit. on p. 36).

[239] Alireza Shameli-Sendi, Mohamed Cheriet, and Abdelwahab Hamou-Lhadj. "Taxonomy of Intrusion Risk Assessment and Response System". In: *Computers & Security* 45 (Sept. 2014), pp. 1–16. DOI: `10.1016/j.cose.2014.04.009` (cit. on pp. 5, 42, 48–50, 53, 65, 99, 150, 153).

[240] Alireza Shameli-Sendi, Naser Ezzati-Jivan, Masoume Jabbarifar, and Michel Dagenais. "Intrusion Response Systems: Survey and Taxonomy". In: *Int. J. Comput. Sci. Netw. Secur* 12.1 (2012), pp. 1–14 (cit. on pp. 48, 98).

[241] Alireza Shameli-Sendi, Habib Louafi, Wenbo He, and Mohamed Cheriet. "Dynamic Optimal Countermeasure Selection for Intrusion Response System". In: *IEEE Transactions on Dependable and Secure Computing* 15.5 (2018), pp. 755–770. DOI: `10.1109/TDSC.2016.2615622` (cit. on pp. 50, 51, 53, 59, 65, 71, 72, 96, 97, 150, 153).

[242] Zhiyong Shan, Xin Wang, and Tzi-cker Chiueh. "Malware Clearance for Secure Commitment of OS-Level Virtual Machines". In: *IEEE Trans. Dependable Secur. Comput.* 10.2 (Mar. 2013), pp. 70–83. DOI: `10.1109/TDSC.2012.88` (cit. on pp. 43, 45, 46, 97).

[243] Vivek Shandilya, Chris B. Simmons, and Sajjan Shiva. "Use of Attack Graphs in Security Systems". In: *Journal of Computer Networks and Communications* 2014 (2014). DOI: `10.1155/2014/818957` (cit. on p. 50).

[244] Benjamin H. Shelton. "Popcorn Linux: enabling efficient inter-core communication in a Linux-based multikernel operating system". MA thesis. Virginia Polytechnic Institute and State University, May 2013 (cit. on pp. 120, 155).

[245] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M. Wing. "Automated Generation and Analysis of Attack Graphs". In: *Proceedings of the 2002 IEEE Symposium on Security and Privacy*. SP '02. IEEE Computer Society, 2002. ISBN: 0-7695-1543-6 (cit. on p. 50).

[246] Robert W. Shirey. *Internet Security Glossary, Version 2*. RFC 4949. Internet Engineering Task Force, Aug. 2007, pp. 1–365 (cit. on pp. 25, 26).

[247] *snapper*. 2018. URL: `http://snapper.io/` (visited on 08/05/2019) (cit. on pp. 79, 153).

[248] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. "Enforcing Kernel Security Invariants with Data Flow Integrity". In: *Proceedings of the 2016 Annual Network and Distributed System Security Symposium* (San Diego, CA). NDSS '16. Feb. 2016. DOI: `10.14722/ndss.2016.23218` (cit. on p. 62).

[249] WonJun Song, John Kim, Jae-Wook Lee, and Dennis Abts. "Security Vulnerability in Processor-Interconnect Router Design". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, AZ, USA). CCS '14. ACM, 2014, pp. 358–368. DOI: `10.1145/2660267.2660290` (cit. on p. 122).

[250] Brad Spengler. *In reply to: Kernel vulnerabilities: old or new?* Comment posted on a LWN article. Oct. 20, 2010. URL: `https://lwn.net/Articles/410674/` (visited on 08/05/2019) (cit. on p. 43).

[251] Brad Spengler. *False Boundaries and Arbitrary Code Execution*. Jan. 2, 2011. URL: `https://forums.grsecurity.net/viewtopic.php?f=7&t=2522` (visited on 08/05/2019) (cit. on p. 22).

[252] Natalia Stakhanova, Samik Basu, and Johnny Wong. "A Taxonomy of Intrusion Response Systems". In: *Int. J. Inf. Comput. Secur.* 1 (Jan. 2007), pp. 169–184. DOI: `10.1504/IJICS.2007.012248` (cit. on pp. 49, 99).

[253] *systemd System and Service Manager*. 2019. URL: `https://www.freedesktop.org/wiki/Software/systemd/` (visited on 08/05/2019) (cit. on pp. 21, 77, 78, 153).

[254] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal War in Memory". In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. S&P '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 48–62. DOI: `10.1109/SP.2013.13` (cit. on p. 23).

[255] Kacper Szurek. *Gitea 1.4.0 Unauthenticated Remote Code Execution*. 2018. URL: `https://security.szurek.pl/gitea-1-4-0-unauthenticated-rce.html` (visited on 08/05/2019) (cit. on p. 85).

