Security at the Hardware/Software Interface

Guillaume Hiet

Université de Rennes 1
Spécialité Informatique

Security at the Hardware/Software Interface

Habilitation à Diriger des Recherches

Guillaume Hiet

December 15, 2021

Contents

Acronyms

AXI	Advanced eXtensible Interface. 31, 34
BIOS	Basic Input/Output System. 13, 14, 18, 20, 39, 53, 57, 59, 65, 66, 70–73
BRAM	Block RAM. 34
C&C	Command and Control. 43, 49, 50
CDI	Control-Data Isolation. 52
CFI	Control-Flow Integrity. 12, 16, 19–21, 35, 74
COW	Copy-On-Write. 48
CPU	Central Processing Unit. 11, 13–15, 18, 19, 21, 23–26, 28, 34, 36, 49, 54–56, 59, 60, 64–67, 69, 71, 74
CSME	Converged Security and Management Engine. 11
CTI	Cyber Threat Intelligence. 41, 42, 46
DBI	Dynamic Binary Instrumentation. 52, 53, 77
DDoS	Distributed Denial-of-Service. 49
DFI	Data-Flow Integrity. 35
DIFT	Dynamic Information Flow Tracking. 12, 22–28, 32–36, 74, 77
DMA	Direct Memory Access. 55, 60
DRAM	Dynamic Random Access Memory. 18, 29, 54–56, 66–69, 71
DRM	Digital Rights Management. 11
DSL	Domain Specific Language. 76
eBPF	extended Berkeley Packet Filter. 76
EMIO	Extended Multiplexed Input/Output. 28

FIFO First In First Out. 15, 20, 21, 26, 28, 31, 33

FP Frame Pointer. 32

FPGA Field-Programmable Gate Array. 10, 25–27, 31, 34–36, 58, 74

GPIO General Purpose Input/Output. 55

GPU Graphics Processing Unit. 36, 54

HDL Hardware Description Language. 36, 58, 74

HIDS Host-based Intrusion Detection System. 10

HLS High Level Synthesis. 36

HSE Hardware-based Security Enforcement. 57, 59–66, 70–74

HTTP Hypertext Transfer Protocol. 50, 51

IDMEF Intrusion Detection Message Exchange Format. 41

IDS Intrusion Detection System. 10, 37, 40, 41, 43, 46, 76

IR Intermediate Representation. 19

ISA Instruction Set Architecture. 10

JIT just-in-time. 76, 77

LSM Linux Security Module. 32

LTS Labeled Transition System. 59

MAC Mandatory Access Control. 40

MAEC Malware Attribute Enumeration and Characterization. 41, 43, 45

MMU Memory Management Unit. 56

MOO Multi-Objective Optimization. 46, 47

MSSP Managed Security Service Provider. 37

MTRR Memory Type Range Registers. 56

OS Operating System. 9, 11–13, 22, 24, 26, 29, 32, 38, 39, 41, 48, 53, 56, 57, 60, 64–67, 70–72, 76

P2P Peer-to-peer. 49

PAT Page Attribute Table. 56
PC Program Counter. 28, 29, 31
PCH Platform Controller Hub. 54, 56, 73
PCI Peripheral Component Interconnect. 56
PFT Program Flow Trace. 28, 33
PID Process IDentifier. 28
PL Programmable Logic. 25, 28
PS Processor System. 25
PSP Platform Security Processor. 11, 14
PTM Program Trace Macrocell. 26–29, 35

QPI QuickPath Interconnect. 15, 19

ROM Read-Only Memory. 55
ROP Return-Oriented Programming. 16

SEP Secure Enclave Processor. 11
SGX Software Guard Extensions. 11
SIEM Security Information and Event Manager. 37
SMBASE SMRAM Base address. 13, 18, 20, 21, 67, 71, 72
SMI System Management Interrupt. 13, 18–21, 69, 70
SMM System Management Mode. 13–22, 53, 57, 61, 65–72
SMRAM System Management RAM. 13, 16, 18–20, 65, 66, 68, 69, 71, 72
SMRAMC System Management RAM Control. 66, 68, 69, 71, 72
SMRR System Management Range Register. 66–69, 71, 72
SOAR Security Orchestration, Automation, and Response. 37
SOC Security Operations Center. 37
SoC System on a Chip. 25, 28, 33
SOO Single-Objective Optimization. 47
SP Stack Pointer. 32
SSD Solid-State Drive. 54
STIX Structured Threat Information eXpression. 41, 46

TCB Trusted Computing Base. 29, 61

TEE Trusted Execution Environment. 11

TLB Translation Lookaside Buffer. 56

TMC Tag Management Core. 27, 34

TOLUD Top of Low Usable DRAM. 56

TPIU Trace Port Interface Unit. 28

TPM Trusted Platform Module. 11, 13

TXT Trusted Execution Technology. 11, 57

UEFI Unified Extensible Firmware Interface. 7, 13, 14, 39, 53, 57

UIO Userspace I/O. 31

VM Virtual Machine. 57

WCET Worst Case Execution Time. 35

Chapter 1

Introduction

This document gives a synthetic overview of my research activities in the CIDRE team of the IRISA laboratory since October 2010. After completing my Ph.D. in 2008 on policy-based intrusion detection, I worked for Amossys as a security expert. In October 2010, I joined CentraleSupélec as an Assistant Professor. Since then, I have pursued my research works on cybersecurity.

In the last few years, my research has been mainly focused on Host-based Intrusion Detection Systems. I became more and more interested in the security of software close to hardware, such as operating systems or UEFI (Unified Extensible Firmware Interface) firmware, and I proposed detection approaches specifically targeting these systems. This evolution led me to be interested in the security of interfaces between software and hardware in a more general way. I was in secondment at Inria from 2018 to 2019 (partial time) and then from 2019 to 2020 (full time) to develop my research project in that domain.

1.1 Context

Software security is still one of the main concerns of today's systems, although much research has been done on the subject. Indeed, many vulnerabilities still affect computer systems, which are increasingly targeted by a wide range of sophisticated attacks. These attacks are strongly connected to the underground economy and military/intelligence activities. They can combine different malicious behaviors, exploiting vulnerabilities at different levels, e.g., hijacking the control flow of a program by exploiting a buffer overflow or accessing some restricted part of the file system by exploiting a directory traversal.

Cybersecurity consists of ensuring the fundamental properties of data confidentiality (only legitimate users can read information), data integrity (only legitimate users can modify information), and data availability (legitimate access to data cannot be prevented or disturbed). These properties are defined by a security policy that some protection mechanisms must enforce (e.g., authentication and access control, cryptography, anti-virus, etc.).

The best strategy to secure embedded systems would be to avoid vulnerabilities. Indeed, many preventive approaches have been proposed, such as static analysis of software code, dynamic verification enforced by the runtime environment (e.g., the Android Virtual Machine), or the use of cryptographic mechanisms. In practice, however:

- Preventive approaches are not systematically used (e.g., a lot of Android applications are still using C code, which is not protected by the Android Virtual Machine runtime verification);
- These approaches are insufficient to prevent all attacks (e.g., using Java or OCaml does not prevent all the logical errors that could lead to potential vulnerabilities).

It is thus also essential to monitor systems to **detect intrusions at runtime**. However, detecting attacks or intrusions is just the first step of reactive security. Different **reactions** must be implemented, such as notifying security incidents to administrators, stopping or modifying the execution of systems under attack, or putting the computer system in quarantine. Moreover, the addition of new security mechanisms raises the question of the confidence we can have in these mechanisms. In this context, **formal specification and proof** can enhance the trust in such security mechanisms.

In my research, I mainly focus on the security of hardware platforms. Such platforms consist of all the hardware and the low-level software components (operating system, hypervisor, firmware) that enable the execution of user applications. Even if it is essential to implement security mechanisms at the network level to protect the data that passes through it and the network itself, it is also fundamental to secure the platforms that ultimately allow executing applications and storing data. Moreover, these platforms are not always connected to a network.

In this work, we only consider software attacks in which the attacker provides the system with data or malicious code to exploit a vulnerability present on a hardware platform. This type of attack differs from physical attacks (e.g., fault injection or side-channel attacks) that require externally measuring or influencing a physical quantity (e.g., current or electromagnetic field). In contrast to the latter, software attacks can be executed at a significant distance from the targeted device, even if they exploit vulnerabilities in low-level components. This type of attack requires paying attention to the interactions between software and hardware.

The evolution of my research field led me to take a more general interest in the security of interfaces between software and hardware. In this perspective, I proposed, with my colleagues Clémentine Maurice (CNRS, IRISA EMSEC), Frédéric Tronel (CentraleSupélec, IRISA CIDRE), Jean-Louis Lanet (Inria, IRISA CIDRE) and Ronan Lashermes (Inria), a project for a Thematic Semester on the Security of Software/Hardware Interfaces (SILM), which has been selected by the partners of the PEC (Pôle d'Excellence Cyber) research centre. Such a semester was funded by the DGA and managed by Inria.

As the chair-holder of the SILM thematic semester[1], I organized various events and invited many researchers from academia, industry, and governmental organizations working in this field. I also wrote a white paper for the DGA (the French Defence Procurement Agency), outlining the state of the art of the domain and the strategic axes to be developed, both from a scientific and industrial point of view. During this semester, I was in partial secondment (50%) at Inria from October 2019 to October 2020.

[1] `https://silm.inria.fr/`

1.2 Challenges

The research I detail in this manuscript focuses on computer security at the software/hardware interface, focusing on intrusion detection and reaction. More precisely, I am interested in finding solutions to respond to the following challenges:

1. How to leverage hardware features to isolate HIDS and bridge the semantic gap resulting from the isolation?
2. How to automatically react to intrusions at the host level?
3. Using formal methods, how to specify and verify security mechanisms that require interactions between software and hardware?

1.3 Other contributions in security monitoring

I have also conducted researches in various security monitoring fields, which I do not detail in this manuscript. Thus, I co-advised the Ph.D. thesis of Georges Bossert with Ludovic Mé from CentraleSupélec and Frédéric Guihéry from AMOSSYS. In this project, we worked on the reverse engineering of communication protocols. Indeed, for intrusion detection purposes, it is often necessary to analyze network flows whose protocols are proprietary or whose specification is not public (for example, protocols used in botnets). This work was published at the ACM AsiaCCS 2014 conference [21].

Today, many attacks target the web browser and, more particularly, the JavaScript language. In the SecCloud project, funded by the Labex CominLabs, we proposed a DIFT approach implemented within the web browser with my Ph.D. student Deepak Subramanian. I co-advised Deepak with my colleague Christophe Bidan, and we published our results in FPS 2016 [18] and ACM SIN 2016 [19] conferences. The latter publication won the best paper award.

From 2015 to 2018, I also co-advised the Ph.D. thesis of Oualid Koucham in collaboration with Jean-Marc Thiriet and Stéphane Mocanu, from GIPSA-Lab in Grenoble, and Frédéric Majorczyk from the Direction Générale de l'Armement. We proposed intrusion detection and alert correlation approaches for industrial control systems. This work has been published in the NordSec 2016 [16] and SafeProcess 2018 [8] conferences.

1.4 Organisation of the manuscript

This manuscript highlights the main results of my research on the security of software/hardware interfaces since 2010. Chapter 2 presents our different contributions in the field of hardware-assisted intrusion detection. We detail the different approaches we proposed to isolate the monitor and solve the semantic gap resulting from this isolation. Chapter 3 focuses on the step following the detection, i.e., attack reaction, and gives an overview of our work on intrusion survivability at the OS (Operating System) level. Chapter 4 describes our effort in formalizing hardware-based security mechanisms. Finally, Chapter 5 concludes this manuscript and gives future directions I would like to explore.

Chapter 2

Hardware-assisted intrusion detection

2.1 Introduction to hardware-assisted intrusion detection

Intrusion detection approaches can be split into two different categories. Misuse-based approaches, commonly used in the IDS (Intrusion Detection System) available on the market, define the malicious behavior using attack signatures. Such approaches are only effective in detecting attacks that are publicly known at the time of signature definition. It is, therefore, necessary to regularly update these signatures, and new attacks, exploiting so-called zero-day vulnerabilities, can remain undetected. I am focusing my research on the second category, i.e., anomaly detection. Such approaches rely on the definition of legal behavior and consider any significant deviation from this behavior as intrusions.

The use of intrusion detection systems directly located on monitored machines, i.e., HIDS (Host-based Intrusion Detection System), poses several challenges:

- These approaches can be costly in terms of execution time, which can penalize the performance of programs running on the monitored system;
- The intrusion detector is located on the monitored target and must also protect itself from attacks initiated from a compromised target.

My current research aims at addressing these challenges by taking advantage of dedicated hardware components. Using a dedicated coprocessor allows for offloading the detection computation from the target processor. Moreover, placing the IDS in an isolated hardware component protects it from software attacks executed on the target processor.

2.1.1 Hardware-based security mechanisms

Several academic works have focused on using hardware support to detect or prevent software attacks. Some approaches rely on the existing mechanisms of the processors available on the market, e.g., virtualization extensions or existing enclave mechanisms. Other approaches propose new hardware features, e.g., ISA (Instruction Set Architecture) extensions or security coprocessors. Such approaches cannot be validated on existing processors but require implementations on FPGA (Field-Programmable Gate Array) or simulations. Zhao et al. provide a survey of these different hardware-based security mechanisms, the properties they guarantee, and the vulnerabilities that affect them [Zha+19].

The advantages of hardware-based solutions compared to purely software-based approaches are multiple:

- hardware features cannot be modified, which makes them more robust against software attacks;
- they offer better efficiency of execution time and energy consumption;
- the security mechanisms expose a reduced and controlled interface, which allows better protection of data and code.

Historically, processors have implemented mechanisms to isolate different user applications and the OS kernel. Many processors also provide hardware extensions to isolate different virtual machines and the hypervisor that controls them. These mechanisms rely on a vertical trust hierarchy: isolation between components located in the same level is implemented and controlled by a component located in a lower level, which must be trusted. This approach requires trusting the low-level software components, such as OSes or hypervisors. Some work also proposed enclave mechanisms to isolate the critical parts of an application without trusting these low-level components.

Some work also proposed enclave mechanisms to isolate the critical parts of an application without trusting the low-level components. Such approaches can rely on pre-existing processor features, such as Intel TXT (Trusted Execution Technology) [McC+08] or virtualization extensions [RR18]. Other projects have proposed modifying the architecture of the processors to implement dedicated mechanisms [Owu+13; Noo+13]. Hardware manufacturers also adopted these approaches to provide dedicated hardware features to implement enclaves in their processors, such as Intel SGX (Software Guard Extensions) [Cor] or ARM TrustZone [Lim].

However, such approaches execute the isolated software component within the platform processor. They rely on isolation mechanisms provided by the processor but the trusted and isolated software share hardware resources, e.g., CPU (Central Processing Unit) cores, caches, RAM, and peripherals. This sharing of hardware resources makes them, in general, vulnerable to side-channel attacks [Wan+17; Cho+18; Sch+20]. In addition, they may be vulnerable to a defect in the isolation mechanism [She; 19a].

2.1.2 Coprocessor-based intrusion detection

Some approaches have proposed using a dedicated processor, physically separated from the main processor of the platform, to strengthen the isolation of security mechanisms. Historically, IBM was one of the first manufacturers to follow this approach with the IBM 4758 processor [Dye+01]. The goal was to provide a TEE (Trusted Execution Environment) to run secure applications. In addition, IBM researchers proposed using this coprocessor to implement a monitoring service for applications running on the main processor to detect possible intrusions [Zha+02a].

Nowadays, various processor or motherboard manufacturers integrate such coprocessors into their platforms, e.g., AMD PSP (Platform Security Processor) [EB20], Intel CSME (Converged Security and Management Engine) [Cor20], Apple SEP (Secure Enclave Processor) [MSW] , or HP SureStart [HP 19; ANS]. However, their intrusion detection capabilities are limited. They are mainly used to implement cryptographic primitives or DRM (Digital Rights Management), protect confidential data (e.g., cryptographic keys or biometric data), or implement a virtual TPM (Trusted Platform Module). Some of these coprocessors can provide services to verify the

integrity of the firmware [HP 19; ANS] or the hypervisor [BS08a]. Nevertheless, these different technologies are proprietary, and little public information is available about their internal design. Moreover, their implementation details have often been revealed thanks to reverse engineering efforts that exploit vulnerabilities [EB20; FO18; GE19]. In addition, they essentially allow the implementation of security services provided by the platform designer.

Some research works have also focused on using an isolated coprocessor to monitor applications running on the main processor, including the OS and the hypervisor [Zha+02a; Pet+04a; Pet+06a; WSG10]. Such snapshot-based approaches regularly inspect the content of the system memory. However, they are vulnerable to transient attacks, which attack the monitored system and then clean up their trace between two memory acquisitions. The inability to detect transient attacks motivated researchers to develop event-driven approaches where the monitor observes events generated by the monitored system to detect intrusions [Moo+12a; Lee+13a; Lee+15b; Kor+16; Zho+21].

These different approaches focus on integrity checking. However, coprocessor-based intrusion detection can verify more complex security policies, such as DIFT (Dynamic Information Flow Tracking) [KDK09a; Lee+15a] or CFI (Control-Flow Integrity) [Tim+14; Lee+15c]. These approaches are an alternative to software-based approaches or verifications implemented inside the main processor. The use of hardware support allows to protect the detection mechanism and to speed up the processing. In addition, offsetting in an external processor has the advantage of not involving invasive modification of the main processor microarchitecture.

2.1.3 Our contributions

By isolating the security monitor, the problem of the semantic gap arises [Jai+14; BAL15b]. The external processor has limited visibility into the status of the main processor. To bridge this gap, we have to extract the necessary information from the main processor and send it to the security coprocessor. This information depends on the implemented security policy.

We proposed different approaches to isolate intrusions detection monitors in dedicated coprocessors. More precisely, we explored different strategies to bridge the semantic gap resulting from the isolation of the monitor. We designed solutions that do not require any modification of existing processors, which could ease their adoption by manufacturers.

In the Ph.D. of Ronny Chevalier, we developed a coprocessor-based monitoring approach to detect intrusions against the runtime services of the BIOS. We rely on the static analysis of the BIOS to extract the legitimate behaviors of the monitored services, i.e., their control flow and some data invariants. Moreover, we instrument the code of the services to send information to the monitor at runtime and bridge the semantic gap. We detail this first contribution in Section 2.2.

In the HardBlare project, funded by the Labex CominLabs, we defined a coprocessor-based approach to monitor the execution of Linux applications. We implemented a DIFT monitor on the FPGA part of a ZYNQ SoC. We rely on a combination of different strategies to bridge the semantic gap. We use the ARM PTM trace mechanism available on the hardcore CPU of the ZYNQ SoC to send information about application control flows to the monitor implemented on the FPGA. The monitor combines those traces with annotations computed during the compilation of the application to track information flows. Section 2.3 presents this second contribution.

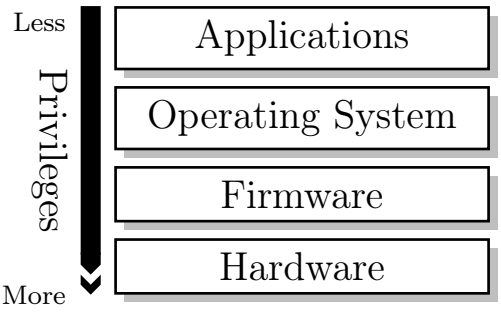


Figure 2.1: Computer abstraction layers

2.2 Detecting intrusions against the SMM

2.2.1 Introduction to SMM monitoring

Firmware, such as UEFI boot firmware or BIOS (Basic Input/Output System), is one of the most privileged software components in the hardware platform, as illustrated by Figure 2.1. Indeed, it has direct access to hardware, is the first piece of software executed during the boot sequence, and is in charge of the initialization of the platform. Any alteration of the behavior of the firmware can have dramatic consequences for the integrity, confidentiality, or availability of the upper layers, i.e., the OS and applications. As a consequence, attackers are increasingly targeting such a privileged component [Lin13; Liv11; Gal13; Res18]. Indeed, classical anti-malware tools are not able to monitor the behavior of the firmware. Such malware can survive after a complete reinstallation of the OS and applications, even if the hard drive is changed.

The SMM (System Management Mode) is the most privileged execution mode on the x86 architecture, introduced by Intel in its 386SL processor. It allows the execution of privileged runtime services [ZRM10, Chapter 5] of the boot firmware, even after the boot sequence, during the execution of the OS [Int19d; YZZ14]. Such services can access a dedicated part of the memory called the SMRAM (System Management RAM), which is only accessible in SMM. The SMBASE (SMRAM Base address) register stores the base address of the SMRAM, and a dedicated interrupt called SMI (System Management Interrupt) switches the processor into SMM. Software code can also make the hardware trigger this interrupt. When receiving the interrupt, the CPU saves its context, switch to SMM, and executes code located at `SMBASE` + `0x8000`. Leaving the SMM is done by executing a special-purpose instruction called `rsm` (for *resume*).

The SMM was initially not designed for security but for allowing the OS to delegate specific tasks to the firmware that are very dependent on the hardware configuration, such as power management. Indeed, the motherboard manufacturer is partially involved in the development of the boot firmware. However, this mode has, over time, acquired fundamental importance in the security of the hardware platform. It isolates the execution of some critical services, such as updating the firmware code or configuring the TPM. Indeed, only a service executed in SMM can unlock the flash memory containing the firmware code to modify its content.

Existing security mechanisms used to protect the boot firmware mainly consist in verifying its integrity at boot time using signed updates [Coo+11; Reg18], Intel Boot Guard [YZ], or TPM measurements [Tru11]. However, such approaches let the runtime services unprotected between two consecutive reboots. Those services are not exempt from vulnerabilities that attackers

can exploit to disable boot time protections and remain persistent [Ole16b; WT09; Ole16a; Baz+15; Bul+17; Puj16]. In particular, they are prone to memory corruption bugs [Cor13; Cor16; Len16a; Len16b] since they are developed using low-level languages, like C or assembly language.

Even if a vulnerability related to the SMM has been found, reported, and patched in the BIOS's source code, all affected platforms need to update their BIOS. In practice, however, the BIOS is not updated frequently [KK15]. Consequently, there is a need to protect privileged runtime services at runtime to prevent attackers from altering their legitimate behavior. In the following section, we present our contribution to runtime firmware intrusion detection.

2.2.2 SMM intrusion detection approach

In the context of Ronny Chevalier's Ph.D., we proposed an event-based behavior monitoring approach that relies on a dedicated coprocessor to isolate the monitor. This work has been funded and realized in collaboration with HP Labs. Figure 2.2 gives a general overview of our approach and how we apply it to detect intrusion against UEFI privileged runtime services. The main goal was to propose a solution to protect the BIOS of HP laptops using a dedicated micro-controller located on the motherboard. However, this approach is quite generic and could be adapted to monitor other targets and use different detection approaches.

The main characteristics of our approach are the following:

- We isolate the monitor into a coprocessor separated from the main CPU, which uses a dedicated memory;
- To bridge the semantic gap created by the isolation of the monitor, we rely on a unidirectional communication channel between the monitored processor and the monitor, and we enforce the communication by instrumenting the monitored code;
- We rely on a compile-time static analysis of the source code of the monitored software to extract the expected legitimate behavior of the target. Then, the monitor uses this model at runtime to detect intrusions.

Isolation of the monitor

The monitor is a trusted component that we rely on to detect intrusions in our system. In the future, we could also use it to start remediation strategies and restore the system to a safe state. It is thus crucial to ensure its integrity.

When the target and the monitor share the same resources (e.g., CPU or memory), it gives the attacker a broad 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 [ARM09]). However, we cannot rely on those mechanisms to monitor the code running inside such isolated environments.

In our approach, we use a coprocessor to execute the monitor. Our design relies on an ARM Cortex A5 coprocessor, similar to AMD PSP. Such a coprocessor uses a dedicated memory. Thus, attackers cannot directly access this isolated memory even if they successfully compromised the target. Instead, they could only influence the monitor's behavior via the communication channel, which becomes the only remaining attack surface. However, the simplicity of such an

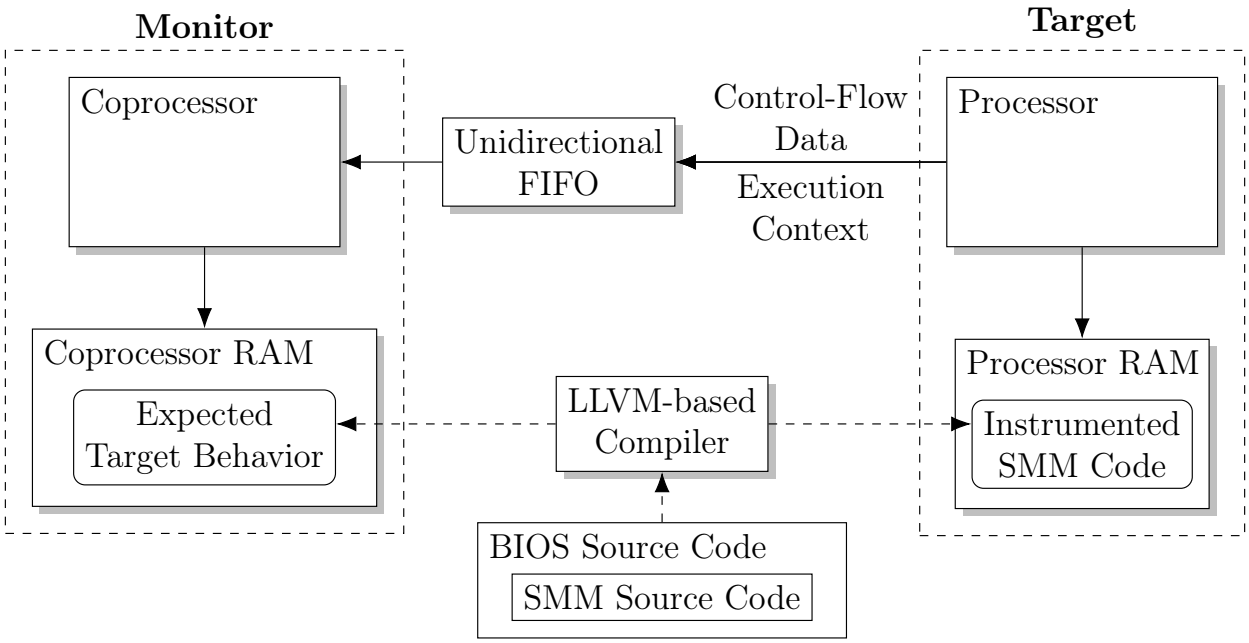


Figure 2.2: High-level overview of our approach

interface makes it harder to find vulnerabilities and attack the monitor. Thus, such a design reduces the attack surface.

Communication channel

Being isolated from the target, the monitor cannot retrieve all the execution context of this target. The isolation creates a semantic gap between the current behavior of the target and what the monitor can infer about this behavior. For example, the monitor does not have sufficient information to infer the virtual to physical address mapping or the current execution trace.

We introduce a communication channel between the monitor and the target. It allows the target to send different types of information, such as the value of a variable stored in memory or inside a register. The nature of such information depends on the detection approaches implemented on the monitor, providing flexibility in our approach.

We propose implementing this communication channel with a dedicated FIFO (First In First Out), connected between the main CPU and the coprocessor. This FIFO is memory-mapped into the main CPU physical memory, and the monitored code can only write into it. Thus, attackers cannot modify or delete messages sent to the coprocessor.

We ensure that the monitored code has exclusive access to the FIFO to prevent other software and hardware component from forging messages faking a legitimate behavior. In our use case, we rely on the SMIACT# signal of the CPU, which is only active is once the CPU is in SMM. We configure the FIFO such that the code executed on the CPU can only write into the FIFO if the CPU is in SMM. This configuration prevents malware executed in kernel or user mode from sending fake messages to the monitor.

We propose to use a fast interconnect between the main CPU executing the monitored component and the FIFO such as QPI (QuickPath Interconnect) [Int09b] from Intel or HyperTrans-

port [Hol+08] from AMD.[1]

Intrumentation and static analysis

We rely on static analyses and code instrumentation performed at compile time to infer the legitimate behavior of the target and enforce the communication between the target and the monitor.

We can implement different detection methods using our approach. Since we target the SMM, we chose the detection methods based on the vulnerabilities most likely to affect the privileged runtime services. Those services being written in C, the lack of memory safety is one of the primary sources of vulnerability. Attackers can corrupt the memory, e.g., by exploiting some buffer overflow to alter the control flow or modify the service data. Since we can rely on page table protections to prevent the attacker from modifying the code and executing code injected into data, ROP (Return-Oriented Programming) is one of the most significant remaining threats. Attackers can also perform non-control data attacks to modify some sensitive data used by the privileged services.

Thus, in our work, we decided to implement two different types of detection methods:

- CFI, to prevent attackers from altering the control flow;
- integrity verification of some critical data saved in the SMRAM.

We consider the two methods the most relevant based on the state-of-art attacks targeting the SMM.

We enforce a CFI policy to protect the forward edges (targets of indirect calls) and the backward edges (return addresses) of the CFG.

We propose a type-based CFI [NT14; Tic+14; PaX15] approach to protect the 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.

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 [Bur+17b] containing all the possible targets for each call site.

Figure 2.3 illustrates our approach to protect indirect calls. During the compilation of the privileged runtime services:

1. We assign a unique identifier (call site ID) to each indirect call site.
2. We also generate a mapping between each indirect call site ID and the corresponding type of the function called.
3. Then, we instrument each indirect call site to send the call site ID and the branch target address to the monitor before executing the indirect call.

[1] 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 [Int17] and AMD Infinity Fabric [Lep+17].

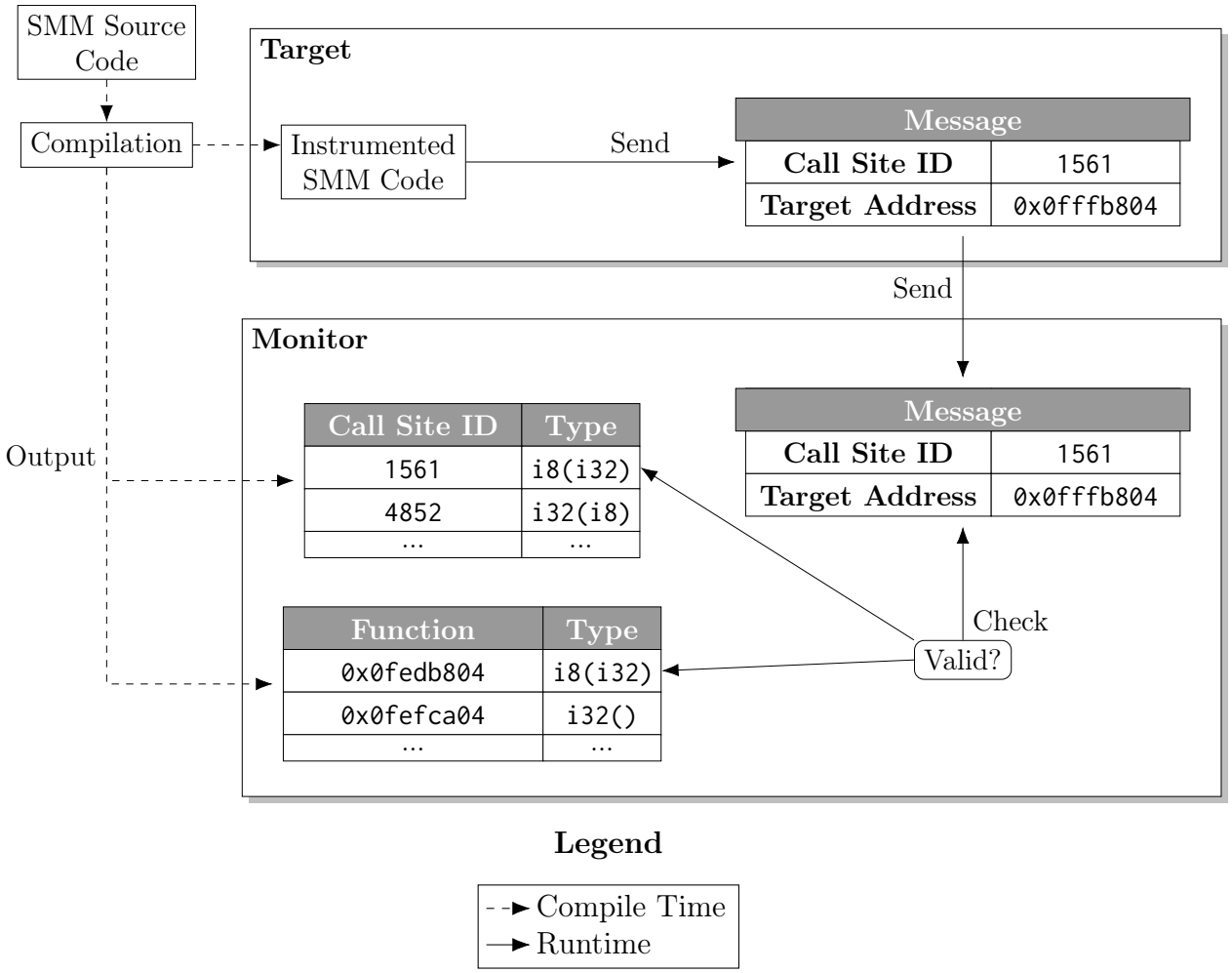


Figure 2.3: How the monitor detects illegitimate indirect calls

4. Finally, we generate a mapping between the address and the type of all the functions targeted by indirect calls.

Both the mappings are sent to the monitor at boot time.

At runtime, the monitor receives the message sent by the monitored code before each indirect branch. The monitor then checks if the function type of the called function is the same as the one associated with the call site.

To protect backward edges, we implement a shadow call stack. This structure is a copy of the regular stack, which only contains the succession of return addresses. Contrary to the regular stack, this structure is protected in the memory of the coprocessor, and attackers cannot modify it.

To implement such an approach, we instrument the SMM code at the prologue and epilogue of each function. The instrumented code sends a push message at the epilogue and a pop message at the prologue, as illustrated by Figure 2.4. Each message contains the return address pushed or popped from the regular stack by the function code. At runtime, the monitor performs the following actions:

- If it receives a push message, it pushes the return address of the message on the shadow stack;

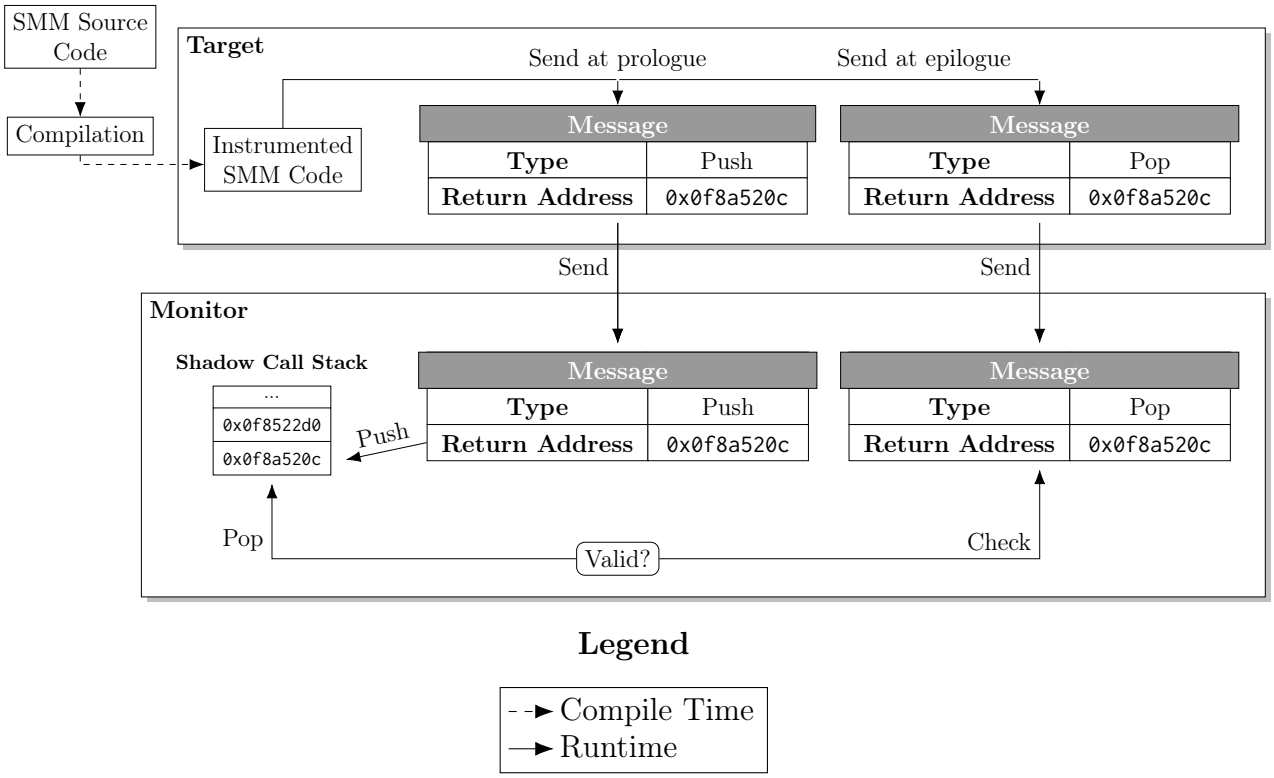


Figure 2.4: How the monitor detects illegitimate returns using a shadow call stack

- If it receives a pop message, it pops the address from the shadow stack and compares it with the message's return address.

Compared to other shadow call stack implementations, we do not have to deal with exceptions, longjmp, or even multi-threading due to the SMM environment and constraints [PC03].

We also check the integrity of some sensitive data saved in the SMRAM to detect any malicious data modifications. For example, when entering SMM, the CPU stores its context, including the value of the SMBASE register, in the save state area [Int19d]. The processor uses this register every time an SMI is triggered to jump to the SMM entry point. When exiting the SMM, it restores the value of the SMBASE register from the value stored in the save state area.

Attackers do not have direct access to the SMBASE register, but they can 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 the DRAM (Dynamic Random Access Memory). At runtime, however, there is no valid reason to modify this value. If an attacker changes the SMBASE value, it may result 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 two successive SMI. We instrumented the BIOS code to send the value of the SMBASE initialized at the end of the boot sequence 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 an attacker modified the SMBASE to compromise the SMM.

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 [Jan+14]. We need to protect their integrity since recent firmware use page tables [Wil15; YZ15; YZF15]. 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 saved in memory before the `rsm` instruction. The monitor can then detect attacks similar to the one described by Jang et al. [Jan+14].

Implementation

We rely on LLVM 3.9 [LA04], a compilation framework widely used in the industry and the research community, to analyze and instrument the SMM code. Ronny Chevalier implemented 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., that indirect calls always branch to legitimate targets. It is performed on the LLVM IR (Intermediate Representation) 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. Moreover, and more importantly, we want to ensure that it will not be optimized away or placed outside the prologue or epilogue of the functions.

2.2.3 Evaluation and results

At the time of the experiments, to the best of our knowledge, no off-the-shelf FPGA-based solution with direct access to HyperTransport or Intel QPI was commercially available. Therefore, we used a more flexible simulation-based prototype to evaluate our approach.

Experimental setup

We used EDK II [Tia17] and coreboot [The17], two real-world implementations of code running in SMM. We built these implementations using our modified LLVM toolchain, and we only instrumented their SMM-related code.

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 do not accurately model x86 or ARM platforms' performance.

On the other hand, simulators try to model the performance of the platforms accurately 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 the performance of our implementation accurately.

We used the QEMU 2.5.1 emulator [Bel05] for the security evaluation, and we modified it to emulate our communication channel.

We used the gem5 cycle-accurate simulator [Bin+11] to estimate the performance impact on the main CPU by modeling an x86 system and on the coprocessor 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.

Security evaluation

There is no public data set of vulnerable SMM codes in contrast to userland applications. Indeed, attacks targeting the SMM are highly dependent on the architecture and the proprietary BIOS code. However, these BIOS codes are not publicly available and would not execute on our experimental setup.

Consequently, we have implemented SMI handlers with vulnerabilities similar to previously disclosed ones affecting real-world firmware. We reproduced attacks exploiting the following vulnerabilities giving arbitrary execution:

- a buffer overflow in an SMI handler, which allows an attacker to modify the return address stored on the stack [Kal+13];
- an arbitrary write allowing an attacker to modify a function pointer used in an indirect call [Ole16a];
- an arbitrary write allowing an attacker to modify the SMBASE [Puj16];
- an insecure indirect call that retrieves the function pointer from a data structure controlled by the attacker [Ole16b].

As shown in Table 2.1, the monitor detected all these attacks since they modifed the control flow of the SMM code. We did not encounter false positives, which we expected since we used a conservative strategy regarding indirect calls. Also, while poor software engineering practices using function typecasting could introduce false positives, we did not encounter such cases in our evaluation.

Vulnerability	Attack Target	Security Advisories	Detected
Buffer Overflow	Return address	CVE-2013-3582 [Cor13]	Yes
Arbitrary write	Function pointer	CVE-2016-8103 [Cor16]	Yes
Arbitrary write	SMBASE	LEN-4710 [Len16a]	Yes
Insecure call	Function pointer	LEN-8324 [Len16b]	Yes

Table 2.1: Effectiveness of our approach against state-of-the-art attacks

Notice that 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 exploit a type collision and redirect the control flow to another function with the same type signature.

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. We analyzed the code of EDK II and found that the vast majority of function types only correspond to a unique function, as illustrated by Table 2.5.

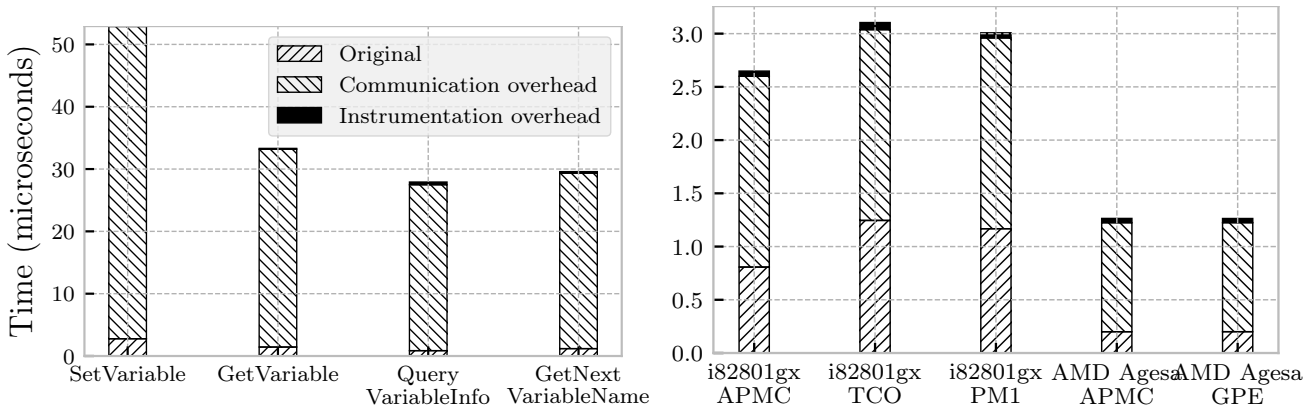
Number of Distinct Equivalent Classes	158	24	42	2	1	1
Size of the Equivalent Classes	1	2	3	5	9	13

Figure 2.5: Function types equivalent classes in EDK II

Performance evaluation

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. 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. Figure 2.6 illustrates the results we obtained.



(a) EDK II VariableSmm

(b) coreboot

Figure 2.6: Time (in microseconds) to execute each SMI handler (averaged over 100 executions)

We can observe a high overhead due to the communication, even with a low latency communication channel (128 ns). The instrumentation overhead, on the other hand, is negligible in comparison. 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 recommended by Intel [Int11]. Thus, the impact of our approach on the system's performance is low and not noticeable for the user.

2.2.4 Conclusion

In this section, we proposed an event-based detection approach for low-level software using three key components: a coprocessor to isolate the monitor, a communication channel to reduce the semantic gap, and a compilation phase to extract the correct behavior of the target. We show that we can use this approach to detect intrusions targeting SMM services by verifying a CFI policy and the integrity of security-critical registers (CR3 and SMBASE). 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 simulating the coprocessor 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 was presented and published at ACSAC 2017 conference [10] and was done during the Ph.D. of Ronny Chevalier. I co-advised Ronny with David Plaquin, in a bilateral collaboration project with HP Labs at Bristol. Ronny realized all the development of the prototype, which is the property of HP and has not been released in open source. This work has also been the subject of two patent applications currently in progress: EP3413531A1 and EP3413532A1.

The following section presents our second contribution in the field of hardware-assisted intrusion detection.

2.3 Hardware-assisted Information Flow Tracking

In the HardBlare project, we proposed a generic anomaly-based approach to detect software attacks against confidentiality and integrity at different levels, e.g., low-level attacks such as control-flow hijacking and more high-level attacks such as confidential files leakage. In that context, we propose a hardware-assisted DIFT approach.

2.3.1 Introduction to Dynamic Information Flow Tracking

DIFT consists in:

1. Attaching **labels** called tags to **containers** (e.g., files, program variable, or registers) and specifying an information flow **policy**, i.e., relations between tags;
2. **Propagating** tags at runtime to reflect information flows that occur during execution and **detecting** any **policy violation**.

DIFT can be implemented at different levels. OS-level DIFT [Efs+05; Zel08; Roy+09; Geo+17] is a coarse-grained approach which only monitors system calls. OS-level containers are files or memory pages. The monitor is usually implemented in the OS kernel in such approaches, protecting it from userland applications. Moreover, the end-user can easily specify the policy by tagging files. These approaches also have a relatively low runtime overhead on the execution. However, they suffer from two main limitations:

- They over-approximate the internal behavior of applications;
- They cannot be used to detect low-level attacks such as control-flow hijacking, which requires handling more fine-grained containers such as registers.

To illustrate the problem of over-approximation, let us consider an application that reads the file f_1 and then writes some data into the file f_2. OS-level approaches consider that the read syscall on f_1 followed by a write syscall on f_2 always results in an information flow from f_1 into f_2. However, if the application code does not use the data read from f_1 to compute the data written into f_2, there is indeed no information flow.

Fine-grained software approaches [Qin+06; NS05] monitor each instruction executed by an application. They consider fine-grained containers such as registers or words stored in memory. Such approaches can track information flows more precisely and detect low-level attacks, e.g., control-flow hijacking. However, DIFT monitors implemented in software [Qin+06; NS05] are not isolated from their target since they weave the monitor into the application code.

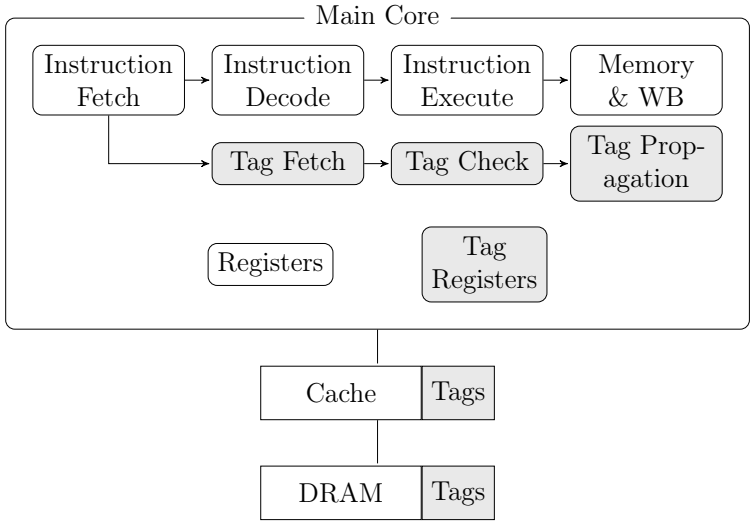


Figure 2.7: In-core DIFT (adapted from [KDK09b])

Implementing a fine-grained approach in hardware is a solution to protect the monitor from software attacks. In-core approaches [DKK07; Dha+15; Suh+04], illustrated by Figure 2.7, modify the microarchitecture of the CPU to compute tags in parallel to regular operations. Such approaches extend memory and caches with tag bits. Each stage of the pipeline is duplicated with a specific hardware module to realize tag-related operations all along with the program execution. However, those modifications to the core microarchitecture are invasive, which limits their adoption and industrialization. Thus, those approaches are only validated on simulation or implemented in softcores.

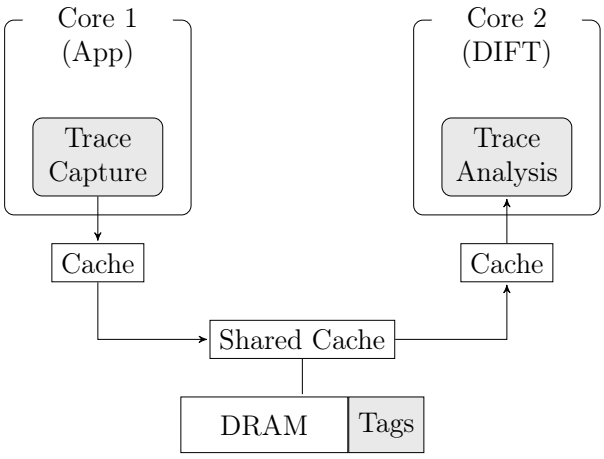


Figure 2.8: Offloading DIFT (adapted from [KDK09b])

As illustrated by Figure 2.8, offloading approaches [Che+08a; Ruw+08] take advantage of multi-core systems to offload tag computation on a general-purpose core. In addition to program execution, the monitored core also compresses and stores the information required for

DIFT inside a shared buffer located in memory or cache. The second core decompresses this information and realizes tag computation to check whether an illegal information flow occurred. This approach does not require invasive modification of the processor. However, it wastes a general-purpose core for DIFT operations, which is not energy efficient.

Off-core approaches [KDK09b; Ven+08] use a custom DIFT coprocessor to compute and check tags. The idea is similar to the offloading solution, but instead of wasting a general-purpose core, this approach uses a custom DIFT coprocessor. The coprocessor is thus dedicated and optimized for DIFT. However, some modifications on the main core are still required to send the execution trace to the coprocessor. Figure 2.9 illustrates such approaches.