[256] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. "Can we make operating systems reliable and secure?" In: *Computer* 39.5 (May 2006), pp. 44–51. DOI: `10.1109/MC.2006.156` (cit. on p. 20).

[257] The PaX Team. *Homepage of PaX*. URL: `https://pax.grsecurity.net/` (visited on 08/05/2019) (cit. on p. 43).

[258] United States Computer Emergency Readiness Team. *Automated Indicator Sharing (AIS)*. URL: `https://www.us-cert.gov/sites/default/files/ais_files/AIS_fact_sheet.pdf` (visited on 10/04/2019) (cit. on pp. 69, 77).

[259] The coreboot community. *coreboot*. 2017. URL: `https://www.coreboot.org/` (visited on 08/05/2019) (cit. on p. 125).

[260] *The Linux Audit Project*. URL: `https://github.com/linux-audit/` (visited on 08/05/2019) (cit. on p. 81).

[261] Michael E. Thomadakis. *The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms*. Tech. rep. Texas A&M University, Mar. 2011 (cit. on p. 9).

[262] Yohann Thomas. "Policy-Based Response to Intrusions Through Context Activation". PhD thesis. Télécom Bretagne, 2007 (cit. on pp. 53, 55).

[263] Tianocore. *EDK II*. 2017. URL: `http://www.tianocore.org/edk2/` (visited on 08/05/2019) (cit. on pp. 123, 125).

[264] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. "Enforcing forward-edge control-flow integrity in gcc & llvm". In: *Proceedings of the 23rd USENIX Security Symposium* (San Diego, CA, USA). USENIX Association, Aug. 2014, pp. 941–955 (cit. on pp. 30, 31, 39, 40, 111).

[265] Craig Timberg. "The kernel of the argument over Linux's vulnerabilities". In: *The Washington Post* (Nov. 5, 2015) (cit. on p. 43).

[266] Thomas Toth and Christopher Kruegel. "Evaluating the Impact of Automated Intrusion Response Mechanisms". In: *Proceedings of the 18th Annual Computer Security Applications Conference*. ACSAC '02. IEEE Computer Society, 2002. DOI: `10.1109/CSAC.2002.1176302` (cit. on p. 96).

[267] Trend Micro Cyber Safety Solutions Team. *Cryptocurrency Miner Distributed via PHP Weathermap Vulnerability, Targets Linux Servers*. Mar. 2018. URL: `https://blog.trendmicro.com/trendlabs-security-intelligence/cryptocurrency-miner-distributed-via-php-weathermap-vulnerability-targets-linux-servers/` (visited on 08/05/2019) (cit. on pp. 84, 96).

[268] Trusted Computing Group. *TPM Main, Part 1 Design Principles*. Trusted Computing Group. Mar. 2011 (cit. on pp. 13, 109, 151).

[269] Trusted Computing Group. *TCG Storage Security Subsystem Class: Opal*. Trusted Computing Group. Aug. 2015 (cit. on p. 127).

[270] Jim Turley. *Introduction to Intel Architecture*. Tech. rep. Intel, Jan. 2014 (cit. on p. 10).

[271] UEFI Forum. *UEFI Platform Initialization Specification*. Version 1.7. Jan. 2019 (cit. on p. 11).

[272] UEFI Forum. *Unified Extensible Firmware Interface Specification*. Version 2.8. Mar. 2019 (cit. on pp. 11, 13, 44, 62, 104, 151).

[273] Martin Untersinger. "Le piratage de TV5 Monde vu de l'intérieur". In: *Le Monde* (June 10, 2017) (cit. on pp. 6, 151).

[274] Yoann Vandoorselaere. *Prelude SIEM*. URL: `https://www.prelude-siem.org/` (visited on 08/05/2019) (cit. on p. 35).

[275] Sven Vermeulen. "Handling SELinux-aware Applications". In: *SELinux Cookbook*. Packt Publishing, Sept. 2014. Chap. 10. ISBN: 9781783989669 (cit. on p. 77).

[276] Zhi Wang and Xuxian Jiang. "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity". In: *Proceedings of the 31st IEEE Symposium on Security and Privacy* (Oakland, CA, USA). SP '10. IEEE Computer Society, May 2010, pp. 380–395. DOI: `10.1109/SP.2010.30` (cit. on pp. 31, 39).

[277] Ashton Webster, Ryan Eckenrod, and James Purtilo. "Fast and Service-preserving Recovery from Malware Infections Using CRIU". In: *Proceedings of the 27th USENIX Security Symposium* (Baltimore, MD). USENIX Association, 2018, pp. 1199–1211 (cit. on pp. 43, 45, 54, 59, 89, 97, 99, 150).