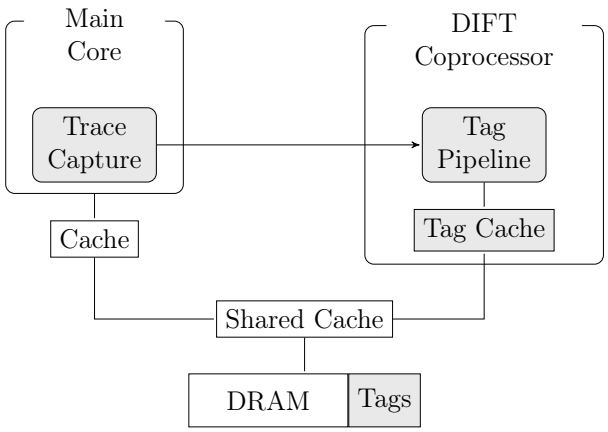


Figure 2.9: Off-core DIFT (adapted from [KDK09b])

2.3.2 HardBlare: a hardware/software codesign for DIFT

In the HardBlare project, we target embedded systems using rich OSes in security-critical contexts. Such systems cannot be redeveloped from scratch for economic reasons. However, it is possible to make changes to the hardware and software for security, provided that we maintain compatibility with the OS's applications and drivers. In this context, we proposed an off-core DIFT approach to detect a broad range of attacks. The originality of our approach lies in the following:

- We combine fine-grained DIFT with OS-level tagging to be able to attach tags to files. This hybrid approach can save the security contexts between reboots, files being persistent containers. It also helps the end-user to specify the security policy.
- We implement tag propagation in a hardware coprocessor to isolate the monitor. Contrary to other hardware approaches, our coprocessor approach requires no modification of the main CPU.

Isolating the monitor in a dedicated coprocessor creates a semantic gap between the monitor and the monitored system. This semantic gap leads to the following challenge: how can the isolated coprocessor extract some information from the main CPU to infer the behavior of the monitored code? Reducing this gap without modifying the main CPU is one of the main challenges of the project. We solve the semantic gap issue by an original combination of approaches:

- We pre-compute **annotations** during the compilation of applications. Those annotations reflect the information flows in each basic block;

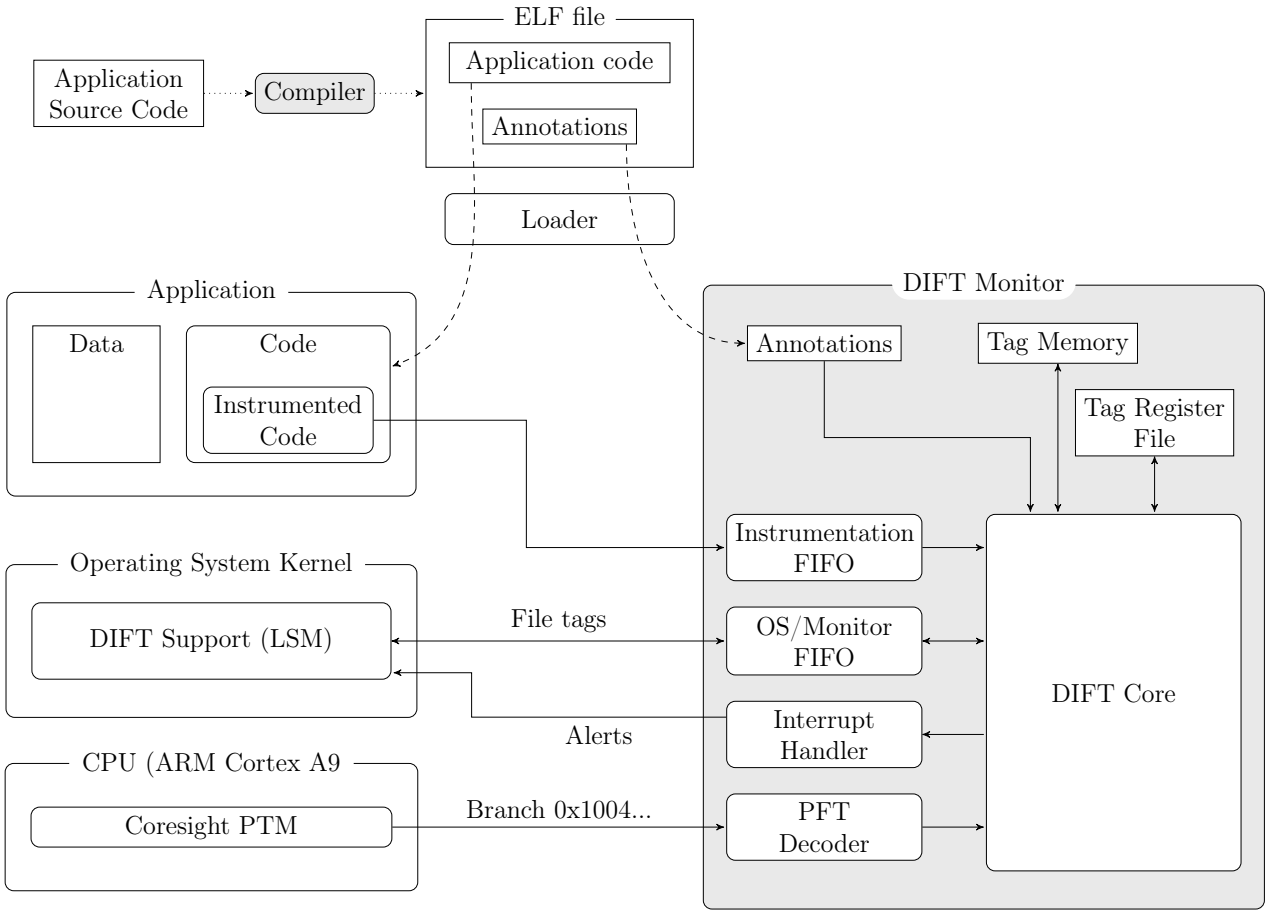


Figure 2.10: General overview of HardBlare

- We send branching information using **hardware trace mechanisms** at runtime;
- We send addresses of data memory accesses by **instrumenting** the application code.

Our approach can be implemented on systems using an SoC (System on a Chip) that combines a CPU and an FPGA, such as Xilinx ZYNQ[2] or Intel/Altera Cyclone VS[3]. In this type of SoC, the CPU corresponds to the PS (Processor System) and the FPGA to the PL (Programmable Logic). We execute the monitored system on the PS and the DIFT monitor on the PL. Our experiment relies on the Digilent ZedBoard[4]. This board uses a Xilinx ZYNQ SoC, which combines a dual-core ARM Cortex-A9 CPU with a Xilinx FPGA.

Figure 2.10 gives a general overview of our approach. The DIFT monitor is implemented on the FPGA of the SoC. It monitors the behavior of applications executed on the main CPU and manages tags corresponding to fine-grained containers (CPU registers and words stored in the application's memory). Those tags are themselves stored in two different memory regions:

- Tags corresponding to registers of the main CPU are stored in *shadow registers* inside the FPGA (Tag Register File);
- Tags corresponding to the application memory are stored into a dedicated part of the memory (Memory tags) that only the DIFT monitor can access.

[2] `https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html`
[3] `https://www.intel.com/content/www/us/en/products/programmable/soc/cyclone-v.html`
[4] `http://zedboard.org/`

To capture the runtime behavior of the applications and solve the semantic gap issue without modifying the main CPU, we rely on four communication channels: trace, annotation, instrumentation, and OS/monitor channels.

We configure the ARM CoreSight PTM (Program Trace Macrocell) hardware trace mechanism so that the CPU automatically sends branching information to the monitor executed on the FPGA. This trace mechanism was initially designed for debugging purposes. In HardBlare, we use it to monitor the executed application for security purposes. However, the PTM can only send sparse information about the behavior of the monitored application. It only provides the address taken in branching instructions and exceptions. Thanks to this mechanism, the monitor can track the address of each basic block executed by the application. Nevertheless, this information is not sufficient to retrieve the information flow that occurred in each basic block.

During the compilation of each application, we statically analyze application's code and compute annotations for each basic block. Then, we saved those annotations in a dedicated section of the elf binary file. Annotations correspond to information flows that will occur in each basic block. They describe the tag propagations that the monitor will have to perform at runtime. We also modified the OS loader to send annotations to the coprocessor while loading the application on the main CPU.

Each time the main CPU executes a basic block of the application, the PTM trace mechanism sends the address of this basic block to the DIFT monitor. The monitor then retrieves the annotations corresponding to this basic block and executes them to propagate the tags stored inside the tag register file or the tag memory. It also checks the security policy and sends an interrupt to the OS in case of intrusion. The handling of this interrupt, i.e., the reaction part, is not in this project's scope.

To handle tags corresponding to application data stored in memory, the monitor needs to know the virtual address of each memory access performed by the application. Thus, the monitor can maintain a map between those addresses and the corresponding tags. Sometimes, the address of memory accesses is computed at runtime and saved in a register by the application code. In this case, the address can be difficult to predict at compile time. We instrument the program code to solve this issue. The instrumented code sends the address to the coprocessor via a dedicated FIFO just before accessing the memory.

Finally, we associate a tag to each file of the system. We saved those tags in the extended attributes of the file. We propagate the file's tag to the tag corresponding to the memory buffer each time the monitored application reads a file. We do the opposite when the application writes into files. Tags associated with files cannot be handled directly by the DIFT monitor but only by the OS. To manage such cases, we modified the Linux kernel. Our modified kernel sends the file tag to the coprocessor for each `read` syscall thanks to a dedicated FIFO. For each `write` syscall, the kernel retrieves the tag corresponding to the buffer from the FIFO and propagates it to the file attributes.

DIFT coprocessor

We use different strategies to implement the coprocessor. Our first solution, mainly developed by Mounir Nasr Allah during his Ph.D., relies on a generic MIPS-based soft core. In this case, the firmware executed by the coprocessor implements all the DIFT logic. Mounir has integrated this flexible solution with all the other software components of HardBlare.

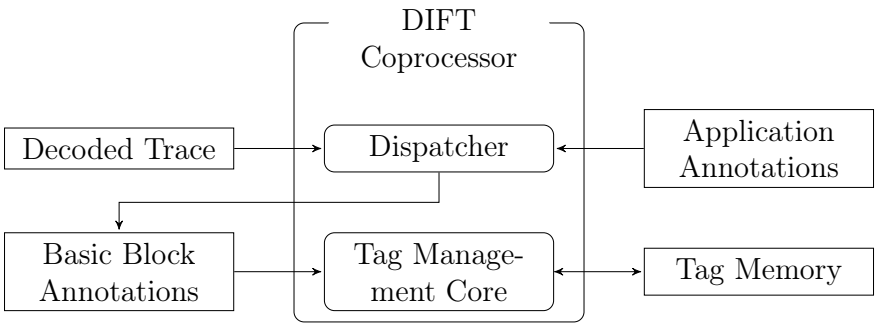


Figure 2.11: DIFT coprocessor.

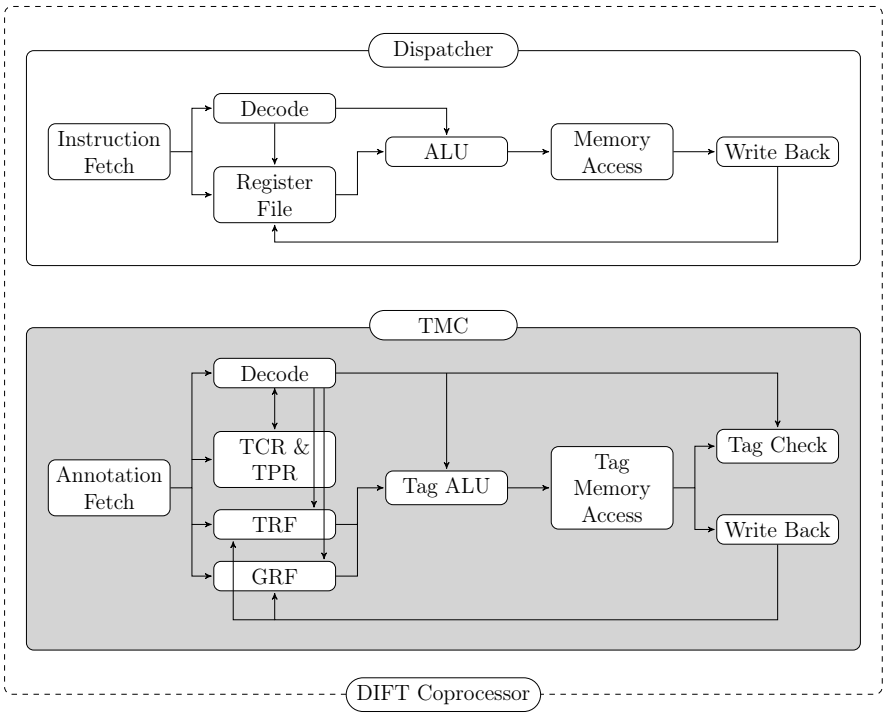


Figure 2.12: DIFT coprocessor internal architecture.

We also designed and implemented a dedicated multi-core DIFT coprocessor on FPGA. Muhammad Abdul Wahab is the principal contributor to this work, he realized during his Ph.D.

Figure 2.11 gives a general overview of this coprocessor, and Figure 2.12 illustrates its internal design. The first core (Dispatcher) is a generic fully-pipelined MIPS processor. It executes a firmware that parses the trace from the PTM, identifies the current basic block, and retrieves the corresponding annotations from the annotation memory. The dispatcher then sends those annotations to the internal program memory of the second core, i.e., the TMC (Tag Management Core). The TMC executes the annotation to propagate tags and checks the security policy.

In such a multi-core design, we can use multiple TMCs to handle the following use-cases:

- We can enforce different tag propagation and check policies in parallel (one policy by TMC). This approach therefore detect different types of attacks.
- We can monitor different applications in parallel (one application by TMC).

Trace generation and decoding

In HardBlare, we rely on instruction tracing mechanisms provided by existing processors to bridge the semantic gap between the application executed on the main CPU and the DIFT monitor isolated on the coprocessor. We implemented our approach on the ARM Coresight PTM tracing mechanisms available on the Zynq SoC.

When this component is activated and correctly configured, it generates traces at waypoints. These points correspond to modifications of the value of the PC (Program Counter) register and therefore allow us to follow the path taken by the processor in the control flow graph. The component receiving the traces should have access to the application code, making it possible to infer all the instructions executed in the trace. More precisely, the PTM generate traces for the following waypoints:

- when the CPU executes direct and indirect branching instructions ;
- when an exception is triggered;
- when the instruction set or the security state of the processor change.

For branching instructions, the trace contains the target address and the result of the conditional code. For exceptions, the trace describes the exception reasons, the address at which this event occurred, and the return address in the application code. Finally, the trace describes the reasons and the consequences of modification of the CPU state.

In our design, these traces are compressed and sent in packets to the TPIU (Trace Port Interface Unit) and then to the security monitor located in the PL using the EMIO (Extended Multiplexed Input/Output) bus. The traces follow the specification of the PFT (Program Flow Trace) [Ltd11]. Finally, the PFT decoder, an hardware component mainly developed by Muhammad Abdul Wahab, decompresses the trace, decodes the messages, and sends the results to a dedicated FIFO.

The PTM has to be configured by kernel code. We had to patch the official Linux PTM driver, which did not fully support all the features of the PTM. This patch, mainly developed by Muhammad Abdul Wahab, has been officially adopted by the vanilla Linux kernel distribution[5]. We also modified the Linux kernel to trace only some processes, based on their PID (Process IDentifier). It is necessary to activate the PTM only when the CPU executes the monitored application. Thus, the trace mechanism has to be stopped when the OS schedules another process or executes some kernel code.

Annotations

The trace provided by the PTM is not sufficient to retrieve all the necessary information needed to track application information flows. It provides only the addresses of the basic blocks that the processor has executed so far. However, the DIFT monitor has to infer the information flow that occurred inside each basic block. We proposed to compute such information at compile-time and to store the results in a dedicated section of the binary application code. We implemented the generation of annotations by modifying the LLVM backend for ARM V7.

We compute the sequence of tag propagation operations, called annotations, that the coprocessor must execute at runtime for each basic block. Each operation of such a sequence corresponds

[5] `https://lore.kernel.org/patchwork/patch/723740/`

to one or several instructions of the coprocessor. Annotations of a basic block reflect the information flows resulting from the execution of the basic block.

We propose associating a tag to each register of the processor and each word stored inside the DRAM. We use a 32 bit ARM Cortex-A9 processor in our implementation, and we store tags in 32-bit tag registers of the coprocessor and 32-bits words of the tag memory area. Only the coprocessor can access tag registers and tag memory area.

We note $\underline{Rn}$ the tag corresponding to the register Rn and $\underline{[Rn]}$ the tag of the memory word stored at the address pointed by this register value. We assume that the monitored application is executed on top of an OS that uses pagination. In that case, the PTM reports virtual addresses in the generated trace. Thus, we associate tags to virtual addresses. To handle implicit flows, we manage the APSR and PC registers differently than other registers. We associate a tag to each field of the APSR status register used in conditional branching instructions, and we associate a stack of tags to the PC register.

In the ARM ISA, instruction operands can be immediate values or registers. Thus, we also associate a default tag to immediate values, noted $\underline{Imm}$. The administrator can specify the value of this default tag at runtime.

We only support the standard ARM processor instruction set, and we do not manage the different complementary instruction sets such as Thumb. Some instructions allow switching from one instruction set to another. In our approach, we assume that these instructions should not be present in the application code. We enforce such a restriction at compile time.

The ARM instruction set offers the possibility to execute most instructions conditionally. However, the PTM generates traces exclusively when executing waypoints. Since conditional instructions that do not operate on the PC register are not waypoints, the PTM does not generate any trace for these instructions. Therefore, analyzing the traces produced by the PTM, we cannot infer whether the ARM processor has executed those instructions and whether the coprocessor has to execute annotations corresponding to these instructions. Therefore, we also modified the compiler's back-end to disable conditional forms for instructions that do not correspond to waypoints. In practice, this restriction is not very penalizing [Che+10b]. ARM took a similar decision for the following versions of their architecture, namely the 64-bit ARMv8 [Che+10b].

The monitor does not track instructions executed in kernel mode since the OS kernel is part of our TCB (Trusted Computing Base). Moreover, we only generate annotations for application instructions that generate information flows from their sources to their destination operands. For some instructions, we have to combine the tags of their different sources before propagating the resulting tag to the destination. We also have to combine the source tags with the destination tags when the instruction only modifies part of the destination operand (e.g., the least significant bytes of the destination operand).

We rely on different combining functions, noted $\underset{T}{\oplus}$, where T is the type of the instruction producing the information flow. The operation performed by each combining function can be selected at runtime by the administrator. Thus, we express each annotation as $\underline{s_1} \underset{T}{\oplus} \underline{s_2} \ldots \underset{T}{\oplus} \underline{s_n} \rightarrow \underline{d_1}, \ldots \underline{d_m}$, with $s_1, \ldots s_n$ the source operands and $d_1, \ldots d_m$ the destination operands. When executing this annotation, the coprocessor combines all the source tags and propagates the result to each destination tag.

We manually analyzed the specification of all the instructions of the ARM V7 ISA. We de-

fine three types of annotations: register-based explicit flows, implicit flows due to conditional branching, and memory-based explicit flows.

Table 2.2 gives an overview of the different types of annotations we can generate to handle explicit flows.

Instruction Type	Example	Annotations
Data move	$mov\ Rd, OpSrc_1$	$\underline{OpSrc_1} \rightarrow \underline{Rd}$
Arithmetic & logic	$add\ Rd, OpSrc_1, OpSrc_2$	$\underline{OpSrc_1} \underset{AL}{\oplus} \underline{OpSrc_2} \rightarrow \underline{Rd}$
Comparison	$cmp\ OpSrc_1,\ OpSrc_2$	$\underline{OpSrc_1} \underset{C}{\oplus} \underline{OpSrc_2} \rightarrow \underline{APSR}$
Arithmetic & logic with APSR update	$adds\ Rd, OpSrc_1, OpSrc_2$	$\underline{OpSrc_1} \underset{ALC}{\oplus} \underline{OpSrc_2} \rightarrow \underline{Rd}, \underline{APSR}$
Memory load	$ldr\ Rn, [Ra]$	$\underline{[Ra]} \rightarrow \underline{Rn}$
Memory store	$str\ Rn, [Ra]$	$\underline{Rn} \rightarrow \underline{[Ra]}$

Table 2.2: Annotations generated for explicit flows

Implicit flows occur due to conditional branching, as illustrated by Listing 2.1. In this example, there is an implicit flow from the conditional expression to variables a and b. Consequently, at line 3, the tag of variable i should be propagated to variable a. However, this control flow dependence only occurs until the immediate post-dominator of the conditional branching. Thus, there is no implicit flow from i to c at line 9.

Notice that implicit flows can also occur due to the non-execution of some branches. For example, assume $i = 0$ in the example. There is still an implicit flow from i to b at line 3, even if b is not modified in the executed branch. Indeed, since b is modified in the alternative branch, an attacker can infer some information on i by observing b.

A pure dynamic approach cannot handle such types of implicit flows. Instead, it would be possible to rely on static analyses to infer the variables modified in alternative branches. However, the problem is quite complex in the general case, since the alternative branch can call functions and use pointers. Static analyses handle those cases by over-approximating the set of modified variables, leading to false positives. Designing such a static analysis is out of the scope of our work, and we do not handle such implicit flows in our implementation. However, we could generate additional annotations corresponding to the flows occurring in alternative branches by relying on external static analysis.

```
1     b = 0;
2     if (i == 0) {
3         a = 1;
4     }
5     else {
```

```
6           a = 0;
7           b = 1;
8       }
9       c = 3;
```

Listing 2.1: Example of indirect information flows

We maintain a security stack of tags associated with the PC register, named $\underline{\widehat{PC}}$, to handle implicit flows. The topmost element of this stack is the label corresponding to all values that influence the current control flow. We named $push_{\underline{\widehat{PC}}}$, $pop_{\underline{\widehat{PC}}}$, and $top_{\underline{\widehat{PC}}}$ the functions that respectively push a tag on the stack, pop, or give the topmost element of the security stack.

We push a new tag on the security stack for each conditional branching instruction depending on the $APSR.n$ field of the status register :

$$push_{\underline{\widehat{PC}}}\left(top_{\underline{\widehat{PC}}}() \underset{B}{\oplus} \underline{APSR.n}\right)$$

Then, we combine the topmost element of the security stack with the combined source tag of each annotation. Thus, we transform each annotation $\underline{s} \rightarrow \underline{d_1}, \ldots \underline{d_m}$, corresponding to direct flows, into the following: $\underline{s} \underset{C}{\oplus} top_{\underline{\widehat{PC}}}() \rightarrow \underline{d_1}, \ldots \underline{d_m}$

Finally, we pop the last element of the security stack on each immediate post-dominator of a conditional branching instruction.

Instrumentation

In ARM V7, the address of load/store instructions is specified in a register. The value of that register can be challenging to infer statically at compile-time since it can result from complex computations. This computation can also take input depending on the execution context (e.g., user inputs). However, the monitor needs the value of the address register to propagate the tags. For example, if the application executes the instruction `ldr r5,[r0]`, the monitor needs the value of `r0` to compute the annotations corresponding to this instruction, i.e., $\underline{[r0]} \rightarrow \underline{r5}$. However, the monitor does not have access to this value stored in a register of the main processor.

In HardBlare, we proposed to instrument the application at compile-time to send the value of the address register before load/store instructions. To that end, we rely on a dedicated FIFO. In our implementation, this FIFO is synthesized on the FPGA and connected to the main processor via the AXI (Advanced eXtensible Interface) bus. Thus, software executed on the main processor can send messages to the monitor using memory-mapped IO. However, the FIFO is mapped in the physical address space and cannot be accessed directly by the application code executed in user mode. In our implementation, we mapped the physical address of the FIFO into the virtual address space of the application using a UIO (Userspace I/O) driver [Koc06]. Thus, the application's code can send messages into the FIFO without having to perform a syscall.

During the compilation process, we add a symbol corresponding to the virtual address of the FIFO. Then, the application calls the UIO driver at load time to map the physical address of the FIFO to this symbol. Finally, we store the virtual address into a dedicated register. In our implementation, we have reserved the register `r9` for this purpose. We modified the

LLVM backend to never use that register during the compilation of the application code. This approach simplifies the design of the instrumentation at the cost of the loss of one register. We add the instruction `str ra, [r9]` just before each load/store instruction, with `ra` corresponding to the address register of the load/store instruction.

This instrumentation implies a significant overhead on the execution time. We proposed two optimizations to limit this overhead. First, since the local variables are stored on the stack, most load/store instructions use the SP (Stack Pointer) and FP (Frame Pointer) registers to specify the address. To limit the number of instrumentation points, we proposed to track the value of SP and FP on the monitor using additional annotations. The compiler generates those annotations for each basic block, and the monitor maintains shadow copies of those register values. The compiler also instruments the application code to send the initial values of these registers to the monitor at load time. Thus, the monitor can infer the value of the address register of load/store instructions using SP and FP without instrumentation.

The second optimization tries to group several consecutive load/store instructions and sends the address register value using a single `stm` instruction instead of multiple `str` instructions. This optimization delays the sending of addresses until the last load/store instructions. Thus, we can only rely on this optimization if the address registers of the preceding load/store instructions have not been modified in the meantime. We developed a static analysis to identify those cases.

OS

The DIFT monitor executed on the coprocessor can only propagate tags associated with registers and application memory addresses. However, applications often store and read data from files stored on hard drives or solid-state drives. It is thus necessary to associate tags to files and propagate them to follow the information flows involving files. This approach is also useful for restoring the tags after rebooting the platform since files are persistent containers. Finally, it is easier for the end-user to specify the policy by associating tags with files. Indeed, they represent a more convenient level of abstraction.

To manage tags associated with files, we modified Rfblare, a Linux LSM (Linux Security Module) developed in the CIDRE research team, to implement DIFT at the OS level. More precisely, Rfblare relies on file extended attributes to store the tags. We modified the tag propagation logic of Rfblare to offload the propagation to the coprocessor. Rfblare and the DIFT monitor exchange tags using a dedicated OS/monitor communication channel.

Rfblare uses LSM hooks to intercept the read and write syscalls. In our approach, we implement the following propagation logic:

- for the read hook, we send the file tag to the monitor as well as the read buffer address and size;
- for the write hook, we wait for a tag sent by the monitor and attach the received tag to the file.

2.3.3 Evaluation and results

Mounir Nasr Allah developed all the software components of HardBlare and integrated these components with the hardware components developed by Muhammad Abdul Wahab. Muham-

mad evaluated and Mounir realized security and performance evaluation of the whole solution.

Experimental setup We deployed the hardware and software component on the Zedboard, which includes a Xilinx Zynq-7000 SoC.

We developed the following software components:

- a modified version of the Linux 4.9 kernel (4318 LOC)
- a modified version of LLVM (9873 LOC)
- a modified version of musl [lib] C library (776 LOC) and the loader (`ld.so`) of the binutils (190 LOC)

We integrated all the software development in a dedicated Yocto distribution.

The hardware components, including the DIFT monitor, the PFT decoder, and the different FIFO, represent more than 5229 lines of VHDL code.

All the development have been released in open source and are available on a gitlab repository [6].

Security evaluation To evaluate our approach, we developped some example codes that contains the following weeknesses:

- a stack-based buffer overflow (CWE-121);
- data integrity issues (CWE-1214);
- exposure of sensitive information to an unauthorized actor (CWE-200).

If we correctly configure the policy, all the attacks exploiting these vulnerabilities are detected, as expected. The results also show that our approach does not suffer from the overapproximation issue of OS-level DIFT approaches.

Performance evaluation We evaluated the overhead on the execution time of the monitored programs using the MiBench benchmark [Mic]. The overhead is significant (between $\times 4$ and $\times 12$), which is the main limitation of our approach. However, this overhead is mainly due to the instrumentation, and we could reduce it by limiting the number of instrumentation points. For example, in some cases, we could predict the value of the address pointed to by a register by using some static analysis like constant propagation.

choleski	**crc**	**dft**	**fft**	**lu**	**matrix**	**nbody**	**radix**	**wht**
$\times 12$	$\times 8.5$	$\times 9.5$	$\times 5.5$	$\times 5.25$	$\times 3.25$	$\times 4$	$\times 7$	$\times 5$

Table 2.3: Runtime overhead on MiBench.

We also evaluated the area and power overheads of the hardware design. Table 2.4 shows the area results of the DIFT coprocessor developped by Muhammad Abdul Wahab.

[6] `https://gitlab.inria.fr/blare/hardblare`

IP Name	Slice LUTs (in %)	Slice Registers (in %)	BRAM Tile
Dispatcher	2223 (4.18%)	1867 (1.75%)	3
TMC	1837 (3.45%)	2581 (2.43%)	6
PFT Decoder	121 (0.23%)	231 (0.22%)	0
Instrumentation	676 (1.27%)	2108 (1.98%)	0
Blare PS2PL	662 (1.24%)	2106 (1.98%)	0
Blare PL2PS	62 (0.12%)	56 (0.05 %)	0
Decoded trace memory	0	0	2
AXI Master	858 (1.61%)	2223 (2.09 %)	0
TMMU	295 (0.55%)	112(0.10 %)	3
AXI Interconnect	2733 (5.14%)	2495 (2.34 %)	0
Miscellaneous	1381 (2.6%)	2160 (2.03%)	0
Total Design	**10848 (20.39%)**	**15939 (14.98%)**	**14 (10%)**
Total Available	**53200**	**106400**	**140**

Table 2.4: Post-synthesis area results on Xilinx Zynq Z-7020.

Most of the FPGA area is filled by AXI interconnect (5.14%), Dispatcher (4.18 %), and TMC (3.45%). The overall design takes 20.4% of the FPGA area. If we add another TMC to protect another application in parallel, it will take additional 4095 slice LUTs, 9074 slice registers (i.e., 8% additional FPGA logic), and six BRAM (Block RAM) tiles. In other words, the proposed design can run more than eight security policies or protect more than eight processes at the same time

We also used the results provided by Vivado tools using the post-synthesis design to evaluate the power overhead of our approach. We consider that this overhead is only due to the power overhead of the FPGA. According to Vivado, the power overhead of the FPGA is 0.294 W (with a deviation of 20 %), which represents 16.2 % of the baseline power consumption of Zynq processing system (1.815 W).

2.3.4 Conclusion

This section presents my contribution to hardware-assisted DIFT. We proposed a hardware/-software solution to implement an anomaly-based approach to detect software attacks against confidentiality and integrity using DIFT. HardBlare combines fine-grained DIFT with OS-level tagging which allows end-user to specify security policies. Main outcomes at the hardware level of HardBlare are the design of a dedicated multi-core DIFT coprocessor on FPGA that does not require any modification of the main CPU. This contribution tackles one of the main challenges of the project. Information required to bridge the semantic gap is obtained both at compile-time and runtime through pre-computation during the compilation step, using of hardware trace mechanisms, and instrumentation. Main outcome at the software level of HardBlare is the Linux kernel modification to handle file tags and to support communication between the

instrumented code and the coprocessor, and an LLVM backend pass to compute the annotations and instrument applications. All validation has been performed on the Digilent ZedBoard using Xilinx ZYNQ SoC which combines a dual-core ARM Cortex-A9 with a Xilinx FPGA. HardBlare demonstrates that a hardware/software co-design approach for DIFT is a promising solution.

The initial idea behind this work was presented at RESSI in 2017 [12], and the general approach was presented and published at FPL 2017 conference [11]. The design of the PTM trace decoder was presented at AsianHOST 2018 conference [6]. The DIFT coprocessor was presented and published at ReConFig 2018 conference [5].

This work was realized in the HardBlare project, funded by Labex CominLabs. HardBlare was a collaboration between the CIDRE and SCEE teams at CentraleSupélec and the Lab-STICC at the University of Southern Brittany. In this project, I co-advised the Ph.D. of Mounir Nasr Allah and the Ph.D. of Muhammad Abdul Wahab with my colleague Pascal Cotret from CentraleSupélec, who moved to Thales during the project. We also collaborated with Guy Gogniat, Vianney Lapôtre, and Arnab Kumar Biswas from the University of Southern Brittany.

Muhammad Abdul Wahab developed the hardware components of the project, including the PTM trace decoder and the DIFT coprocessor. Mounir Nasr Allah mainly realized the software stack development, including the LLVM static analysis and instrumentation pass, the integration with the Blare OS-level monitor, and the packaging in the Yocto Linux distribution. He is also the principal integrator of the complete hardware/software solution.

2.4 Perspectives

We consider the following perspectives for hardware-assisted intrusion detection.

Exploring different detection approaches In our works, we focused on ensuring CFI and DIFT. In the future, we want to explore other detection approaches to cover a larger spectrum of attacks. For example, Chen et al. [Che+05] 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 [CCH06] could be implemented using hardware-assisted detection.

Moreover, we would like to study different targets. In particular, we are interested in embedded systems whose limited resources represent an appealing challenge to solve.

In a collaboration with the Inria PACAP team, I am co-advising the Ph.D. of Nicolas Bellec with my colleagues Isabelle Puaut from PACAP and Frédéric Tronel from CIDRE. Nicolas Bellec started his Ph.D. in 2019 at the intersection between real-time systems and security. Real-time systems must ensure they complete their tasks before a fixed deadline, which requires evaluating their WCET (Worst Case Execution Time). However, attacks can also target these systems, which are often poorly protected against cyber threats. Nicolas proposed to implement a DFI (Data-Flow Integrity) mechanism and optimize it for real-time systems. He implemented the DFI protection using the Clang/LLVM compiler tool chain and the PhASAR static analysis tool. We could rely on hardware to protect or optimize the DFI.

Monitoring hybrid applications In our works, we assume that the monitored application is executed on a single CPU. However, an increasing number of applications offload part of their computation to hardware accelerators like GPU (Graphics Processing Unit) or FPGA.

The approach we explored cannot follow the information flows in hybrid applications, where part of the processing is deported to an FPGA. Some works proposed to verify the information flows of a circuit at the HDL (Hardware Description Language) level [Fer+17a; Ard+17; Ard+19]. Nevertheless, these works are exclusively interested in static verification of information flows within a hardware circuit.

We think that the most promising approach is to analyze and instrument the intermediate code of an HLS (High Level Synthesis) tool in order to insert a circuit-specific DIFT monitor [Pil+19; Jia+18]. However, these works mainly focus on the generation of hardware tag propagation circuits corresponding to functional blocks implemented on FPGA, and the authors do not consider the case where the DIFT of software components is also realized in hardware.

In the ANR TrustGW project, which will start in January 2022, we consider a system composed of IoT objects connected to a gateway. This gateway is, in turn, connected to one or more cloud servers. The architecture of the gateway, which is at the heart of the project, is heterogeneous (software-hardware), composed of a baseband processor, an application processor, and hardware accelerators implemented on an FPGA. In this project, we will explore, with my colleagues Frédéric Tronel and Pierre Wilke, hardware-assisted DIFT approaches for such hybrid applications.

Reducing the overhead of the DIFT Runtime overhead is still an important limitation of our harware-assisted DIFT approach. This overhead is mainly due to the instrumentation. We have to reduce the number of instrumentation points drastically to improve performance. We believe that using more advanced static analyses could reduce this overhead by inferring the application code's behavior at compile-time more precisely.

Chapter 3

Host-based intrusion survivability

3.1 Introduction to OS-based intrusion survivability

Intrusion detection is the first step in reactive security approaches. Once an intrusion has been detected, it is necessary to contain the attack, restore the infected system into a secure state, and prevent any reinfection. These last steps often require some manual intervention from a human. For example, security mechanisms like IDS usually report their alerts to SIEM (Security Information and Event Manager), which security operators manage in a SOC (Security Operations Center).

SOC are organizational units that "combine processes, technologies, and people to manage and enhance an organization's overall security posture" [Vie+20]. To achieve these goals, the SOC monitors the different components of the organization's information system to detect, report, and mitigates cyber threats. SOC can be operated internally by the organization or outsourced, at least partially, to MSSP (Managed Security Service Provider). People working in a SOC are generally organized in three tiers, supervised by a SOC manager. Security operators of Tier one are in charge of the collection, correlation, and triage of security events. They can also directly respond to these security events in simple cases. For more complex situations, the more critical security events are cascaded to Tier 2 or Tier 3 security analysts.

Management of security events can be automated in the different steps using, for example, misuse-based or anomaly-based detection, correlation rules, or SOAR (Security Orchestration, Automation, and Response). However, this process still mainly relies on human actions. Moreover, managing security events within a SOC supposes that the different devices of the information system are connected to the SOC or MSSP. However, there are still a lot of end-user devices which are not always connected to such tools. Consequently, there is a need to provide end-user devices with capabilities to detect and respond to intrusions autonomously without being connected to external services.

Automation is crucial to handle intrusions at the end-user device level, since we cannot expect this end-user to be a security expert. Moreover, it is also essential to maintain some availability of the data and services hosted by the end-user device. Many host-based intrusion detection approaches have been proposed in the literature, and some of them are actually implemented in existing tools such as antivirus and endpoint security suites. However, such tools provided limited functionalities to survive or withstand an intrusion once it has been detected.

Therefore, in the context of the Ph.D. of Ronny Chevalier, we addressed the following issue:

How to design an OS-level intrusion response system so that the OS services can survive intrusions?

This question poses several challenges:

1. How to recover the services in a sane state after an intrusion?
2. How to respond automatically to block intrusions and prevent any further reinfection?
3. How to maintain service availability during and after the reaction to intrusions?

3.1.1 Intrusion recovery and response

Intrusion recovery systems help administrators restore a compromised system into a sane state to limit the damage done by security incidents. However, most existing approaches do not preserve the availability [Goe+05; Kim+10; Hsu+06], e.g., they force a system shutdown. Moreover, they neither stop intrusions from reoccurring nor withstand reinfections [XJL09; Goe+05; Kim+10; Hsu+06; WEP18]. Even if the recovery mechanism restores the system into a sane state, vulnerabilities are still present, and nothing stops attackers from reinfecting the vulnerable system. Such a situation could lead to a loop of infections and recoveries.

Intrusion response systems apply countermeasures to stop an intrusion or limit its impact on the system [Foo+05]. However, existing approaches apply coarse-grained responses that affect the whole system and not just the compromised services [Foo+05]. For example, they can block a given TCP port for the whole system because a single compromised service uses this port maliciously. Moreover, their response selection process relies on a strong assumption of having complete knowledge of the vulnerabilities present on the infected system and exploited by the attacker [Foo+05; Sha+18].

Due to these limitations, existing approaches cannot respond to intrusions without affecting the availability of some system services. However, availability is crucial for safety-critical applications, but it is also essential for business continuity and user experience. For example, while websites, code repositories, or databases are not safety-critical, they can be essential for companies or user activities.

3.1.2 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" [Ell+97]. Avizienis et al. suggested that—based on this definition—dependability and survivability were similar concepts, where dependability is "the ability of a system to avoid service failures that are more frequent or more severe than is acceptable" [Avi+04]. Knight et al. 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, they defined survivability as the ability "to provide one or more alternate services (different, less dependable, or degraded) in a given operating environment" [KSS03] essentially providing "a trade-off between functionality and resources" [KSS03].

In the rest of this manuscript, 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 been applied to networked systems or critical information systems. In our case, however, we apply the concept of intrusion survivability to commodity OS (e.g., Linux-based distributions or Windows).

3.1.3 Our contribution

Our approach distinguishes itself from prior work on three fronts. First, we combine the restoration of the files and processes of a service with the ability to apply responses after the restoration to withstand 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 needed by the service.

We introduced this degraded mode on purpose. When the intrusion is detected, we assume that we do not have a patch available or precise information about the vulnerabilities to patch them. 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 achieving their malicious goals. Second, it keeps available as many service functions as possible, thus maintaining availability while waiting for a proper patch.

We maintain the availability by ensuring that at least the core functions of services are still operating. However, intrusion responses may degrade the availability of non-essential functions. For example, "providing read access to the website" could be the core function of a web server, whereas "logging accesses" could be considered 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.

In the following sections, we detail our contribution regarding OS-level intrusion recovery and response. Section 3.2 describes our approach, which relies on a cost-sensitive response selection. Our solution automatically selects a response that maximizes the effectiveness against the threat while minimizing its impact on the service availability. Then, Section 3.3 presents the results of the experimental evaluation of our approach.

3.2 Intrusion survivability approach at the OS-level

We propose a novel intrusion survivability approach to withstand ongoing intrusions at the OS level. Our approach relies on the orchestration of fine-grained recovery and per-service responses, such as privileges removal.

3.2.1 General overview of our approach

The main threat that we address is the compromise of services inside an OS. However, we assume the integrity of the platform's firmware (e.g., BIOS or UEFI-compliant firmware) and the OS kernel. Indeed, if attackers compromise such components at boot time or runtime, they could circumvent our recovery and reaction mechanisms. Such assumption is reasonable in recent firmware using a hardware-protected root of trust [Rua14; HP 19] at boot time and protection of firmware runtime services [YZ17; YZ15].

Moreover, we can ensure the integrity of the OS kernel at boot time by using UEFI Secure Boot [UEF19b]. We can also protect the kernel integrity at runtime by checking some

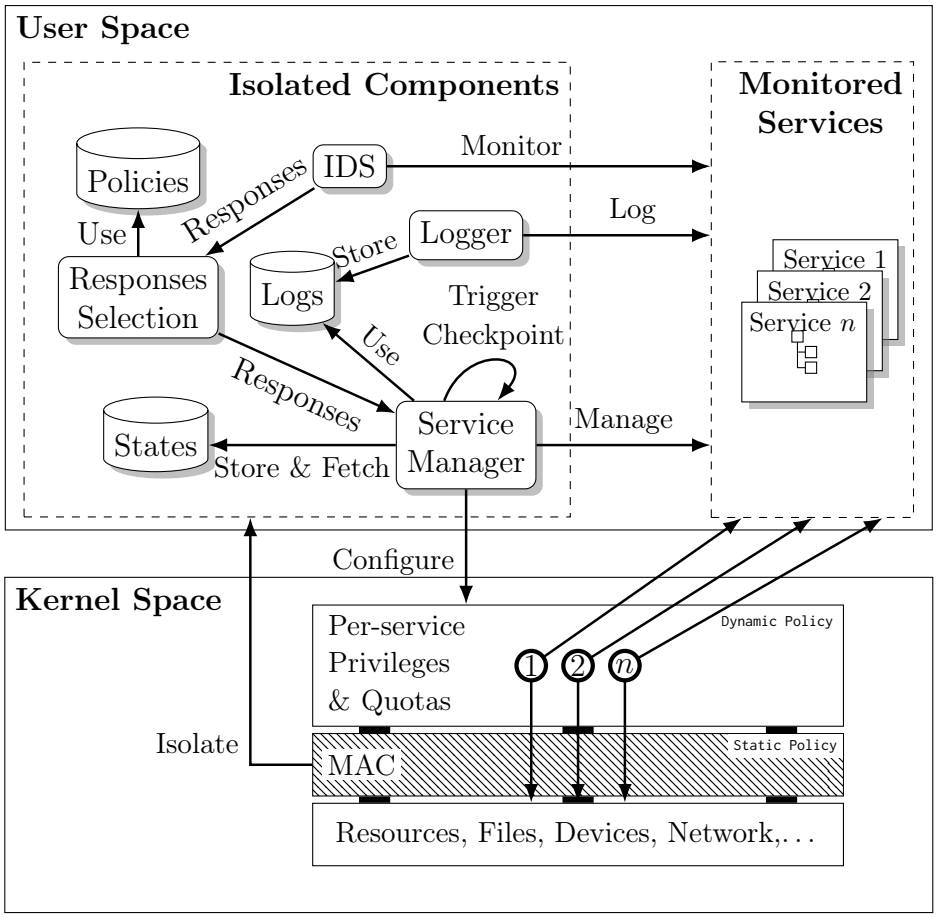


Figure 3.1: Overview of Survivor architecture

invariants within the kernel code [Son+16] or relying on a hardware-based integrity monitor [Aza+14].

Figure 3.1 gives a general overview of the architecture of our solution. We assume that some IDS monitor the system and raise alerts when they detect intrusions targeting the system services. We suppose that the services are managed by a *service manager*, such as systemd [Poe19] on Linux or Service Control Manager [RSI12] on Windows. We modified this service manager to takes periodic checkpoints of the services and snapshots of the file system. In addition, a *logger* trace 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.

Based on the information reported by the IDS, the *response selection* component selects the optimal response and sends this information to the *service manager*. Then, the service manager restores the infected service to the last known safe state, including all the files modified by the infected service. Finally, it configures kernel-enforced per-service privilege restrictions and quotas based on the selected responses. Since different processes can belong to the same service, we must be able to restore and apply restrictions to all the processes of a given service. In our implementation, we rely on Linux cgroups [Heo15] to bound processes to services. On Windows, job objects [Mic18a] provide similar features.

We also have to ensure the integrity of the different components of our solution executed in user-mode. Different solutions could be used, such as using a dedicated hardware isolated execution environment or a hypervisor. We propose to rely on a static MAC (Mandatory

Access Control) policy enforced by the OS kernel to isolate these components from monitored services and user applications. This solution does not require any modification of the hardware and software components and can be implemented using standard OS-level mechanisms such as SELinux [NR]. This policy must enforce the following restrictions:

- User applications and monitored services cannot stop the execution of isolated components;
- Only the isolated components have access to their dedicated storage (e.g., to store the logs or checkpoints) and configuration files;
- Attackers cannot impersonate the isolated components;
- Communications between isolated components are restricted (e.g., the logger cannot communicate with the response selection component).

The IDS internal design is out of the scope of this work, and we do not make any assumptions on the detection approach or the IDS localization. We could, for example, rely on traditional antivirus, endpoint security tools, or hardware-assisted intrusions detection approaches described in Chapter 2. However, we assume that the alerts give enough information to:

1. identify the services targeted by the intrusion;
2. map each alert to a set of malicious behaviors[1] and to a set of responses that can stop or mitigate them;
3. associate each alert to a confidence level[2].

Generic responses can be inferred from the type of intrusion if the IDS cannot report such information. Moreover, the administrator can map a confidence level to each IDS. In that case, all the alerts from that IDS will share the same confidence level.

All these data can be directly reported in the alerts or inferred from CTI (Cyber Threat Intelligence) provided by third parties, based on the information mentioned in the alert. For example, information about the alert, the responses, or malicious behaviors can be shared using standards such as STIX (Structured Threat Information eXpression) [Bar14] and MAEC [Kir+11; MITa].

3.2.2 Response selection

Different responses can be selected to mitigate a given malicious behavior. In our approach, we proposed an automatic cost-sensitive response selection process that maximizes the mitigation efficiency while minimizing the loss of availability on core functions. Figure 3.2 illustrates such a process.

In comparison to previous approaches [Foo+05; SCH14; Sha+18; Khe+10], we do not rely on vulnerability or attack graphs. Instead, 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

[1] For example, the malware capabilities [MITb] from MAEC (Malware Attribute Enumeration and Characterization) [Kir+11]

[2] For example, the IDMEF (Intrusion Detection Message Exchange Format) alert format defines the confidence class [DCF07]

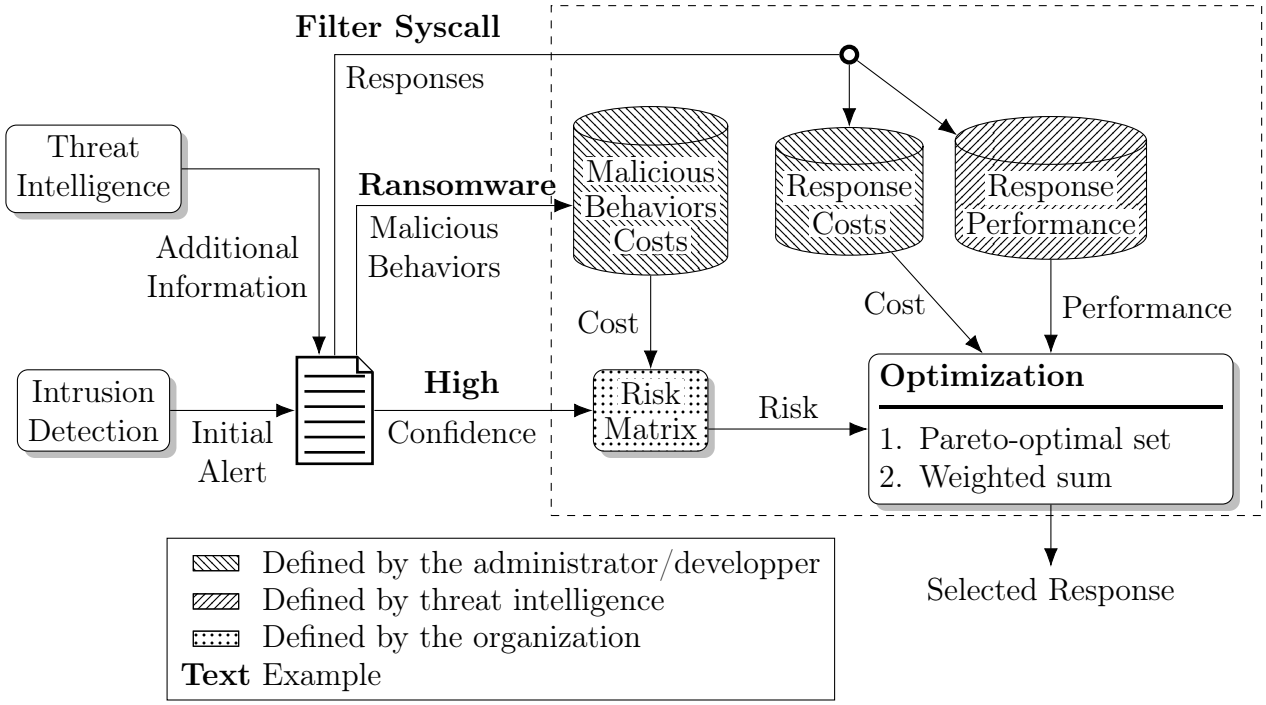


Figure 3.2: High-level overview of the response selection process

have information about the characteristics of the intrusions or the behaviors exhibited by the attacker.

Our selection process inputs are the confidence of the alert, the set of malicious behaviors associated with the reported intrusion, and the corresponding responses. The selection process also takes into account different models corresponding to the response selection policy:

- The *malicious behavior costs* model the impact cost of each malicious behavior on the availability, confidentiality, or integrity of the service;
- The *response selection costs* give the impact cost of each response on the availability of a given service;
- The *response performance* gives the effectiveness of each response to mitigate a given malicious behavior.
- The *risk matrix* defines the level of risk based on the likelihood and the cost of malicious behaviors.

The response performance only depends on the response and the malicious behavior. It depends neither on the service nor on the organization's policy. Thus, it can be directly inferred from CTI and shared between different organizations. However, it must be updated regularly to take into account the new type of malicious behaviors.