[278] Dong Wei. "Goodbye PXE, Hello HTTP Boot". LinuxCon Europe, UEFI Mini-Summit. Oct. 2015 (cit. on p. 12).

[279] Evan Wheeler. "Risky Business". In: *Security risk management: Building an information security risk management program from the Ground Up*. 1st. Syngress Publishing, May 2011. Chap. 2, pp. 37–40. ISBN: 978-1597496155 (cit. on pp. 52, 65).

[280] Ric Wheeler. *fs-mark*. URL: `https://sourceforge.net/projects/fsmark/` (visited on 08/05/2019) (cit. on p. 90).

[281] Dick Wilkins. "UEFI Firmware – Securing SMM". UEFI PlugFest. May 2015 (cit. on pp. 109, 117).

[282] Rafal Wojtczuk and Corey Kallenberg. "Attacking UEFI boot script". 31st Chaos Communication Congress (31C3). 2014 (cit. on pp. 17, 103).

[283] Rafal Wojtczuk and Joanna Rutkowska. "Attacking SMM memory via Intel CPU cache poisoning". Invisible Things Lab. Mar. 2009 (cit. on pp. 16–18, 103, 109).

[284] Rafal Wojtczuk and Alexander Tereshkin. "Attacking Intel BIOS". Black Hat USA. July 2009 (cit. on pp. 17, 18, 103).

[285] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. "CFIMon: Detecting Violation of Control Flow Integrity Using Performance Counters". In: *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (Washington, D.C., USA). DSN '12. IEEE Computer Society, 2012, pp. 1–12. ISBN: 978-1-4673-1624-8. DOI: `10.1109/DSN.2012.6263958` (cit. on p. 38).

[286] Xi Xiong, Xiaoqi Jia, and Peng Liu. "SHELF: Preserving Business Continuity and Availability in an Intrusion Recovery System". In: *Proceedings of the 25th Annual Computer Security Applications Conference*. ACSAC '09. IEEE Computer Society, 2009, pp. 484–493. ISBN: 978-0-7695-3919-5. DOI: `10.1109/ACSAC.2009.52` (cit. on pp. 6, 43, 46, 47, 54, 59, 89, 93, 97, 150).

[287] Jiewen Yao. *[edk2] [PATCH V3 0/6] Enable SMM page level protection*. Patch posted to the EDK2 development mailing list. Nov. 11, 2016. URL: `https://www.mail-archive.com/edk2-devel@lists.01.org/msg19551.html` (visited on 08/05/2019) (cit. on p. 36).

[288] Jiewen Yao and Vincent J. Zimmer. *A Tour Beyond BIOS Supporting an SMM Resource Monitor using the EFI Developer Kit II*. Tech. rep. Intel, June 2015 (cit. on pp. 31, 62, 117, 142).

[289] Jiewen Yao and Vincent J. Zimmer. *A Tour Beyond BIOS - Memory Protection in UEFI BIOS*. Tech. rep. Intel, Mar. 2017 (cit. on pp. 7, 36, 109).

[290] Jiewen Yao, Vincent J. Zimmer, and Matt Flemming. *A Tour Beyond BIOS Memory Practices in UEFI*. Tech. rep. Intel, June 2015 (cit. on pp. 62, 109, 117).

[291] Jiewen Yao, Vincent Zimmer, and Star Zeng. *A Tour Beyond BIOS Implementing UEFI Authenticated Variables in SMM with EDKII*. Tech. rep. Intel, Sept. 2014 (cit. on pp. 14, 127, 154).

[292] S. S. Yau and Fu-Chung Chen. "An Approach to Concurrent Control Flow Checking". In: *IEEE Transactions on Software Engineering* SE-6.2 (Mar. 1980), pp. 126–137. DOI: `10.1109/TSE.1980.234478` (cit. on p. 38).

[293] Yang Yu, Fanglu Guo, Susanta Nanda, Lap-chung Lam, and Tzi-cker Chiueh. "A Feather-weight Virtual Machine for Windows Applications". In: *Proceedings of the 2nd International Conference on Virtual Execution Environments*. ACM. 2006, pp. 24–34 (cit. on p. 45).

[294] Eric Yuan, Naeem Esfahani, and Sam Malek. "A Systematic Survey of Self-Protecting Software Systems". In: *ACM Transactions on Autonomous and Adaptive Systems* 8.4 (Jan. 2014), 17:1–17:41. DOI: `10.1145/2555611` (cit. on pp. 5, 42, 49).

[295] Jim Yuill, Mike Zappe, Dorothy Denning, and Fred Feer. "Honeyfiles: deceptive files for intrusion detection". In: *Proceedings of the IEEE Workshop on Information Assurance*. IEEE. June 2004, pp. 116–122. DOI: `10.1109/IAW.2004.1437806` (cit. on p. 64).