The risk matrix does not depend on the malicious behaviors, responses, or services. It is only based on the organization's policy and expresses the risk tolerance of this organization. Thus, it is not supposed to evolve frequently.

Malicious behavior costs and response costs are specific to each service. They also depend on the deployment context. They have to be defined by developers or administrators of the service and should be updated to consider new threats.

We formally define the response selection policy by a tuple $\langle c_r, p_r, c_{mb}, ri \rangle$, with c_r the function defining the response cost, p_r the function defining the response performance, c_{mb} the function defining the malicious behavior cost, and ri the function defining the risk matrix. We define more precisely those functions in the following sections.

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. In contrast, lower levels describe more specific behaviors, e.g., encrypt files. For example, we can rely on the malware capabilities hierarchy [MITb] from the MAEC project [Kir+11]. We can also use the MITRE ATT&CK knowledge base [MIT19], but it does not directly provide a hierarchy of malicious behaviors.

Figure 3.3 illustrates an example of a non-exhaustive malicious behavior hierarchy with behaviors that relates to availability violations and to a C&C (Command and Control).

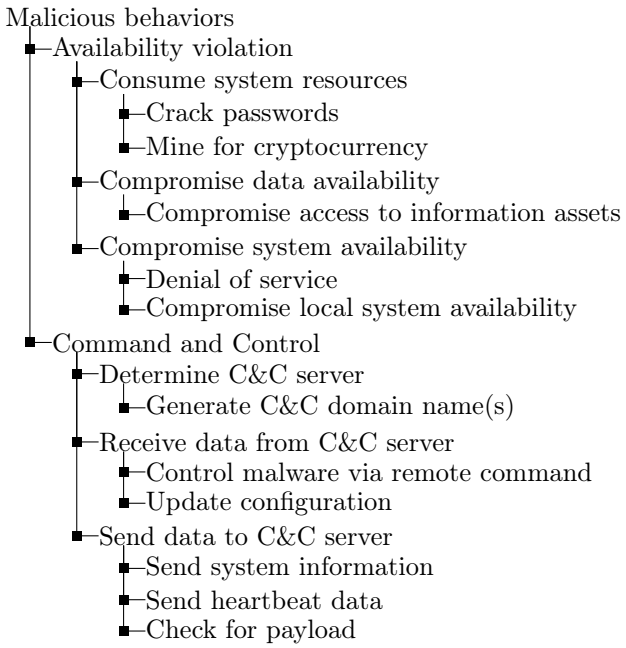


Figure 3.3: Example of a non-exhaustive malicious behavior hierarchy

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 specific behavior than m'. Let $\mathbf{I}$ be the space of intrusions reported by the IDS. We assume that for each intrusion $i \in \mathbf{I}$, 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 $m \in M^i \implies m' \in M^i$. Thus, if an alert associates a malicious behavior to an intrusion, we also associate all the less specific behaviors of the same type to this intrusion, even if they are not explicitly mentioned in the alert. For example, if the alert maps the "Mine of cryptocurrency" malicious behavior to an intrusion, we also map the "Consume system resources" and "Availability violation" behaviors to this intrusion.

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). Such responses rely on OS-level privilege and quota restriction mechanisms available in existing systems. 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 $r \in R^m \implies r' \in R^m$. As with malicious behaviors, if a fine-grained response is mapped to an intrusion, we also map all the more coarse-grained responses of the same type.

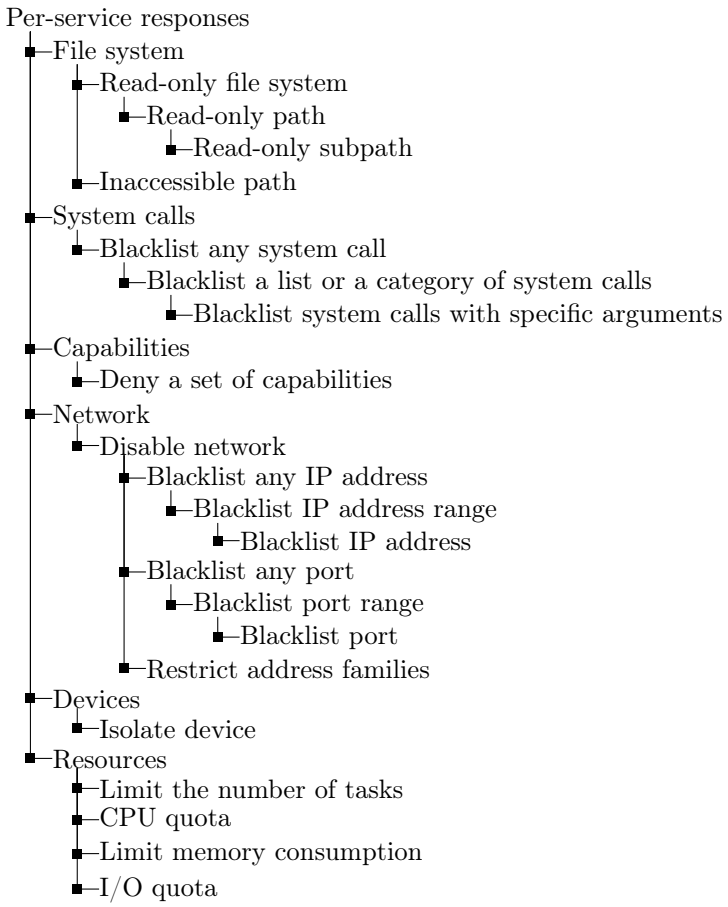


Figure 3.4: Example of a non-exhaustive per-service response hierarchy

Figure 3.4 is an example of such a response hierarchy. Notice that some responses require parameters, such as remounting a path as read-only or blocking a set of IP addresses. In that case, the hierarchy also includes sub-responses corresponding to the same response but applied on a subset of the response parameter. For example, remounting `/var/www` as read-only is a sub-response of remounting the whole `/var` path as read-only.

Malicious Behavior Cost and Response Cost

Let $\mathbf{S}$ be the set of all services and let $\mathbf{Q}$ be a totally ordered set of qualitative linguistic constants composed as follows: none < very low < low < moderate < high < very high < critical.

We require administrators to set a malicious behavior cost for at least each malicious behavior of the first level of the hierarchy. Administrators can specify this mapping in a dedicated

configuration file associated with each service. We believe that this constraint is reasonable since there is a limited number of first-level behaviors. For example, there are only 20 elements on the first level of the MAEC hierarchy. Administrators can also set costs to more specific behaviors, which is not required but make the response selection process more accurate.

We define $c_{mb}\colon \mathbf{S} \times \mathbf{M} \to \mathbf{Q}$, the function that takes a service, a malicious behavior, and returns the associated cost. Since administrators do not have to set a cost for all the malicious behaviors of $\mathbf{M}$, the function c_{mb} directly derived from the configuration file can be a partial function. We proposed to extend this function to the total function $c^e_{mb}\colon \mathbf{S} \times \mathbf{M} \to \mathbf{Q}$ using the following approach:

$$c_{mb}(s,m) \neq undef \implies c^e_{mb}(s,m) \triangleq c_{mb}(s,m)$$

$$\left.\begin{array}{c} c_{mb}(s,m) = undef \\ \wedge \\ c_{mb}(s,m') \neq undef \\ \wedge \\ m \prec_{\mathbf{M}} m' \\ \wedge \\ \forall m'', m < m'' < m' \implies c_{mb}(s,m'') = undef \end{array}\right\} \implies c^e_{mb}(s,m) \triangleq c_{mb}(s,m')$$

In each branch of the malicious behavior hierarchy, we identify the most specific behavior for which the c_{mb} function is defined. Then, we assign the cost of this behavior to all more specific behaviors in the branch.

Response costs allow service administrators or developers to specify how a response would impact the availability of the 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 critical value, we consider that a response would disable a core function of the service and thus should never be applied.

We define $c_r : \mathbf{S} \times \mathbf{R} \to \mathbf{Q}$, the function that takes a service, a response, and returns the associated response cost. Similarly to c_{mb}, this function may be partial, and we extend it the same way.

Response Performance

Our approach also takes into account the performance of a response to mitigate a given malicious behavior. Of course, the most effective response in terms of performance would be to stop the infected service. While our model allows it, we focus our work on fine-grained responses to maintain the availability of the core functions of the service.

We define $p_r\colon \mathbf{R} \times \mathbf{M} \to \mathbf{Q}$, the function that takes a response, a malicious behavior, and returns the associated performance.

Unlike the costs of malicious behaviors and responses, such a value depends only on the malicious behavior. It should be provided by security experts that analyzed similar intrusions. For

example, threat intelligence sources shared using STIX [Bar14] could produce such information.[3] More precisely, STIX defines a property called "efficacy" in its "course-of-action" object, representing response performance.

Risk Matrix

We rely on the definition of a risk matrix that satisfies the axioms proposed by Anthony Tony Cox [Ant08] 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 risks qualified as low.

Betweenness A slight increase in confidence or 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 organization's risk attitude: whether the organization is risk-averse, risk-neutral, or risk-seeking.

We define $ri\colon \mathbf{Q} \times \mathbf{Q} \to \mathbf{Q}$, the function representing the risk matrix that takes a malicious behavior cost, an intrusion confidence, and returns the associated risk.

3.2.3 Optimal response selection

We rely on known MOO (Multi-Objective Optimization) methods [MA04] to select the most cost-effective response. Our goal is to maximize the performance of the response while minimizing its availability cost to the service.

We select a response to protect the service from a malicious behavior notified by the IDS. We iterate the selection process on each malicious behavior $m \in M^i$ notified by the IDS. Thus, the service and the malicious behavior are fixed parameters of the selection algorithm. In the following, we omit these parameters in the response performance $p_r(r)$, the response cost $c_r(r)$, and the malicious behavior cost c_{mb}, for the conciseness of the notations.

From the information provided by the IDS and CTI sources, we build the set of response R^m that can mitigate the malicious behavior m. Before selecting an optimal response, we filter out from R^m any responses that have a critical response cost. Indeed, such responses would impact the availability of core functions of the service. Let $\hat{R}^m \subseteq R^m$ be the resulting set of this filtering:

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

Then, we select an optimal response for the malicious behavior m, by computing the Pareto-optimal set from $\hat{R}^m$. A solution is Pareto-optimal if it is impossible to find other solutions

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

that improve one objective without weakening another. The set of all Pareto-optimal solutions is called a Pareto-optimal set.[4] 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:

$$[p_r(r) > p_r(r') \wedge c_r(r) \leq c_r(r')] \vee [p_r(r) \geq p_r(r') \wedge c_r(r) < c_r(r')]$$

Let O_m be the resulting Pareto-optimal set:

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

We now have to specify preferences among the optimization criteria to select a solution in the Pareto-optimal set. For example, should we prioritize the performance of the response or reduce its cost on the availability of the service? We use MOO methods [MA04] for this last step. Such approaches rely on scalarization to convert a MOO problem into an SOO (Single-Objective Optimization) one.

The weighted sum method is a popular scalarization approach. It assigns a weight to each objective and computes the sum of the weighted objectives. However, this method does not always give solutions in the Pareto-optimal set [MA04]. Thus, we follow the approach of Shameli-Sendi et al. [Sha+18] that applies the weighted sum method on the Pareto-optimal set instead of on the whole solution space.

Before selecting a response from the Pareto-optimal set using the weighted sum method, we need to set weights and convert the linguistic constants into numerical values. We rely on a function l that maps the linguistic constants to a numerical value[5] between 0 and 1. Table 3.1 gives the mapping we adopted during our experiments.

Table 3.1: Mapping of linguistic constants to numerical values

Critical	Very High	High	Moderate	Low	Very Low	None
1	0.9	0.7	0.5	0.3	0.1	0

We want to prioritize the performance if the risk is high and the cost of the response if the risk is low. Thus, we use the risk matrix to set the weights as follows:

$$w_p = l\big(ri(c_{mb}, cf_a)\big), \text{ with } cf_a \in \mathbf{Q} \text{ the confidence of the alert}$$

$$w_c = 1 - w_p$$

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

$$\arg\max_{r \in O_n} w_p \cdot l\big(p_r(r)\big) + w_c \cdot \Big(1 - l\big(c_r(r)\big)\Big)$$

[4] Also known as a Pareto front.

[5] An alternative would be to use fuzzy logic to reflect the uncertainty regarding the risk assessment from experts when using linguistic constants [Den+11].

3.3 Evaluation and results

Ronny Chevalier developed a Linux-based prototype during his Ph.D., in collaboration with HP Labs at Bristol. We then use this prototype to evaluate the effectiveness of the response selection process and its impact on service availability and performance.

Throughout the experiments, we tested our implementation on different types of services: web servers (nginx [ngi] and Apache [The]), database (mariadb [Mar]), work queue (beanstalkd [Rar]), message queue (mosquitto [Ecl19]), and git hosting services (gitea [The19]).

3.3.1 Implementation

While our implementation relies on Linux features such as namespaces [Ker13], seccomp [Cor09], or cgroups [Heo15], our approach does not depend on OS-specific paradigms. For example, on Windows, one could use Integrity Mechanism [Mic18d], Restricted Tokens [Mic18c], and Job Objects [Mic18a].

At the time of writing, the most common service manager on Linux-based systems is systemd [Poe19]. We modified it to checkpoint and to restore services using CRIU [CRI19] and snapper [SUS18]. Our modified version of systemd also applies responses at the end of the restoration.

CRIU can checkpoint the state of Linux applications by fetching this state from different kernel APIs. Then, the CRIU store the checkpoint inside an image file. Finally, CRIU reuses this image and relies on other kernel APIs to restore the application. We chose CRIU because it allows us to transparently checkpoint and restore services, i.e., without modifying or recompiling them.

Snapper provides an abstraction for snapshotting different Linux filesystems. Moreover, it can compare different snapshots, including the current state of the filesystem, to track file modifications. In our implementation, we chose BTRFS [RBM13] since this filesystem provides a COW (Copy-On-Write) snapshot and comparison features, allowing fast snapshots and comparisons of the filesystem.

When checkpointing a service, we first freeze its cgroup to avoid inconsistencies, i.e., we remove the processes from the scheduling queue. Thus, it cannot interact with other processes nor with the filesystem. Second, we take a snapshot of the filesystem and a snapshot of the service's metadata kept by systemd, e.g., status information. 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 service metadata and ask Snapper to create a read-only snapshot of the current state of the filesystem. Then, we ask Snapper to compare 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 recover only the files modified by the monitored service, we filter the result based on our log of files modified by this specific service.

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 of the restoration process to avoid interfering with CRIU that requires certain privileges to restore processes.

The Linux auditing system [Jah+17] is a standard way to trigger events from the kernel to userspace based on a set of rules. For example, it can trigger events when a process performs write accesses on the filesystem. 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 the Linux audit system 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 the files modified by the monitored service. 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 an empty file `/private/abc.service/a/b/c/test`. This solution allows us to log modified files without keeping a data structure in memory.

We rely on Linux features such as namespaces, seccomp, or cgroups, to apply responses. Here is a non-exhaustive list of responses that we support: filesystem constraints (e.g., mounting all or any part of the filesystem read-only), system call filters (e.g., blocking a list or a category of system calls), network socket filters (e.g., denying access to a specific IP address), or resource constraints (e.g., applying CPU quotas or limiting memory consumption).

3.3.2 Security evaluation

For the experiments, we installed Fedora Server 28 with 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.

While malware could use anti-virtualization techniques [Pal+09; Che+08b], to the best of our knowledge, none of our samples used such techniques.[6]

Our first experiments focus on evaluating the effectiveness of the responses we proposed against different types of intrusions. In this first step, we do not evaluate the response selection process. We assume that the IDS detects the intrusion and reports the correct malicious behaviors corresponding to each intrusion.

We selected a list of attacks covering different types of malicious behavior rather than covering all the different malware families since we do not focus on detecting intrusions but on recovering from and withstanding them.

The following list describes the malware and attacks we used:

Linux.BitCoinMiner is a cryptocurrency mining malware that connects to a mining pool using attackers-controlled credentials [Tre18].

Linux.Rex.1 malware joins a P2P (Peer-to-peer) 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 (Distributed Denial-of-Service) attack, or send spam [Dr 16].

Hakai malware 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 [Nig18; Dr 18].

[6] This is consistent with the study of Cozzi et al. [Coz+18] 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.

Linux.Encoder.1 ransomware encrypts files commonly found on Linux servers (e.g., configuration 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 [Dr 15].

GoAhead exploit gives remote code execution to an attacker on all versions of the GoAhead embedded web server prior to `3.6.5` [Hod17].

We start a vulnerable service for each experiment, checkpoint its state, infect it, and wait for the payload to execute. 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 checkpoint. Finally, we consider the responses effective if we cannot reinfect the service or the payload cannot achieve its goals anymore.

Table 3.2: Summary of the experiments that evaluate the effectiveness of the responses against various malicious behaviors

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 filesystem
GoAhead exploit	Exfiltrate via network	Forbid connect syscall
GoAhead exploit	Data theft	Render paths inaccessible

Table 3.2 summarizes the results we obtained. As expected, in each experiment, our solution successfully restored the service after the intrusion to a previous safe state. In addition, as expected, each response was able to withstand reinfection for its associated malicious behavior and only impacted the specific service and not the rest of the system.

3.3.3 Performance evaluation

We evaluate the overhead of our solution due to the checkpoint, restore and monitor processes.

Each time we checkpoint a service, we freeze its processes. As a result, a user might notice slower responsiveness from the service. Hence, we measured the time to checkpoint different services: Apache HTTP (Hypertext Transfer Protocol) 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. On average, checkpointing always takes less than 300 ms.

In Figure 3.5, we illustrate the results of the availability cost that users could perceive by measuring the latencies of HTTP requests sent to an nginx server. We generated 100 requests per second for 20 seconds with the HTTP load testing tool Vegeta [Sen]. 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

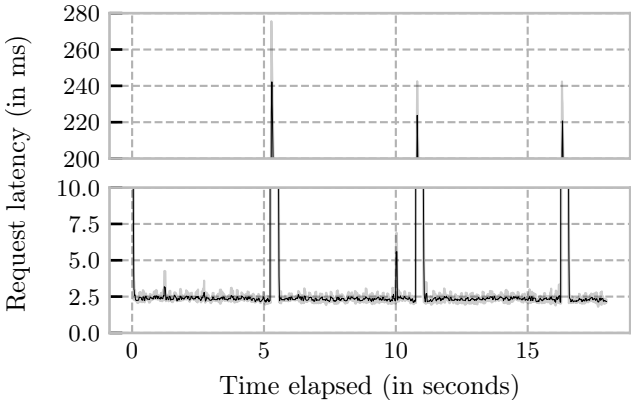


Figure 3.5: Impact of checkpoints on the latency of HTTP requests made to an nginx server (less is better)

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, and we only increase the requests' latency when the service is frozen. Since the latency increases only during a short time (maximum 300 ms), we consider such a cost acceptable. In comparison, SHELF [XJL09] incurs a 7.6 % latency overhead for Apache during the whole execution of the system.

In contrast to the checkpoint, the restore procedure loses all network connections since we kill the processes before restoring them. The experiments, however, show that the time to restore services is short (less than 325 ms). For example, in comparison, CRIU-MR [WEP18] took 2.8 s on average to complete their restoration process.

Finally, we evaluated the overhead of the file monitoring by running synthetic and real-world workload benchmarks. Indeed, our solution logs the path of any file modified by a monitored service.

We first ran synthetic I/O benchmarks from the Phoronix test suite [LT]. Such benchmarks stress the system by performing many open, read, and write system calls. We only notice a small overhead when the service is not monitored (between 0.6 % and 4.5 %). Moreover, we do not observe any noticeable overhead (less than 1 %, within the margin of error) for the benchmarks that only read files. However, with `fs-mark`, a benchmark that creates many files and directories, we observe a 7.3 % overhead when 1000 files are written synchronously. In comparison, there is a 27.3 % overhead when the files are written asynchronously. With `Postmark`, a benchmark emulating an email server, we observe that the overhead is significant (28.7 %) when writing 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 bursts.

The real-world workload benchmarks confirm those results. We measure the time to extract the archive of the Linux kernel source code and the time to compile the Linux kernel. We observe a 23.7 % overhead for the archive extraction when the service is monitored but only a small overhead of 1.1 % for the kernel compilation. In comparison, SHELF [XJL09] has a 65 % overhead when extracting the archive and an 8 % overhead when building the kernel.

In conclusion, the results show that our solution is more suitable for workloads that do not write many small files asynchronously in bursts. For instance, our approach would be more suitable for protecting web, databases, or video encoding services.

3.4 Conclusion and perspectives

This chapter presents my contribution to intrusion survivability for commodity OS. In contrast to other intrusion recovery approaches, our solution not only restores files or processes, but it also applies responses to withstand 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. 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 bursts.

The initial idea behind this work was presented at RESSI in 2018 [7]. The main contribution was presented and published at ACSAC 2019 conference [2]. An extended version of the article was published in the ACM DTRAP journal [30]. This work was done during the Ph.D. of Ronny Chevalier, in the context of a bilateral collaboration project with HP Labs at Bristol. Ronny realized the development of the prototype and contributed to different open-source Linux projects. However, the prototype itself is the property of HP and has not been released in open source. This work has also led to the publication of a patent (US10896085B2) and a patent application pending (EP3647980A1).

We consider the following perspectives for OS-level intrusion survivability.

Deploying fine-grained countermeasures inside the service Our approach can only apply responses at the OS level, limiting the type and the scope of the responses. Thus we can only filter system calls and cannot prevent the execution of some internal functions of the service or filter the parameters of these functions. However, preventing or restricting the execution of a known vulnerable function is appealing. From a more general point of view, we cannot modify the internal behavior of the service. To overcome this limitation, we need to modify the code of the service. Nevertheless, modifying the binary code of the service at runtime while maintaining the availability of the service poses a significant challenge.

In collaboration with the Inria PACAP team, we are working on an adaptive approach to instrument the code of an application at runtime to apply security protections. In this project, I am co-supervising the Ph.D. of Camille Le Bon with Erven Rohou from the PACAP team and Frédéric Tronel from the CIDRE team. Our DBI (Dynamic Binary Instrumentation) approach, published at SILM 2021 [1], can deploy protections on an executing application and uses runtime information to optimize those protections during the process execution. We implemented a coarse-grain CDI (Control-Data Isolation) protection using this framework and evaluated its impacts on the performance.

We could use the framework developed by Camille in our intrusion recovery and response system. Using DBI would help us to apply more fine-grained responses directly inside the application code.

Dynamic adaptation of the countermeasures In the future, we would like to investigate how we could automatically adapt the system to remove gradually the responses that we applied to withstand reinfection. The objective would be to leave the degraded mode and move towards operating modes increasingly close to the nominal function mode.

However, to prevent exploitation of existing vulnerabilities while maintaining the availability of all service functions, such an approach must apply fine-grained and precise countermeasures. Ideally, we should be able to patch the service automatically [Dua+19; Xu+20]. Nevertheless, one of the main challenges is being able to identify where and what the vulnerabilities are. Moreover, live patching implies modifying code at runtime without impacting the service availability. Such modifications would require a DBI approach similar to the one mentioned in the previous perspective.

Proposing recovery and response mechanisms for other types of software components We developed an OS-level approach to recover OS services and respond to intrusions targeting such OS services. We want to explore how we could apply similar approaches to other types of software components. A first step would be to study user applications, which a service manager like systemd does not directly manage. Since our approach heavily relies on such service managers, we will have to develop another component to snapshot, restore and restrict the behavior of user applications. We also have to isolate all the processes and data belonging to this application from the process and data of other applications of the same user. A possible solution would be to use containers [Vie].

We are also interested in porting our solution to more privileged components, like OS kernel, hypervisors, or UEFI firmware. The challenge, in this case, would be to rely on a more privileged component to apply the recovery and intrusion responses at runtime. For example, current state-of-the-art solutions can restore the BIOS at boot time in a safe state if it has been compromised or corrupted unintentionally [HP 19]. However, to the best of our knowledge, no solution exists to recover the state of the SMM at runtime without impacting the user experience.

Chapter 4

Formal specification and verification of hardware-assisted security mechanisms

4.1 Introduction to formal specification of hardware-assisted mechanisms

Using hardware components to implement security mechanisms is appealing. However, hardware platforms are increasingly complex and involve interactions between different hardware components. Figure 4.1 illustrates such complexity for a typical Intel architecture (generation *Nehalem* [Tho11] and newer). The core of this platform is the CPU, which is directly connected to high-speed peripheral devices such as the DRAM memory and the display controller. Th CPU is also connected to the so-called PCH (Platform Controller Hub), which is often located on the same package as the CPU. This component corresponds to the chipset that handles the communications with other peripherals like SSD (Solid-State Drive), hard drives, or sound cards.

Currently, manufacturers tend more and more to integrate these different components on the same chip, adopting an SoC-type architecture. This is particularly the case in embedded systems and smartphones, but traditional PC architecture is also following this trend. Thus, the CPU chip embeds more and more components. It is now a complex system, as shown in Figure 4.2, which includes:

- several execution cores,
- the GPU,
- the memory controller,
- the display controller.

The tendency is to integrate within the same chip components that were previously located within the chipset or in external devices. For example, the memory controller was located in the chipset in previous x86 architectures. Recently, Intel has even integrated a Thunderbolt controller directly on the CPU chip, which allows connecting external devices directly to the CPU without going through the PCH. All these components that were previously located in an external chipset correspond to the System Agent in Intel recent architecture. These developments have been driven by performance, form-factor, and power consumption optimization.

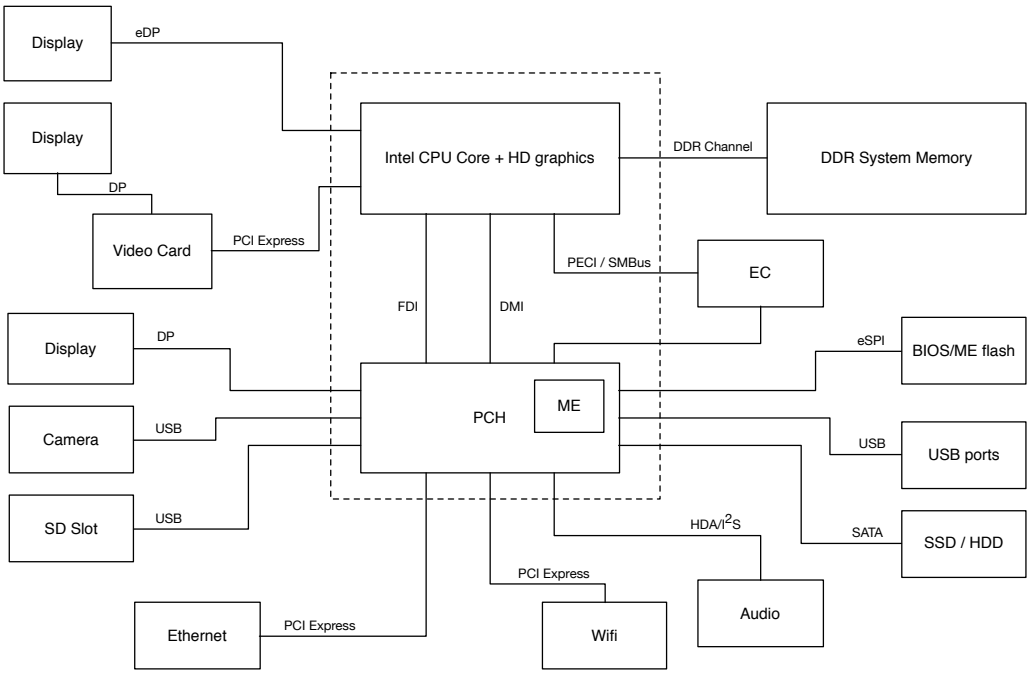


Figure 4.1: Intel x86 architecture

However, they come at the price of increasing complexity.

The internal microarchitecture of CPU cores, which executes software instructions, are also increasingly complex and implement many optimizations. On a modern x86 CPU, instructions are decoded by a front-end unit and translated into one or several microinstructions. This translation is performed by the microcode stored inside an internal ROM (Read-Only Memory) but can be patched at runtime. Those microinstructions are then executed in parallel by several execution units of a RISC-based execution engine. The microarchitecture implements many optimizations, such as multithreading [Mar+02], instruction pipelining [Fog12], out-of-order execution [Fog12] or predictive branching [MMK02]. Memory accesses are also optimized by different cache levels.

CPU cores are not the only components of the platform that execute software and perform memory accesses. The different peripheral devices generally embed one or several dedicated CPU or microcontrollers executing firmware. These devices also embed some dedicated memory that can be exposed to the main CPU cores. Indeed, memory-mapped I/O is now the privileged mechanism to perform I/O in x86-64 architecture. Thus, a simple `movb $0xFF, (%ebx)` instruction can have different effects depending on the physical destination address stored inside the `ebx` register: the byte `0xFF` can be written in the DRAM of the platform, in an external memory (e.g., the video memory of an external graphical card) or a register exposed by a peripheral device. In this last case, the writing can cause a specific effect implemented by the device, e.g., the modification of the output voltage of a GPIO (General Purpose Input/Output). Notice that software can modify the physical memory mapping using different registers located inside the System Agent and the PCH. Conversely, peripheral devices can often directly read or modify the values stored inside the DRAM of the platform, using DMA (Direct Memory Access).

It is thus difficult to accurately predict the evolution of the platform state because of the complex interactions between the different software and hardware components of this platform. For example, the semantic of the `movb $0xFF, (%ebx)` instruction highly depends on the state

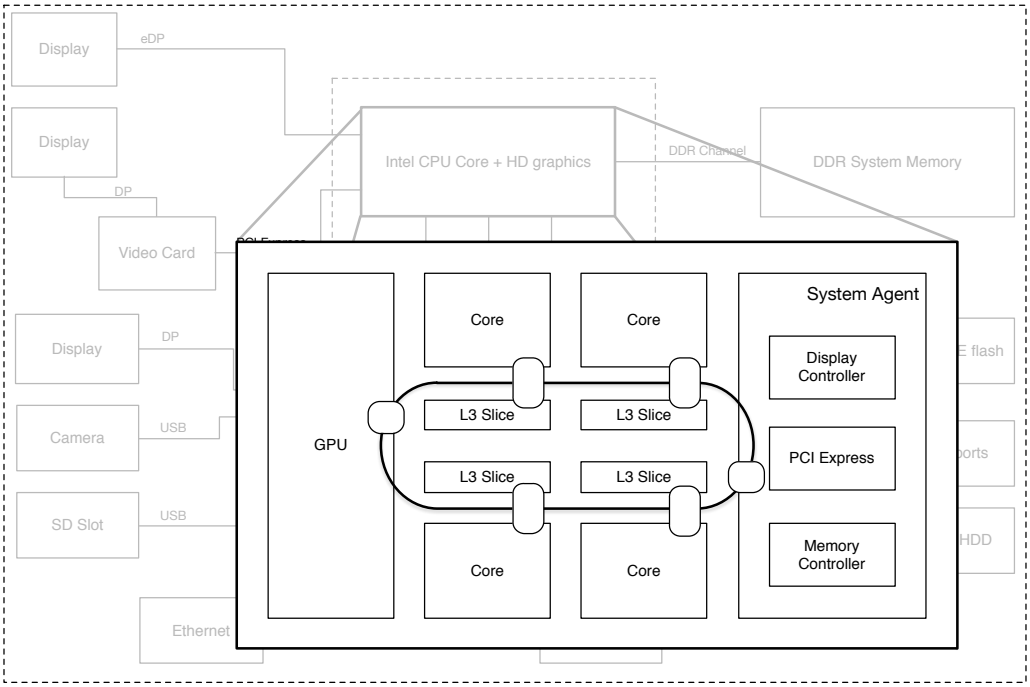


Figure 4.2: Intel CPU die architecture

of the DRAM and of different registers located inside the CPU and the chipset. First, most programs, including OS kernel and user applications, are now executed in long mode. In this mode, paging is mandatory, and the value of `ebx` is a virtual address that has to be translated to a physical address by the MMU (Memory Management Unit). If such a translation has been done recently, the result may be available in one of the TLB (Translation Lookaside Buffer) caches located inside the core. Otherwise, the MMU has to perform a page walk, using the value of the `CR3` register, to fetch the page table, a data structure located in the memory. The page walk implies that the MMU reads the memory at different addresses to get the data needed for address translation. Such data can reside in the different levels of caches available on the CPU (cache hit). In that case, there is no access to the DRAM, and the value is directly retrieved from the cache. Otherwise, the CPU issues a memory transaction that can be routed to the main DRAM, to a peripheral device directly connected to the CPU or to the PCH.

This process is highly influenced by the value of configuration registers, such as `CR3` or TOLUD (Top of Low Usable DRAM). Indeed, this last register influences the routing of memory transactions, since every physical address between the value of this register and 4 GB is routed to PCI (Peripheral Component Interconnect) devices. The value of these registers depends on the instructions executed before. Additionally, these instructions may have changed the values of structures stored in memory, such as the page table used during the page walk. They also influence the behavior of the caches, depending on the memory accesses made and the cache policy. The latter can be set via MTRR (Memory Type Range Registers) registers or PAT (Page Attribute Table), a set of bits located in each page table entry. Furthermore, peripheral devices and instructions executed on other CPU cores can have modified the value of the DRAM content and some registers. Finally, the execution of the instruction can be interrupted, e.g., by page fault.

4.1.1 Hardware-based Security Enforcement Mechanisms

This complexity can be the source of bugs in the various hardware and software components. In terms of security, this can lead to vulnerabilities that attackers can exploit. Indeed, the different software and hardware components of the platform can be malicious or vulnerable to attacks. If one of these components is compromised, the platform offers different mechanisms to protect other components and ensure their integrity, confidentiality, and availability.

Such mechanisms often rely on hardware features and a hierarchy of software components. A more privileged software component configures hardware features to control the execution of less privileged components and enforce a given security policy. We call HSE (Hardware-based Security Enforcement) such a class of security mechanisms.

For example, OS kernels rely on x86 privilege levels and paging mechanisms to isolate user applications, preventing them from interfering with the execution of other applications and the kernel itself. In the same way, hypervisors use virtualization extensions to isolate different VM (Virtual Machine) and the UEFI firmware relies on SMM to isolate some critical runtime services from OS and applications, as explained in Section 2.2.

However, the complexity of the hardware platform leads us to the following question: can we trust these HSE mechanisms? This question poses several challenges:

1. What are precisely the security guarantees the HSE is supposed to offer, i.e., what is the definition of the security policy?
2. How does the trusted software have to configure the HSE mechanism's hardware features to enforce this security policy?
3. What is the hardware specification, and will a correct implementation effectively enforce the security property?
4. Can we ensure that software and hardware implementations are correct regarding this specification?

The public specifications of the x86 hardware platform are published in informal textual descriptions. Such descriptions are voluminous and sometimes ambiguous. For example, the Intel 64 and IA-32 Architectures Software Developer's Manual [Int21] is 4,778 pages long, and the two volumes of the Platform Controller Hub datasheet [Int20] represent 1659 pages. The detailed description of the semantics of some mechanisms are dispersed in several documents, and the security policies that are supposed to be enforced are rarely formally stated.

In practice, developers of trusted software components do not always configure HSE hardware mechanisms correctly, leading to vulnerabilities [Bul+14]. In addition, the hardware mechanisms may themselves also suffer from vulnerabilities. In particular, the interactions between different hardware components can lead to vulnerabilities that attackers can exploit, even if the trusted software components comply with the specifications. Hardware designers can introduce such vulnerabilities by adding or modifying hardware components whose behavior can interfere with an HSE. For example, early versions of Intel TXT [Int15] disabled hardware interrupts that were otherwise used to protect BIOS flash memory [Kov+15]. Similarly, the SMM cache poisoning attack [Loi+09] exploits cache edge effects on the SMM isolation mechanism.

Such vulnerabilities are related to errors or lack of clarity in the specifications. Therefore, it is essential to formalize these specifications and ensure that the interactions between the different components guarantee the expected security policies.

4.1.2 Using formal methods to reason about HSE mechanisms.

Recent works have proposed to formalize processor semantics. For example, ARM has developed an executable formal specification of its processors to perform formal verification by model-checking and testing [Rei16]. ARM has applied this approach to several of its microprocessors [Rei+16], and this formal specification is available on the manufacturer's website [Lim21].

Some work has also tried to formalize the semantic of x86 processors [Das+19], but they only model the user part, which is still a significant work, given the complexity of this instruction set. However, security mechanisms relying on privileged instructions cannot be verified.

The SAIL project at the University of Cambridge has developed a framework and a specification language to specify different architectures. This formal specification can be used in various formal testing and validation tools [Arm+19]. The framework can also automatically generate a textual description of the architecture. The authors used this framework to formally model the ISA of ARM (in collaboration with the manufacturer), RISC-V, and CHERI processors. Moreover, the RISC-V Foundation has officially adopted this language for the formal specification of their RISC-V architecture.

One of the challenges is to ensure that the hardware implementation, whether it is an ASIC or a softcore implemented on FPGA, respects this formal specification. Chlipala *et al.* [Cho+17] have proposed the Kami hardware description language, which allows designing hardware components (like VHDL or Verilog languages), but also to prove (thanks to the Coq language) the equivalence between such components. Chlipala *et al.* have proposed, as an application of their language, an implementation of a part of a RISC-V processor. More recently, the authors of Kami have proposed Kôika [Bou+20], a verified compiler of an HDL close to Bluespec to RTL circuits.

The SiFive company[1], which proposes, among other things, implementations of RISC-V processors, has taken over the formalization of Kami. A more complex implementation of a RISC-V processor in Kami is under development [SiF]. However, the project does not seem to be active anymore.

All these works mainly focus on the functional specification of the processors and do not focus on the security mechanisms and the security properties they are supposed to enforce.

Ferraiuolo [Fer+17b] proposed a strongly typed HDL for specifying and proving isolation mechanisms using information flow verification and static type checking analysis [Fer+17b]. They applied their approach by implementing a mechanism similar to TrustZone on a MIPS softcore. Other researchers have proposed a similar approach based on Chisel, a high-level HDL used in particular to implement RISC-V [Den+19] processors. They verify that the information flow policy is respected using a solver.

Recently, researchers involved in the CHERI and SAIL projects have used the SAIL language to specify the CHERI architecture and prove that it guarantees some security properties, including monotonic evolution of capabilities (an attacker cannot modify a capability to make it more permissive) [Nie+20]. SAIL can also be used to test an implementation to ensure that it conforms to the specification. However, the project does not currently allow formal verification that an implementation respects the semantics of the formal specification.

[1] `https://www.sifive.com/`

However, these works focus on hardware security mechanisms that are transparent to the executed software and do not have to be configured by some trusted software. Moreover, they mainly focus on the internal design of the CPU. Verifying HSE mechanisms is more challenging since it requires considering both the software and hardware part of the specification. Moreover, it is essential to analyze not only the CPU but also the different hardware components of the platform.

4.1.3 Our contribution

In the context of Thomas Letan's Ph.D., we developed SpecCert, a framework for specifying and verifying HSE mechanisms against hardware architecture models. SpecCert relies on a three-step methodology. First, we model the hardware architecture specifications. Then, we specify the software requirements that the trusted software components must satisfy to configure the HSE mechanism. Finally, we prove that the HSE mechanism is sound under the assumption that the software components comply with the specified requirements. Such a proof implies that the hardware involved in the HSE mechanism indeed provides the security properties they promise. We believe this approach to be beneficial to both hardware designers and software developers. The former can verify the soundness of the hardware part of HSE specification. The latter can get unambiguous specifications in the form of a list of requirements that trusted software must comply with and the provided security properties.

In the following sections, we detail our contribution regarding the formal specification and verification of HSE mechanisms. Section 4.2 is a general overview of the SpecCert framework. Section 4.2.3 presents the formal definition of code injection policy, an example of security policy that several HSE mechanisms can enforce. We then introduce a minimal x86 hardware model (Section 4.3) that we rely on to specify and verify a BIOS HSE mechanism (Section 4.4).

4.2 A formal model to specify HSE mechanisms

To specify HSE mechanisms, we must first rely on a hardware model that describes the system's possible states and evolutions. Then, we have to specify the security policy that HSE mechanism are supposed to enforce

4.2.1 Hardware model and HSE mechanisms

Our hardware model is a LTS (Labeled Transition System) [Loi+95], a generic model well suited to reasoning about the interactions between a system and its environment. In our case, we focus on the interactions between hardware components and the software they execute.

Definition 4.1 (Hardware Model)
A hardware model Σ is a tuple $\langle H, L_S, L_H, \rightarrow \rangle$, where

- *H is the set of configurations of the hardware architecture*
- *L_S is the set of labels to identify software transitions*
- *L_H is the set of labels to identify hardware transitions*
- *$\rightarrow$ is the transition relation of the system*

The transition relation $\rightarrow$ is a predicate on $H \times L \times H$, where $L = L_S \uplus L_H$ and $\uplus$ is the disjoint union which requires that $L_S \cap L_H = \emptyset$. A transition labeled with l from h to h' is denoted by

$$h \xrightarrow{l} h'$$

and we write $\mathcal{T}(\Sigma)$ for the set of triples which satisfy the transition relation, that is

$$\mathcal{T}(\Sigma) \triangleq \{\ (h, l, h') \mid h \xrightarrow{l} h'\ \}$$

The set of hardware configurations H represents the possible states of the hardware, i.e., CPU modes, values stored in registers, or memory content. The granularity of the model depends on the HSE mechanism we would like to specify and the corresponding security property.

We distinguish two types of transitions:

- Software transitions L_S directly results from the execution of instructions;
- Hardware transitions L_H model interaction between hardware components, e.g., the keyboard controller raises hardware interrupts when the user presses a key.

The execution of a single instruction may correspond to a succession of different transitions that differ according to the state preceding the execution of the instruction. For example, the execution of the instruction `mov (%ecx),%eax` will usually result in four software transitions: the CPU reads the content of the register `ecx`, interprets this value as an address and reads the main memory at this address, writes this content into the register `eax`, and updates the register `eip` with the address of the next instruction to execute. However, hardware transitions can occur between these transitions, e.g., if a peripheral device performs a DMA. Moreover, if the address stored in `ecx` is not valid, the processor raises an interrupt, and the end of the sequence is different.

The transition relation $\rightarrow$ corresponds to the functional specification of the hardware and defines how the hardware states evolve when transitions occur. From this model, we can define traces, i.e., non-empty sequences of transitions of Σ, such that for two consecutive transitions, the resulting state of the first one is the initial state of the next one.

Definition 4.2 (Traces)
We write $\mathcal{R}(\Sigma)$ for the set of traces of a hardware model Σ, and we consider the following functions:

- *$init : \mathcal{R}(\Sigma) \rightarrow H$ maps a trace to its initial state*
- *$trans : \mathcal{R}(\Sigma) \rightarrow \wp(\mathcal{T}(\Sigma))$ maps a trace to the set of transitions which occurred during the trace*

One of the essential aspects of HSE mechanisms is that they are not safe by default. Indeed, they have to be adequately configured by some trusted software to enforce a given security policy. The functional specification of the hardware does not prevent trusted software from misconfiguring the hardware features of the HSE mechanism. For example, nothing prevents the OS kernel from configuring its memory pages so that user applications can modify them. Such configuration, however, will break the isolation the kernel is supposed to enforce.

To specify an HSE mechanism, we also have to specify the requirements that the trusted

software must comply with to guarantee the security policy. Such requirements consist of restrictions on software transitions. However, such restrictions only apply to trusted software. Indeed, in our attacker model, we consider that the trusted software is not malicious. However, the untrusted part can be malicious or vulnerable to attacks. Since it is not trusted, we assume that the attacker can execute any instructions in any order allowed by the functional specification of the hardware.

To distinguish transitions performed by the trusted components from the other ones, we first need to map each state to the software component currently executed. The definition of the trusted software depends on the HSE mechanism. From a practical point of view, this mapping usually corresponds to the privilege mode of the processor, i.e., the different rings or the SMM. Indeed, trusted components are supposed to be executed in the most privileged modes of the CPU. For example, OS kernels should be executed in ring 0, whereas user applications should be executed in less privileged ring 3.

Definition 4.3 (Hardware-Software Mapping)
Given S a set of software components, a hardware-software mapping context : $H \rightarrow S$ is a function which takes a hardware state and returns the software component currently executed.

Dealing with multi-core architectures would require additional efforts and notations. One possible solution could be to define an identifier per core and to use this identifier in addition to the current hardware state to deduce the software component currently executed by the corresponding core. However, this is out of the scope of this work.

We can then define an HSE by a set of trusted software components, a mapping between the system states and the executed software component, and requirements on transitions and system states. These requirements specify the safe states and transitions, i.e., software executions that should ensure the security policy.

Definition 4.4 (HSE Mechanism)
A HSE mechanism Δ is a tuple $\langle S, T, context, hardware\_req, software\_req \rangle$, such that

- *S is the set of software components executed by the hardware architecture.*
- *$T \subseteq S$ is the set of trusted software components which implement the HSE mechanism and form its TCB.*
- *context is a hardware-software mapping to determine which software component is currently executed by the core.*
- *hardware_req is a predicate on H to distinguish between safe hardware configurations and potentially vulnerable ones.*
- *software_req is a predicate on $H \times L_S$ to distinguish between software transitions that trusted software components can safely use, and potentially harmful ones they need to avoid.*

To have a consistent definition of HSE, we also have to impose some restrictions on the *hardware_req* and *software_req* predicates, called HSE laws. First, as explained before, we must not impose any restriction on the execution of the non-trusted software components since they may be under the attacker's control. Thus, when the processor executes non-trusted code,

software_req must hold. Moreover, safe transitions must preserve safe states, i.e., any safe transition executed from a safe state must reach another safe state.

Definition 4.5 (HSE Laws)
A HSE mechanism Δ has to satisfy the following properties:

1. *Untrusted software transitions satisfy software_req:* $\forall (h, l, h') \in \mathcal{T}(\Sigma)$, $\forall x \notin T$,

$$(l \in L_S \wedge context(h) = x) \Rightarrow software\_req(h, l)$$

2. *hardware_req is an invariant with respect to software_req:* $\forall (h, l, h') \in \mathcal{T}(\Sigma)$,

$$(hardware\_req(h) \wedge (l \in L_S \Rightarrow software\_req(h, l))) \Rightarrow hardware\_req(h')$$

Requirements on states and transitions define a safety property that trusted software must verify: starting from a safe state, they shall never generate unsafe transitions. We call compliant traces the set of execution traces that start from safe states and always satisfy the requirement on transitions.

Definition 4.6 (Compliant Traces)
We write $\mathcal{C}(\Delta)$ for the set of the traces which comply with Δ. Given $\rho \in \mathcal{R}(\Sigma)$, then $\rho \in \mathcal{C}(\Delta)$ iff

$$hardware\_req(init(\rho)) \wedge \forall (h, l, h') \in trans(\rho),\ l \in L_S \Rightarrow software\_req(h, l)$$

From the previous definition, we can easily prove that compliant traces always stay in safe states, i.e., states that verify the hardware requirement.

Lemma 4.1 (HSE Invariant Enforcement)
As intended, hardware_req is an invariant of traces which comply with Δ, that is

$$\forall \rho \in \mathcal{C}(\Delta), \forall (h, l, h') \in trans(\rho), hardware\_req(h) \wedge hardware\_req(h')$$

Proof. By definition of $\mathcal{C}(\Delta)$, we know the initial state of the trace satisfies the *hardware_req* invariant, and thanks to the second HSE law, we can conclude the state after the first transition also satisfies *hardware_req*. We generalize to the trace by induction. □

Defining a hardware model and a HSE mechanism satisfying HSE laws is just a first step corresponding to the formal specification of the mechanism. This specification cover both the hardware and the software part of the mechanism. In particular, the *software_req* predicate is a specification of the trusted software behavior.

However, this definition of HSE does not specify the security guarantees that these mechanisms are supposed to provide. Thus, a HSE mechanism satisfying HSE laws can fail to enforce a given security property. In such a case, an attacker can violate the security property even if the trusted software always complies with the software requirement, i.e., the system remains in states that are considered "safe" by the hardware requirement. It is thus necessary to formally verify that a HSE mechanism can enforce a given security policy.

4.2.2 Security policy

The security policy can be a predicate on sets of traces, on traces, or on transitions. Such a policy does not generally hold for all the states in the system. Indeed, HSE hardware mechanisms are not safe by default. On the other hand, we expect executions that comply with the software and hardware requirements to satisfy the policy. In other words, HSE mechanisms must enforce their policy if the trusted software correctly configures them.

Definition 4.7 (Correct HSE Mechanism)
A HSE mechanism Δ is correct with respect to a security policy P (denoted by $\Delta \models P$) if and only if the set of compliant traces of Δ satisfies P, that is

$$\Delta \models P \triangleq \begin{cases} P(\mathcal{C}(\Delta)) & \text{if } P \text{ is a predicate on sets of traces} \\ \forall \rho \in \mathcal{C}(\Delta), P(\rho) & \text{if } P \text{ is a predicate on traces} \\ \forall \rho \in \mathcal{C}(\Delta), \forall tr \in trans(\rho), P(tr) & \text{if } P \text{ is a predicate on transitions} \end{cases}$$

Depending on the type of security property and the complexity of the hardware model, verifying that an HSE mechanism is correct can be quite complex.

To the extent of our knowledge, most HSE mechanisms enforce simple access control policies that can be modeled as predicates on transitions. Thus, we focus in this work on such types of predicates. In that case, the definition of a correct HSE mechanism leads to Theorem 4.1.

Theorem 4.1 (Correct HSE Mechanism for Predicate on Transitions)
Given a security policy P defined as a predicate on transitions, then

$$\begin{aligned} &\forall (h, l, h') \in \mathcal{T}(\Sigma), \\ &\quad (hardware\_req(h) \wedge (l \in L_S \Rightarrow software\_req(h, l))) \Rightarrow P(h, l, h') \end{aligned}$$

is a sufficient condition for

$$\Delta \models P$$

Proof. By definition, a HSE mechanism Δ is correct with respect to a security policy characterized by a predicate on transitions P if the transitions of its compliant traces satisfy P. With Lemma 4.1, we know that initial states of a transition of compliant traces satisfy *hardware_req*. We also know by definition of compliant traces that their transitions satisfy *software_req*. Therefore, if we can prove that transitions which satisfy both *hardware_req* and *software_req* also satisfy P, then we can conclude that transitions of compliant traces satisfy P. □

Predicates on traces correspond in the more general case to safety properties [Lam85]. Such properties may be more difficult to prove since they may require reasoning on all the trace prefixes and not only on every single transition. This type of property corresponds to access control policies that runtime monitors can enforce [Sch00a].