[296] Mingwei Zhang and R. Sekar. "Control Flow Integrity for COTS Binaries". In: *Proceedings of the 22th USENIX Security Symposium* (Washington, D.C., USA). USENIX Association, Aug. 2013, pp. 337–352 (cit. on pp. 38–40).

[297] Xiaolan Zhang, Leendert van Doorn, Trent Jaeger, Ronald Perez, and Reiner Sailer. "Secure coprocessor-based intrusion detection". In: *Proceedings of the 10th workshop on ACM SIGOPS European workshop* (Saint-Émilion, France). EW 10. ACM, 2002, pp. 239–242. DOI: `10.1145/1133373.1133423` (cit. on p. 32).

[298] Vincent Zimmer, Michael Rothman, and Suresh Marisetty. *Beyond BIOS: Developing with the Unified Extensible Firmware Interface*. 2nd ed. Intel Press, 2010. ISBN: 978-1-934053-29-4 (cit. on pp. 11, 13).

COPYRIGHT PERMISSIONS

Figure 1.1 is derived from the work of Gorak tek-en under CC BY-SA 4.0.

COLOPHON

This document—except the first and last page—was typeset in LaTeX 2_ε using a modified version of the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". The font for the main body of the text is TeX Gyre Pagella 11pt. The graphics were drawn using `pgfplots` and `tikz`, while some of them were generated with `matplotlib` and `seaborn`. The bibliography was typeset using `biblatex` and `biber`.

UNIVERSITE
BRETAGNE
LOIRE / MATHSTIC

CentraleSupélec

Titre : Détecter et survivre aux intrusions : exploration de nouvelles approches de détection, de restauration, et de réponse aux intrusions

Mots clés : sécurité, détection d’intrusion, réponse aux intrusions, survivabilité

Résumé : Les systèmes informatiques, tels que les ordinateurs portables ou les systèmes embarqués, sont construits avec des couches de mécanismes de sécurité préventifs afin de réduire la probabilité qu’un attaquant les compromettent. Néanmoins, malgré des décennies d’avancées dans ce domaine, des intrusions surviennent toujours. Par conséquent, nous devons supposer que des intrusions auront lieu et nous devons construire nos systèmes afin qu’ils puissent les détecter et y survivre.

Les systèmes d’exploitation généralistes sont déployés avec des mécanismes de détection d’intrusion, mais leur capacité à survivre à une intrusion est limitée. Les solutions de l’état de l’art nécessitent des procédures manuelles, comportent des pertes de disponibilité, ou font subir un fort coût en performance. De plus, les composants de bas niveau tels que le BIOS sont de plus en plus la cible d’attaquants cherchant à implanter des logiciels malveillants, furtifs, et résilients. Bien que des solutions de l’état de l’art garantissent l’intégrité de ces composants au démarrage, peu s’intéressent à la sécurité des services fournis par le BIOS qui sont exécutés au sein du System Management Mode (SMM).

Ce manuscrit montre que nous pouvons construire des systèmes capables de détecter des intrusions au niveau du BIOS et y survivre au niveau du système d’exploitation. Tout d’abord, nous démontrons qu'une approche de survivabilité aux intrusions est viable et praticable pour des systèmes d’exploitation généralistes. Ensuite, nous démontrons qu'il est possible de détecter des intrusions au niveau du BIOS avec une solution basée sur du matériel.

Title: Detecting and Surviving Intrusions: Exploring New Intrusion Detection, Recovery, and Response Approaches

Keywords: security, intrusion detection, intrusion response, intrusion recovery, survivability

Abstract: Computing platforms, such as embedded systems or laptops, are built with layers of preventive security mechanisms to reduce the likelihood of attackers successfully compromising them. Nevertheless, given time and despite decades of improvements in preventive security, intrusions still happen. Therefore, systems should expect intrusions to occur, thus they should be built to detect and to survive them.

Commodity Operating Systems (OSs) are deployed with intrusion detection solutions, but their ability to survive them is limited. State-of-the-art approaches from industry or academia either involve manual procedures, loss of availability, coarse-grained responses, or non-negligible performance overhead. Moreover, low-level components, such as the BIOS, are increasingly targeted by sophisticated attackers to implant stealthy and resilient malware. State-of-the-art solutions, however, mainly focus on boot time integrity, leaving the runtime part of the BIOS—known as the System Management Mode (SMM)—a prime target.

This dissertation shows that we can build platforms that detect intrusions at the BIOS level and survive intrusions at the OS level. First, by demonstrating that intrusion survivability is a viable approach for commodity OSs. We develop a new approach that address various limitations from the literature, and we evaluate its security and performance. Second, by developing a hardware-based approach that detects attacks at the BIOS level where we demonstrate its feasibility with multiple detection methods.