However, not all security policies can be formalized with predicates on traces. For instance, *noninterference* [GM82] is a confidentiality policy that requires a system to always produce the same public outputs for the same public inputs, regardless of the secret inputs. In this

case, it is not sufficient to reason on each trace independently to verify the policy. Such *hyperproperties* [CS10] correspond to predicate on a set of traces and are the most difficult to prove.

SpecCert is a generic framework that can be used to specify and verify HSE mechanisms. This framework is parametrized by the hardware model and the security policy the HSE mechanism is supposed to enforce. One of the critical aspects of HSE mechanisms is that some trusted software must configure them properly to enforce the security policy. Thus, to prove that an HSE mechanism is correct, we assume that the trusted software configures it correctly (i.e., *software_req* is true). This assumption can be verified by analyzing the source or the binary code of the trusted software. However, we also have to verify that an attacker cannot modify this code at runtime, i.e., untrusted software cannot inject code in trusted software memory. Usually, HSE mechanisms implement auto-protection or rely on other HSE mechanisms to prevent malicious code injection. Code injection policy is thus essential for HSE mechanisms, and we decided to focus our work on this type of policy, which is detailed in the following section.

4.2.3 Code injection policy

To enforce a code injection policy, we need to track the owner of each instruction in the memory, i.e., which software component has injected such instruction in the memory. We define a transition-software mapping function $fetched_\Sigma : H \times L \to \wp(S)$, which takes an initial hardware state, a transition label and returns the set of owners of the instruction executed by the CPU during the transition.

Before defining the code injection policy, we first have to give a formal definition of code injection.

Definition 4.8 (Code Injection)
A software component $x \in S$ achieves a code injection against another software component $y \in S$ during a transition labeled with $l \in L$ from a state $h \in H$ to a state $h' \in H$, denoted by

$$h \xrightarrow[x \rightsquigarrow y]{l} h'$$

when the processor fetches an instruction owned by x while executing y, that is

$$h \xrightarrow[x \rightsquigarrow y]{l} h' \triangleq x \in fetched_\Sigma(h, l) \wedge context(h) = y$$

We write $h \xrightarrow[x \not\rightsquigarrow y]{l} h'$ when x does not achieve a code injection against y.

Note that code injection is not necessarily malicious. Trusted software components usually inject code in non-trusted ones. For example, the OS kernel injects the application code before starting a new application, e.g., when executing the `execve` syscall in Linux. A code injection policy can be used to specify the legitimate code injections.

Definition 4.9 (Code Injection Policy)
A code injection policy I is a safety property characterized by a tuple $\langle S, \rightarrowtail, context \rangle$, such that

- *S is a set of software components executed by the hardware architecture*
- *$\rightarrowtail$ is a binary relation on S, such that given $(x, y) \in S \times S$, $x \rightarrowtail y$ means x is authorized to make a code injection against y. $\rightarrowtail$ has to be anti-symmetric and transitive.*

$$\begin{aligned} &\forall (x, y) \in S \times S, \\ &\quad x \rightarrowtail y \wedge y \rightarrowtail x \Rightarrow x = y \quad (1) \\ &\forall (x, y, z) \in S \times S \times S, \\ &\quad x \rightarrowtail y \wedge y \rightarrowtail z \Rightarrow x \rightarrowtail z \quad (2) \end{aligned}$$

- *context is a hardware-software mapping.*

A transition labeled with $l \in L$ from $h \in H$ to $h' \in H$ satisfies I when code injections which occur during this transition are authorized by $\rightarrowtail$, that is

$$I(h, l, h') \triangleq \forall (x, y) \in S \times S,\ h \xrightarrow[x \rightsquigarrow y]{l} h' \Rightarrow x \rightarrowtail y$$

For example, only the BIOS can inject code in itself to prevent applications and the OS from tampering with the BIOS execution. In that case, we have $S = \{\texttt{bios}, \texttt{OS}\}$, $\forall x \in S, x \rightarrowtail x$, and $\forall x \in S, \texttt{bios} \rightarrowtail x$. This leads us to the following definition of $I_{\texttt{bios}}$.

Definition 4.10 (BIOS Code Injection Policy)

$$I_{\texttt{bios}}(h, l, h') \triangleq \forall x \in S, h \xrightarrow[x \rightsquigarrow \texttt{bios}]{l} h' \Rightarrow x = \texttt{bios}$$

To evaluate SpecCert, we use it to specify a BIOS HSE mechanism. We then prove that such a mechanism enforces the code injection policy defined above. To specify this mechanism, we define a minimal model of the x86 platform in the next section, allowing us to reason about the BIOS isolation mechanism.

4.3 A minimal x86 hardware model

MINX86 is a minimal hardware model of the x86 platform. We defined such a formal model based on the public specification released by Intel [Int13; Int09a; Int14]. This model is a proof-of-concept to illustrate how the SpecCert framework can be used, and does not pretend to be exhaustive. This model focuses on the SMM isolation mechanism, but we could extend it to consider other HSE mechanisms.

4.3.1 Model scope

We model a single-core CPU with two execution modes: non-privileged and SMM. As explained in Section 2.2.1, only the processor in SMM can access the SMRAM, which is a memory range

of the DRAM. The BIOS is supposed to store the code of its runtime services inside this isolated part of the memory to protect them from the OS and user applications.

We also model a single-level of cache and consider two caching strategies: uncacheable (UC) and writeback (WB). The software executed by the processor can configure the caching strategy of the SMRAM using SMRR (System Management Range Register). The software can only modify the value of this register if the processor is in SMM. This register specifies the SMRAM memory range and the corresponding caching strategy used when the processor is in SMM. In the un-privileged mode, the processor always used the UC strategy for memory accesses targeting this memory range.

The chipset manages the memory accesses that the cache has not handled. The access can be forwarded to different hardware components depending on the physical address and the memory mapping. We model the following physical mapping of the SMRAM: if the CPU is in SMM, the chipset forwards any accesses targeting the SMRAM address range to the DRAM controller; otherwise, it forwards such accesses to the video controller. This access control mechanism is parametrized by the SMRAMC (System Management RAM Control) register, which can only be modified in SMM.

The SMRR registers have been introduced by Intel to protect the BIOS code from the SMRAM cache poisoning attack disclosed in 2009 [Loi+09; RJ09]. Before that, the HSE mechanisms implemented by the SMRAMC register could not correctly enforce the BIOS code injection policy presented in Section 4.2.3. Indeed, even if the trusted software correctly configures the mechanism, i.e., the BIOS correctly configured the SMRAMC register and places its runtime service code in the SMRAM, an attacker was still able to inject some code from the non-privileged mode.

This attack was possible because the access control was only performed on accesses managed by the chipset and did not take the cache into account. If the OS configured a WB strategy to the memory range corresponding to the SMRAM, it could inject some malicious code in the part of the cache corresponding to the SMRAM. Then, once in SMM, the processor would fetch the code located in the cache rather than the BIOS code stored in the SMRAM. This vulnerability is due to the interactions of different hardware components, i.e., the cache, the chipset, and the CPU's core. Such types of attacks are a strong motivation for our work.

4.3.2 Tracking memory ownership

The definition of the MINX86 hardware model is parameterized by a *context* hardware-software mapping :

$$\text{MINX86}(\mathit{context}) \triangleq \langle H(S), L_S, L_H, \xrightarrow[\mathit{context}]{} \rangle$$

The memory locations of MINX86 are either cache lines or memory cells exposed by the DRAM or the video controllers. The memory ownership is updated through transitions according to three rules:

1. When a cache line gets a copy of a DRAM or video cell content, the owner of this cell becomes the new owner of this cache line.
2. When the content of a cache line is written back to a memory cell, the new owner of this memory cell is the owner of the cache line.

3. During a software transition whose purpose is to overwrite the content of a memory location, the software component currently executed —and therefore responsible for the software transition— becomes the new owner of this memory location.

4.3.3 Hardware states

The hardware state of the platform $H(S)$ is defined as the Cartesian product of the set of states of the core, the cache, the chipset and the memories exposed by both the DRAM and the video controllers:

$$H(S) \triangleq \langle\ core : \texttt{Core},\quad cache : \texttt{Cache}(S),\quad chipset : \texttt{Chipset},\quad mem : \texttt{Mem}(S)\ \rangle$$

Since the chipset can forward a given physical address to different devices, depending on its configuration and the privileged mode of the CPU, we consider two types of addresses: physical and hardware addresses.

Physical addresses (`PhysAddr`) are used by the CPU when performing some memory access. The chipset then translates such an address to a corresponding hardware address (`HardAddr`). For any physical address Pa(i), with $i \in [0, \texttt{max\_addr}]$, we note Dram($i$) the hardware address if the chipset translate the access to the DRAM controller and Vga(i) otherwise. `max_addr` correspond to the highest physical address and depends on the configuration of a given platform, e.g., the total amount of memory available on the platform.

CPU Cores

We define a minimal CPU core state, focused on the SMM:

$$\texttt{Core} \triangleq \left\langle \begin{array}{c} in\_smm : \{\,\texttt{true}, \texttt{false}\,\},\quad pc : \texttt{PhysAddr},\quad smbase : \texttt{PhysAddr}, \\ smrr : \texttt{Smrr},\quad strat : \texttt{PhysAddr} \to \texttt{CacheStrat} \end{array} \right\rangle$$

The core state takes into account the privilege level of the core (whether the core is in SMM or not), the value of the program counter, SMBASE and SMRR registers, as well as the global caching strategy of the platform. We abstract the different mechanisms the OS can use to specify this caching strategy by the *strat* function, which maps each physical address to a caching strategy. In our use case, we consider both the uncacheable (`UC`) and writeback (`WB`) caching strategy, so $\texttt{CacheStrat} \triangleq \{\,\texttt{UC}, \texttt{WB}\,\}$.

We model the SMRR register value by a physical address range and the corresponding caching strategy:

$$\texttt{Smrr} \triangleq \langle\ range : \wp(\texttt{PhysAddr}),\quad strat : \texttt{CacheStrat}\ \rangle$$

This caching strategy can only be set in SMM and has priority over the global caching strategy of the platform, which the OS can define.

Cache

We model a unified single level cache as a mapping between index and cache lines:

$$\texttt{Cache}(S) \triangleq \texttt{Index} \to \texttt{CacheLine}(S)$$

For each cache line, we model the dirty bit state, the value stored in the cache line, and the corresponding physical address. We also track the owner of the value stored in the cache:

$$\texttt{CacheLine}(S) \triangleq \langle\ dirty : \{\,\texttt{true}, \texttt{false}\,\},\quad tag : \texttt{PhysAddr},\quad content : \texttt{Val},\quad owner : S\ \rangle$$

Chipset

We use a very simple model of the chipset, focused on the access control to the SMRAM:

$$\texttt{Chipset} \triangleq \langle\ d\_open : \{\,\texttt{true}, \texttt{false}\,\},\quad d\_lock : \{\,\texttt{true}, \texttt{false}\,\}\ \rangle$$

d_open and d_lock represent the two bits of the SMRAMC register of the chipset.

We also model how the chipset dispatches the memory request by the following function:

$$dispatch(\text{chipset}, \text{in\_smm}, \text{pa}) \triangleq \begin{cases} \textsc{Vga}(i) & \text{if in\_smm} = \texttt{false},\ pa \in \texttt{pSmram}, \\ & \quad \text{and chipset}.d\_open = \texttt{false} \\ \textsc{Dram}(i) & \text{otherwise} \end{cases}$$

$$\text{where } pa = \textsc{Pa}(i).$$

As explained in Section 4.3.1, the chipset dispatches any SMRAM memory access to the DRAM if the processor is in SMM and to the video controller otherwise.

Memory

We model the memory as a mapping between hardware addresses and values. We also track the owner of each value:

$$\texttt{Mem}(S) \triangleq \texttt{HardAddr} \to \langle\ content : \texttt{Val},\quad owner : S\ \rangle$$

4.3.4 Transitions

The transition relation of Minx86 is defined as follows:

$$h \xrightarrow[context]{l} h' \triangleq pre(l, h) \wedge post(l, h, h')$$

For each transition label we define :

- a precondition (*pre*) to determine whether the transition can occur from a given hardware state, and
- a postcondition (*post*) to specify the consequences of the transitions over the hardware state.

Software Transitions

Table 4.1 gives an overview of the different software transitions we consider in Minx86.

We model the core I/Os with *Read*(pa) and *Write*(pa), the configuration of the cache strategy with *SetCacheStrat*(pa, strat), the configuration of the chipset with *OpenBitFlip* and *LockSmramc*, the configuration of the SMRR with *UpdateSmrr*(smrr), the exit of the SMM with *Rsm*, and the update of the core program counter register with *NextInstruction*(pa).

The details of the preconditions and postconditions associated with each software transition are available in the source code of the SpecCert project [Let], in the module `SpecCert.x86.Transition`. In summary, there is no precondition associated with *Read*, *NextInstruction*, and *Write* events

Event	Parameters	Description
Write	pa ∈ `PhysAddr` $v \in$ `Val`	A core I/O to write to physical address pa the value v
Read	pa ∈ `PhysAddr`	A core I/O to read from physical address pa.
SetCacheStrat	pa ∈ `PhysAddr` strat ∈ `CacheStrat`	Change the caching strategy for pa to strat
UpdateSmrr	smrr ∈ `Smrr`	Update the SMRR content with the new value *smrr*
Rsm	—	The core leaves SMM
OpenBitFlip	—	Flip the `d_open` bit
LockSmramc	—	Set the `d_lock` bit
NextInstruction	pa ∈ `PhysAddr`	The program counter register of the core is set to *pa*

Table 4.1: List of labels dedicated to Minx86 software transitions (L_S)

since we only focus our model on the SMM (e.g., pagination is not modeled). However, the postcondition of such events depends on the cache state, the cache strategy of the memory location, the memory address (whether it belongs to the SMRAM or not), the state of the chipset (whether the SMRAM is locked or not), and the CPU state (whether it is executing in SMM or not). This postcondition determines the location of the value that is read or modified. The chipset enforces a simple access control to protect the SMRAM content in the DRAM memory by forwarding the related I/O to the video controller when the CPU is not in SMM. The postcondition of *Write* also updates the owner of the memory location, based on the *context* hardware-software mapping.

Since the caching strategy can be specified in SMM and regular mode, there is no precondition for *SetCacheStrat*. Its postcondition reflects the modification of the caching strategy. However, updating the SMRR can only be done in SMM, so the precondition of *UpdateSmrr* verifies that the CPU is executing in SMM. The preconditions of *OpenBitFlip* and *LockSmramc* require the SMRAMC register to be unlocked. The postconditions of these events modify the value of the corresponding registers.

Hardware Transitions

Table 4.2 gives an overview of the different hardware transitions we consider in Minx86. Since SMM is not reentrant, the precondition of *ReceiveSMI* verifies that the CPU is in SMM. Its postcondition puts the CPU in SMM and jumps to the SMI handlers located in SMRAM.

The *Fetch* event corresponds to the fetching of instructions pointed by the program counter. This hardware event can be triggered by a software modification of the program counter (i.e., the software event *NextInstruction*), but it can also be the result of some hardware events (i.e., during interrupts handling). *Fetch* shares the same preconditions and postconditions as *Read*, since fetching instructions is very similar to reading data in memory.

Event	Description
ReceiveSMI	A SMI is raised and the core enters SMM
Fetch	A core I/O to fetch the instruction stored at the physical address contained in the program counter register

Table 4.2: List of labels dedicated to Minx86 hardware transitions (L_H)

We define Minx86_*fetched*, a transition-software mapping for Minx86. This function maps the initial state and the label of this transition to the set of software components which own the instruction fetched during this transition. In Minx86, only the *Fetch* event is supposed to fetch instructions and Minx86_*fetched* is defined as follows:

$$\text{Minx86\_}fetched(h,l) = \begin{cases} \emptyset & \text{if } l \neq Fetch \\ \emptyset & \text{if } h.core.pc \in h.core.smrr.range \\ & \quad \text{and } h.core.in\_smm = \texttt{false} \\ \{h.cache(\mathrm{i_i}).owner\} & \text{if } read\_from\_cache(h, \mathrm{pc}) = \texttt{true} \\ \{h.mem(\mathrm{h_i}).owner\} & \text{otherwise} \end{cases}$$

with

- $\mathrm{h_i} = dispatch(h.chipset, h.in\_smm, h.core.pc)$, the hardware address of the fetched instruction;
- $\mathrm{i_i} = index(h.core.pc)$, the cache index of the fetched instruction.

This definition relies on the *read_from_cache* helper function, which returns true only if there is a cache hit and the caching strategy is set to writeback.

4.4 Specifying and verifying a BIOS HSE mechanism

We rely on SpecCert to specify the HSE mechanisms used to isolate the BIOS runtime services from other software components. For this use-case, we only consider two software components: `bios` represents the BIOS runtime services executed in SMM while `OS` represents the code of the OS and user applications, which are executed in non-SMM mode. Thus, $S = \{\texttt{bios}, \texttt{OS}\}$ and `bios` is the trusted software component that configures this HSE mechanism.

We define $\Delta_{\texttt{bios}}$ to model the HSE mechanism used by the BIOS:

$$\Delta_{\texttt{bios}} = \langle S, \{\,\texttt{bios}\,\}, context, hardware\_req, bios\_req \rangle$$

We define *context*, the hardware-software mapping of this HSE mechanism, such that

$$context(h) \triangleq \begin{cases} \texttt{bios} & \text{if } h.in\_smm = \texttt{true} \\ \texttt{os} & \text{otherwise} \end{cases}$$

Then, we have to extract from Intel specification how the BIOS has to configure and use this mechanism correctly. This step helps us to define the *hardware_req* and *bios_req* requirements. We then have to verify that such definitions satisfy the HSE laws (Definition 4.5).

4.4.1 Requirements over states.

We have identified six requirements for *hardware_req*:

1. When the core executes the SMM code, the program counter register must point to an address in SMRAM.

$$context(h) = \texttt{bios} \Rightarrow h.core.pc \in \texttt{pSmram}$$

2. The SMBASE register must have been set correctly during the boot sequence to point to the base of the SMRAM.

$$h.core.smbase = \textsc{Pa}(\texttt{smram\_base})$$

3. All the memory locations within the SMRAM must be owned by the BIOS.

$$\forall \text{ha} \in \texttt{hSmram}, h.mem(\text{ha}).owner = \texttt{bios}$$

4. For a physical address in SMRAM, in case of cache hit, the related cache line content must be owned by the SMM code.

$$\forall \text{pa} \in \texttt{pSmram}, cache\_hit(h.cache, \text{pa}) \Rightarrow h.cache(index(\text{pa})).owner = \texttt{bios}$$

5. In order to protect the content of the SMRAM inside the DRAM memory, the boot sequence code must have locked the SMRAMC control register. This ensures that an OS cannot set the `d_open` bit any longer and only a core in SMM can modify the content of the SMRAM.

$$h.chipset.d\_lock = \texttt{true}$$

6. The range of memory declared with the SMRR must include the SMRAM.

$$\texttt{pSmram} \subseteq h.core.smrr.range$$

During the boot sequence, the BIOS is the only software component that the CPU executes. Thus, the requirements only apply once the OS starts its execution after the boot sequence, which corresponds to the initial state in our model. We assume that the BIOS correctly configures the platform and that the initial state of the OS execution satisfies *hardware_req*.

4.4.2 Requirements over transitions.

We only define two restrictions for *bios_req*: the code executed in SMM must only come from the SMRAM and must not update the value of SMRR registers at runtime after the boot sequence. Indeed, the code located outside the SMRAM can be modified by the OS and could be controlled by an attacker. Moreover, the range of addresses corresponding to the SMRAM is not supposed to evolve at runtime.

$$
\begin{aligned}
\mathit{bios\_req}(h,l) \triangleq\ & \mathit{context}(h) = \texttt{bios} \\
& \Rightarrow ((\forall \text{pa} \in \texttt{PhysAddr}, \\
& \qquad l = \mathit{NextInstruction}(\text{pa}) \Rightarrow \text{pa} \in \texttt{pSmram}) \\
& \quad \wedge (\forall \text{smrr} \in \texttt{Smrr}, l \neq \mathit{UpdateSmrr}(\text{smrr})))
\end{aligned}
$$

4.4.3 Verifying HSE laws

We then have to prove that these requirements are consistent, i.e., they satisfy the HSE laws 4.5. First, requirements on transition must only restrict the behavior of the trusted software components, i.e., the BIOS in this use case. This first law can be easily proven, since $\mathit{context}(h) = \texttt{bios}$ is the antecedent of the conditional $\mathit{bios\_req}$.

Secondly, $\mathit{hardware\_req}$ must be preserved by executions that satisfy $\mathit{bios\_req}$. This second law can be proven by case enumeration of events and hardware states. The details of the proof are available in the source code of the SpecCert project [Let], in the module `SpecCert.Smm.Delta.Preserve`.

4.4.4 Proving the correctness of the BIOS HSE mechanism

Our final objective is to prove that the HSE mechanism $\Delta_{\texttt{bios}}$ enforces the BIOS code injection policy (Definition 4.10), i.e., only the BIOS itself can inject code in the BIOS runtime services.

Since $I_{\texttt{bios}}$ is a predicate on transitions, we know from Theorem 4.1 that

$$
\begin{aligned}
&\forall (h,l,h') \in \mathcal{T}(\textsc{Minx86}(\mathit{context})), \\
&\quad (\mathit{hardware\_req}(h) \ \wedge\ (l \in L_S \Rightarrow \mathit{bios\_req}(h,l))) \Rightarrow I_{\texttt{bios}}(h,l,h')
\end{aligned}
$$

is a sufficient condition for $\Delta_{\texttt{bios}} \models I_{\texttt{bios}}$.

This condition can be proven by case enumeration of events and hardware states. The details of the proof are available in the source code of the SpecCert project [Let], in the module `SpecCert.Smm.Delta.Secure`. Notice that most cases are trivial to prove, since policy $I_{\texttt{bios}}$ only imposes restrictions on transitions that fetch instructions. In our use case, only the fetch event is impacted.

This proof states that the HSE mechanism $\Delta_{\texttt{bios}}$, which relies on the SMM to isolate the BIOS runtime services, prevents any OS or application from injecting code in SMM, provided that the BIOS configures and uses this mechanism as expected by the specification. In particular, the BIOS boot sequence must perform a correct configuration of the SRAM, i.e., the SMBASE register and the SMRR registers values must correspond to the SMRAM address range, and the SMRAMC must be locked. Moreover, the BIOS runtime services executed in SMM must not jump into code located outside the SMRAM nor update the SMRR value.

4.5 Conclusion and perspectives

This chapter summarizes my contribution to formal specification and verification of security mechanisms involving hardware and software components. We focused our work on a specific

class of security mechanisms we call HSE. Such security mechanisms rely on hardware features that are not safe by default. They have to be correctly configured by some trusted software components to enforce a given security policy. We proposed SpecCert, a framework in Coq to specify and verify HSE mechanisms. We relied on this framework to specify the HSE mechanisms used by the BIOS to isolate its runtime services and to prove that this mechanism enforces a code injection policy. To the extent of our knowledge, this work was the first formalization of the BIOS security model at runtime.

This work was published at FM (FME) 2016 conference [17] and was done during the Ph.D. of Thomas Letan, in the context of a bilateral collaboration project with the French National Agency for the Security of Information Systems (ANSSI). Thomas realized his Ph.D. while being a research engineer at ANSSI in the Hardware and Software Security Laboratory, and he was co-advised by Pierre Chifflier from the ANSSI. The initial objective of this project was to study the UEFI specifications from a security perspective. However, Thomas proposed to focus his Ph.D. on using formal methods to enhance the security of UEFI firmware and realized all the Coq development of FreeSpec. His scientific contribution goes far beyond the security of the UEFI firmware. Thomas continued his work at ANSSI as a formal method expert in the Software Security Laboratory. He is now an expert engineer at Nomadic Labs, working on the development and formal verification of the Thesos blockchain.

Modular verification. Formalizing a whole hardware platform composed of multiple interconnected components is the major challenge of SpecCert. Indeed, the hardware model implemented in SpecCert is monolithic. Thus, any modification of the hardware model may impact all the proofs. Moreover, the SpecCert hardware model embeds meta-data specific to a particular security policy, e.g., memory ownership. This approach contradicts our objective to provide a generic hardware model to specify and verify different HSE.

To tackle these limitations, we have proposed FreeSpec, a new approach to verify the composition of hardware and software components. This approach allows the modular verification of complex hardware architecture using Coq. However, it is not specific to hardware but could also apply to software applications. This work was published at the FM 2018 conference [9] and in the journal Formal Aspects of Computing [31] in 2020. We also developed this contribution during the Ph.D. of Thomas Letan, in collaboration with Yann Régis-Gianas from the Inria Pi.R2 team, which leads the Coq formal proof assistance software development.

An interesting perspective would be to reuse the FreeSpec framework to rewrite and extend the hardware model of SpecCert. The objective would be to provide a more detailed model to verify and specify x86-based HSE mechanisms.

Hardware model validation. Ultimately, we aim to extend these proofs to a physical hardware platform. Therefore, we have to establish the equivalence between the model and the implementation. We could leverage the code extraction feature of Coq to generate an executable program from the hardware model. We could then use this extracted program to validate the model, e.g., by comparing execution traces of the program with the ones observed on the hardware platform. However, the validation of internal components, like the PCH, may be challenging because we do not have access to their internal interfaces.

Another solution is to generate the hardware implementation from the formal model. In a new collaboration with the ANSSI on the formalization of security mechanisms for the RISC-V processor architecture, we follow that direction in the context of a strategic partnership be-

tween Inria and the ANSSI. In this project, I co-supervise the Ph.D. of Matthieu Baty with my colleagues Pierre Wilke from CentraleSupélec and Arnaud Fontaine from the ANSSI. Targeting an open-source architecture allows us to generate a softcore implementation of our design on FPGA. More precisely, our objective is to develop a hardware model of a RISC-V processor in Coq, specify and verify HSE mechanisms that rely on hardware features provided by this platform, and then extract an HDL description of the CPU from the hardware model. Finally, we would like to use this HDL description to synthesize the hardware design on FPGA. We chose to base our first work on the Kôika framework [Bou+20], developed at MIT. This framework provides a Domain Specific Language embedded in Coq to develop hardware components, a formal semantic of this language, and a formally verified compiler that generates circuits guaranteed to implement the designs correctly. Matthieu is currently developing a formally verified shadow stack mechanism that enforces a CFI policy on a RISC-V core, using Kôika.

Verification of more complex HSE mechanisms. In SpecCert and FreeSpec, we focus on isolation mechanisms that enforce invariants on transitions. Targeting mechanisms that enforce more complex policies, such as DIFT mechanisms, is more challenging. Indeed, such mechanism are supposed to guarantee some forms of non-interference, which correspond to predicates on sets of traces, i.e., hyperproperties [CS10]. Several works proposed formal approaches to verify DIFT implementations at the hardware level [Ard+19; ZSM19; Den+19]. However, these works focused on mechanisms entirely implemented in hardware. We want to explore the formal specification and verification of hybrid DIFT approaches, like the one we proposed in the HardBlare project, as presented in Section 2.3. One of the challenges is to reason on both static and dynamic analysis information [MC11].

Chapter 5

Conclusion and perspectives

This manuscript provides an overview of my research activities in computer security at the hardware/software boundary. I believe that host-level hardware and software mechanisms must be combined to detect and respond to intrusions. Furthermore, formal approaches that cover both hardware and software must be proposed to verify these complex mechanisms.

In Chapter 2, I presented my contributions to hardware-assisted intrusion detection. Then, Chapter 3 detailed our intrusion recovery and response solution. Finally, I introduced in Chapter 4 our approach to specify and verify HSE mechanisms using formal methods.

In the future, I wish to pursue my research project on the security of software/hardware interfaces by developing three complementary axes:

1. taking advantage of hardware resources (e.g., a coprocessor or processor security extensions) and the OS to monitor and protect the platform;
2. proposing mechanisms for self-monitoring and self-protection of applications by static analysis and instrumentation of their code;
3. using formal methods to specify and verify security mechanisms involving software and hardware.

In particular, I would like to focus on issues at the intersection of these different research areas. I have described some perspectives and future work at the end of each chapter in Sections 2.4, 3.4, and 4.5. In the following, I give more general perspectives related to these research areas.

Guest-assisted introspection for intrusion detection and reaction Isolating a host-based monitor creates a semantic gap that remains a significant challenge for this type of approach [Jai+14]. Existing approaches usually rely on manual expert knowledge, binary analysis, or compiler-assisted approaches to bridge the semantic gaps. However, these approaches have limited flexibility, practicality, coverage, or automation [BAL15b].

Combining approaches that rely on a model of the monitored component, established before execution (e.g., during compilation), and on the modification of the monitored component (e.g., by instrumentation) offer the best compromise between the previous metrics. In such target-assisted approaches, the monitored component participates in its monitoring. Thus, the problem of trust in the observation arises, as the attacker can exploit a vulnerability in the monitored component to bypass the monitoring.

Hardware mechanisms could be used to protect the part of the monitor integrated with the monitored components. A complementary approach is to use paraverification, which allows the monitor to ensure that the information provided by the monitored components is consistent [BAL15b]. These approaches are still preliminary and deserve further exploration.

In 2022, we will co-advise a master student internship with my colleagues Pierre Wilke and Frédéric Tronel to design and implement a DSL (Domain Specific Language) for hypervisor-level introspection. Our objective is to rely on a proto-hypervisor to monitor guest OS and applications. The hypervisor will provide services and introspection capabilities to monitor the guest OS. We will design a novel DSL that will expose the introspection capabilities to the guest OS. Such a DSL, inspired by the eBPF (extended Berkeley Packet Filter) approach used in the Linux kernel, will be interpreted inside the hypervisor.

In the future, we would like to develop a correct-by-construction DSL verifier and JIT (just-in-time) compiler. Compared to the eBPF approach, the verifier will be enhanced to ensure safety and termination even in the presence of loops and function calls. Moreover, the verifier will also ensure that hypervisor resources are not exhausted and correctly managed.

Then, we would like to rely on such a DSL to explore a new collaborative design that leverages the fact that a hypervisor extension is formally proven, allowing the guest to inject an introspection code dedicated to its kernel in the hypervisor. This injection will be done just after the host boot process. Indeed, we assume that the host is not yet compromised at this stage.

Hardware-based isolation of the recovery and response mechanisms In our existing approach, the intrusion recovery and response mechanisms mostly rely on user-mode applications. The enforcement mechanisms rely on kernel features isolated from user applications. However, kernel code is also prone to vulnerabilities that attackers could exploit to circumvent the enforcement mechanisms. Thus, we have to protect the integrity and availability of such tools to prevent an attacker from interfering with the snapshotting or recovering processes.

Most of the research works that rely on hardware-based isolated monitors focus on intrusion detection. Few works take advantage of a hardware-isolated monitor to implement countermeasures by modifying the state of the monitored system [BAL15b].

Using hardware components to implement fine-grained countermeasures (e.g., changing the state of an application) and isolating the reaction mechanism is an interesting perspective, both from a scientific point of view and for industrial applications. Indeed, computer manufacturers are increasingly looking to equip computers with automatic intrusion response capabilities in a standalone manner, i.e., without requiring the use of external security management and administration components. This solution ensures the system's resilience, even when it is no longer centrally managed and supervised, e.g., in roaming and telecommuting scenarios.

This perspective shares similar objectives with the projects presented in chapter one. However, here the goal is not to protect the IDS but the recovery and response mechanisms. Similarly, the semantic gap resulting from the isolation would be the main challenge to address.

Developping hybrid analyses for intrusion detection and reaction In our contributions to hardware-assisted detection, we proposed hybrid approaches combining both static and dynamic analyses. In the future, we would like to explore how we can apply such hybrid approaches to binary code analysis for intrusion detection and response.

Indeed, static analysis of binary code can suffer from over-approximation problems. In particular, the handling of indirect branches, anti-disassembly sequences, and dynamically generated code (JIT-compiled or self-modified) remain difficult to handle in a purely static way, despite advances in the field.

DBI allows dynamic analysis of programs during their execution. This type of analysis does not suffer from the previous limitations. However, purely dynamic analysis can only reason on the current execution trace, which is insufficient to determine some properties, such as non-interference. Moreover, such approaches require adding instructions to the application code, impacting their execution time.

In this context, the use of hybrid analysis, combining static and dynamic analysis, seems particularly promising. Thus, the static analysis can provide the dynamic analysis with information about the non-executed branches and the control flow graph of the application. In addition, the pre-computation of certain information by static analysis makes it possible to limit the use of instrumentation and thus the impact on the execution time. On the other hand, dynamic instrumentation allows adapting to the execution context. For example, it allows to dynamically call on static analysis at runtime, like a JIT compiler, to analyze the code dynamically generated by the application.

In 2022, we will co-advise a master internship with my colleagues Pierre Wilke and Frédéric Tronel on hybrid analyses for DIFT. Our objective is to extend the work on DBI developed by our Ph.D. student Camille Le Bon to implement an efficient DIFT approach. Our objective is to limit the runtime overhead of the DIFT, e.g., by using selective data protection [Pal+21]. We could then extend this approach by offloading the dynamic analyses on a hardware-isolated monitor, similar to our Hardblare project contribution. The main challenge would be to implement the approach without having access to the source code of the monitored application.

Hardware/software co-design for security The hardware-assisted detection approach we present in this manuscript relies on compilation to provide information about the monitored program to the monitor. More generally, we want to explore hardware/software co-design of compiler and hardware security extensions to detect and prevent different classes of attacks.

For example, in the SCRATCHS project (2021-2024) funded by Labex CominLabs, we will focus on protection against cache-based side-channel attacks. SCRATCHS is a collaboration between researchers in the fields of formal methods (Celtique, Inria Rennes), security (Cidre, CentraleSupélec Rennes), and hardware design (Lab-STICC).

There are two extreme competing approaches to ensure the absence of timing leaks. The pure software approach (e.g., constant-time programming [Bar+20]) implements costly software countermeasures but makes few assumptions about the hardware. The pure hardware approach (e.g., cache set randomization [Wer+19]) implements costly hardware countermeasures but makes few assumptions about the running software.

We claim that both software and hardware need to cooperate to eliminate timing leaks without sacrificing efficiency. The gain is twofold: the software may delegate security to the hardware, and the hardware may exploit the properties of the software. A challenge is to have hardware with a precise timing specification and security countermeasures against time side-channels while being as efficient as possible. Another challenge is ensuring that the compiler leverages the hardware support while minimizing the software countermeasures to guarantee the absence

of timing leaks.

In the SCRATCHS project, we aim to co-design (1) a secure processor architecture based on RISC-V and (2) a compiler for this new architecture. The compiler will rely on security features (e.g., cache eviction policy or cache partitioning) provided by the hardware.

In this project, I will supervise the Ph.D. of Jean-Loup Hatchikian-Houdot with my colleagues Frédéric Besson (CELTIQUE team) and Pierre Wilke (CIDRE team) from IRISA. We will focus on the implementation of a compiler for C programs enriched with security annotations. The compiler will leverage security mechanisms provided by the processor. Jean-Loup will collaborate with another Ph.D. student working on the design and implementation of the secure processor, supervised by our colleagues of the Lab-STICC.

Bibliography

[ANS] ANSSI. *HP Sure Start Hardware Root of Trust version A2, embarqué sur les puces NPCE586HA2MX, NPCE586HA2BX, NPCE576HA2YX*. URL: https://www.ssi.gouv.fr/entreprise/certification_cspn/hp-sure-start-hardware-root-of-trust-version-a2-embarque-sur-les-puces-npce586ha2mx-npce586ha2bx-npce576ha2yx/ (cit. on pp. 11, 12).

[Cor] Intel Corporation. *Intel® Software Guard Extensions*. URL: https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html (cit. on p. 11).

[IT-] IT-ISAC. *FAQ*. URL: https://www.it-isac.org/faq (cit. on p. 46).

[LT] Michael Larabel and Matthew Tippett. *Phoronix Test Suite*. URL: https://www.phoronix-test-suite.com/ (cit. on p. 51).

[Let] Thomas Letan. *SpecCert: a framework for the Coq Theorem Prover*. URL: https://github.com/lthms/speccert (cit. on pp. 68, 72).

[lib] musl libc. *musl libc*. URL: https://musl.libc.org/ (cit. on p. 33).

[Lim] ARM Limited. *TrustZone – Arm Developer*. URL: https://developer.arm.com/ip-products/security-ip/trustzone (cit. on p. 11).

[MSW] Tarjei Mandt, Mathew Solnik, and David Wang. *Demystifying the Secure Enclave Processor* (cit. on p. 11).

[Mar] MariaDB Foundation. *mariadb*. URL: https://mariadb.org/ (cit. on p. 48).

[Mic] University of Michigan. *MiBench*. URL: https://vhosts.eecs.umich.edu/mibench/ (cit. on p. 33).

[MITa] MITRE. *Encyclopedia of Malware Attributes*. URL: https://collaborate.mitre.org/ema/ (cit. on p. 41).

[MITb] MITRE. *Malware Capabilities*. URL: https://github.com/MAECProject/schemas/wiki/Malware-Capabilities (cit. on pp. 41, 43).

[ngi] nginx. *nginx*. URL: https://nginx.org/ (cit. on p. 48).

[NR] NSA and Red Hat. *SELinux*. URL: https://selinuxproject.org/ (cit. on p. 41).

[Rar] Keith Rarick. *beanstalkd*. URL: https://kr.github.io/beanstalkd/ (cit. on p. 48).

[Sen] Tomás Senart. *Vegeta*. URL: https://github.com/tsenart/vegeta (cit. on p. 50).

[She] Di Shen. *Defeating Samsung KNOX with zero privilege* (cit. on p. 11).

[SiF] SiFive. *GitHub - sifive/RiscvSpecForma*. URL: https://github.com/sifive/RiscvSpecFormal (cit. on p. 58).

[Teab] 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 (cit. on p. 46).

[The] The Apache Software Foundation. *Apache HTTP Server*. URL: https://httpd.apache.org/ (cit. on p. 48).

[Vie] Martin Viereck. *x11docker: Run GUI applications in Docker*. URL: https://github.com/mviereck/x11docker (cit. on p. 53).

[YZ] Jiewen Yao and Vincent J. Zimmer. *Boot Chain – Putting it all together*. Intel Corporation. URL: https://edk2-docs.gitbook.io/understanding-the-uefi-secure-boot-chain/secure_boot_chain_in_uefi/boot_chain__putting_it_all_together (cit. on p. 13).

[Int21] Intel. *Intel 64 and IA32 Architectures Software Developer Manual*. June 2021 (cit. on p. 57).

[Lim21] ARM Limited. *A-Profile Architectures | Exploration tools*. 2021. URL: https://developer.arm.com/architectures/cpu-architecture/a-profile/exploration-tools (cit. on p. 58).

[Pal+21] T. Palit, J. Firose Moon, F. Monrose, and M. Polychronakis. "DynPTA: Combining Static and Dynamic Analysis for Practical Selective Data Protection." In: *2021 2021 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2021, pp. 1919–1937. DOI: 10.1109/SP40001.2021.00082 (cit. on p. 77).

[Zho+21] Lei Zhou, Fengwei Zhang, Jidong Xiao, Kevin Leach, Westley Weimer, Xuhua Ding, and Guojun Wang. "A Coprocessor-Based Introspection Framework Via Intel Management Engine." In: *IEEE Transactions on Dependable and Secure Computing* 18.4 (2021), pp. 1920–1932. DOI: 10.1109/TDSC.2021.3071092 (cit. on p. 12).

[Bar+20] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. "Formal verification of a constant-time preserving C compiler." In: *Proc. ACM Program. Lang.* 4.POPL (2020), 7:1–7:30. DOI: 10.1145/3371075 (cit. on p. 77).

[Bou+20] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. "The Essence of Bluespec: A Core Language for Rule-Based Hardware Design." In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 243–257. ISBN: 9781450376136. DOI: 10.1145/3385412.3385965 (cit. on pp. 58, 74).

[Cor20] Intel Corporation. *Intel Converged Security and Management Engine (Intel CSME)*. 2020 (cit. on p. 11).

[EB20] Alexander Eichner and Robert Buhren. *All you ever wanted to know about the AMD Platform Security Processor and were afraid to emulate*. 2020 (cit. on pp. 11, 12).

[Int20] Intel. *Intel® 495 Series Chipset Family On-Package Platform Controller Hub (PCH)*. May 2020 (cit. on p. 57).

[Nie+20] K. Nienhuis, A. Joannou, T. Bauereiss, A. Fox, M. Roe, B. Campbell, M. Naylor, R. M. Norton, S. W. Moore, P. G. Neumann, I. Stark, R. N. M. Watson, and P. Sewell. "Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process." In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1003–1020. DOI: 10.1109/SP40000.2020.00055 (cit. on p. 58).

[Sch+20] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. *SGAxe: How SGX Fails in Practice*. https://sgaxeattack.com/. 2020 (cit. on p. 11).

[Vie+20] Manfred Vielberth, Fabian Böhm, Ines Fichtinger, and Günther Pernul. "Security Operations Center: A Systematic Study and Open Challenges." In: *IEEE Access* 8 (2020), pp. 227756–227779. DOI: 10.1109/ACCESS.2020.3045514 (cit. on p. 37).

[Xu+20] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. "Automatic Hot Patch Generation for Android Kernels." In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2397–2414. ISBN: 978-1-939133-17-5 (cit. on p. 53).

[Ard+19] Armaiti Ardeshiricham, Yoshiki Takashima, Sicun Gao, and Ryan Kastner. "VeriSketch: Synthesizing Secure Hardware Designs with Timing-Sensitive Information Flow Properties." In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 1623–1638 (cit. on pp. 36, 74).

[Arm+19] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. "ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS." In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290384 (cit. on p. 58).

[19a] *Breaking Samsung's ARM Trustzone*. Quarkslab, 2019 (cit. on p. 11).

[CRI19] CRIU. *CRIU*. 2019. URL: https://criu.org/ (cit. on p. 48).

[Das+19] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. "A Complete Formal Semantics of X86-64 User-Level Instruction Set Architecture." In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 1133–1148. ISBN: 9781450367127. DOI: 10.1145/3314221.3314601 (cit. on p. 58).

[Den+19] Shuwen Deng, Doğuhan Gümüşoğlu, Wenjie Xiong, Y. Serhan Gener, Onur Demir, and Jakub Szefer. "SecChisel Framework for Security Verification of Secure Processor Architectures." In: *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy*. HASP. June 2019 (cit. on pp. 58, 74).

[Dua+19] Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. "Automating Patching of Vulnerable Open-Source Software Versions in Application Binaries." In: *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019 (cit. on p. 53).

[Ecl19] Eclipse Foundation. *Mosquitto*. 2019. URL: https://mosquitto.org/ (cit. on p. 48).

[GE19] Maxim Goryachy and Mark Ermolov. *Intel VISA: Through the Rabbit Hole*. Positive Technology, 2019 (cit. on p. 12).

[HP 19] L.P. HP Development Company. *HP Sure Start - Automatic firmware intrusion detection and repair*. 2019 (cit. on pp. 11, 12, 39, 53).

[Int19d] Intel Corporation. "System Management Mode." In: *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Jan. 2019. Chap. 34 (cit. on pp. 13, 18).

[MIT19] MITRE. *ATT&CK*. 2019. URL: https://attack.mitre.org/ (cit. on p. 43).

[Pil+19] C. Pilato, K. Wu, S. Garg, R. Karri, and F. Regazzoni. "TaintHLS: High-Level Synthesis for Dynamic Information Flow Tracking." In: *IEEE Transactions on*

Computer-Aided Design of Integrated Circuits and Systems 38.5 (2019), pp. 798–808 (cit. on p. 36).

[Poe19] Lennart Poettering. *systemd System and Service Manager*. 2019. URL: https://www.freedesktop.org/wiki/Software/systemd/ (cit. on pp. 40, 48).

[The19] The Gitea Authors. *Gitea*. 2019. URL: https://gitea.io/ (cit. on p. 48).

[UEF19b] UEFI Forum. *Unified Extensible Firmware Interface Specification*. Version 2.8. Mar. 2019 (cit. on p. 39).

[Wer+19] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. "ScatterCache: Thwarting Cache Attacks via Cache Set Randomization." In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 675–692. ISBN: 978-1-939133-06-9 (cit. on p. 77).

[ZSM19] D. Zagieboylo, G. E. Suh, and A. C. Myers. "Using Information Flow to Design an ISA that Controls Timing Channels." In: *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. 2019, pp. 272–27215. DOI: 10.1109/CSF.2019.00026 (cit. on p. 74).

[Zha+19] Lianying Zhao, He Shuang, Shengjie Xu, Wei Huang, Rongzhen Cui, Pushkar Bettadpur, and David Lie. *SoK: Hardware Security Support for Trustworthy Execution*. 2019. arXiv: 1910.04957 [cs.CR] (cit. on p. 10).

[Cho+18] Haehyun Cho, Penghui Zhang, Donguk Kim, Jinbum Park, Choong-Hoon Lee, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. "Prime+Count: Novel Cross-World Covert Channels on ARM TrustZone." In: *Proceedings of the 34th Annual Computer Security Applications Conference*. ACSAC '18. San Juan, PR, USA: Association for Computing Machinery, 2018, pp. 441–452. ISBN: 9781450365697. DOI: 10.1145/3274694.3274704 (cit. on p. 11).

[Coz+18] 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. 49).

[Dr 18] Dr. Web. *Linux.BackDoor.Fgt.1430*. July 2018. URL: https://vms.drweb.com/virus/?i=17573534 (cit. on p. 49).

[FO18] Uri Farkas and Ido Li On. *AMDFlaws - A Technical DeepDive*. CTS-Labs, 2018 (cit. on p. 12).

[Jia+18] Zhenghong Jiang, Steve Dai, G. Edward Suh, and Zhiru Zhang. "High-Level Synthesis with Timing-Sensitive Information Flow Enforcement." In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2018, pp. 1–8. DOI: 10.1145/3240765.3240813 (cit. on p. 36).

[Mic18a] Microsoft. *Job Objects*. MSDN. May 31, 2018. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms684161(v=vs.85).aspx (cit. on pp. 40, 48).

[Mic18c] Microsoft. *Restricted Tokens*. MSDN. May 31, 2018. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/aa379316(v=vs.85).aspx (cit. on p. 48).

[Mic18d] Microsoft. *Windows Integrity Mechanism Design*. MSDN. July 5, 2018. URL: https://msdn.microsoft.com/en-us/library/bb625963.aspx (cit. on p. 48).

[Nig18] 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/ (cit. on p. 49).

[Reg18] 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 p. 13).

[RR18] Maxwell Renke and Chris Riggs. *Virtualization-based security (VBS) memory enclaves: Data protection through isolation*. Microsoft. 2018. URL: https://www.microsoft.com/security/blog/2018/06/05/virtualization-based-security-vbs-memory-enclaves-data-protection-through-isolation/ (cit. on p. 11).

[Res18] ESET Researchers. *LoJax: First UEFI rootkit found in the wild, courtesy of the Sednit group*. Tech. rep. ESET, Sept. 2018 (cit. on p. 13).

[Sha+18] 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. 38, 41, 47).

[SUS18] SUSE. *snapper*. 2018. URL: http://snapper.io/ (cit. on p. 48).

[Tre18] 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/ (cit. on p. 49).

[WEP18] 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. 38, 51).

[Ard+17] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner. "Register transfer level information flow tracking for provably secure hardware design." In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 2017, pp. 1691–1696 (cit. on p. 36).

[Bul+17] 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 p. 14).

[Bur+17b] 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 p. 16).

[Cho+17] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. "Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification." In: 1.ICFP (Aug. 2017). DOI: 10.1145/3110268 (cit. on p. 58).

[Fer+17a] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. "Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis." In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '17. Xi'an, China: Association for Computing Machinery, 2017, pp. 555–568 (cit. on p. 36).

[Fer+17b] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. "Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis." In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '17. Xi'an, China: Association for Computing Machinery, 2017, pp. 555–568. ISBN: 9781450344654. DOI: 10.1145/3037697.3037739 (cit. on p. 58).

[Geo+17] Laurent Georget, Mathieu Jaume, Guillaume Piolle, Frédéric Tronel, and Valérie Viet Triem Tong. "Information Flow Tracking for Linux Handling Concurrent System Calls and Shared Memory." In: *SEFM*. Vol. 10469. Lecture Notes in Computer Science. Springer, 2017, pp. 1–16 (cit. on p. 22).

[Hod17] Daniel Hodson. *Remote LD_PRELOAD Exploitation*. Dec. 18, 2017. URL: https://www.elttam.com.au/blog/goahead/ (cit. on p. 50).

[Int17] 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 (cit. on p. 16).

[Jah+17] 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 p. 49).

[Lep+17] 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. 16).

[The17] The coreboot community. *coreboot*. 2017. URL: https://www.coreboot.org/ (cit. on p. 19).

[Tia17] Tianocore. *EDK II*. 2017. URL: http://www.tianocore.org/edk2/ (cit. on p. 19).

[Wan+17] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2421–2434. ISBN: 9781450349468. DOI: 10.1145/3133956.3134038 (cit. on p. 11).

[YZ17] Jiewen Yao and Vincent J. Zimmer. *A Tour Beyond BIOS - Memory Protection in UEFI BIOS*. Tech. rep. Intel, Mar. 2017 (cit. on p. 39).

[Cor16] The MITRE Corporation. *CVE-2016-8103*. Sept. 2016. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8103 (cit. on pp. 14, 20).

[Dr 16] Dr. Web. *Linux.Rex.1*. Aug. 2016. URL: https://vms.drweb.com/virus/?i=8436299 (cit. on p. 49).

[Kor+16] Lazaros Koromilas, Giorgos Vasiliadis, Ilias Athanasopoulos, and Sotiris Ioannidis. "GRIM: Leveraging GPUs for Kernel integrity monitoring." English. In: *Research in Attacks, Intrusions, and Defenses - 19th International Symposium, RAID 2016, Proceedings*. Vol. 9854 LNCS. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Springer/Verlag, 2016, pp. 3–23. ISBN: 9783319457185. DOI: 10.1007/978-3-319-45719-2_1 (cit. on p. 12).

[Len16a] Lenovo. *Lenovo Security Advisory: LEN-4710*. Sept. 2016. URL: https://support.lenovo.com/us/en/product_security/len_4710 (cit. on pp. 14, 20).

[Len16b] Lenovo. *Lenovo Security Advisory: LEN-8324*. Nov. 2016. URL: https://support.lenovo.com/us/en/solutions/len-8324 (cit. on pp. 14, 20).

[Ole16a] 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 (cit. on pp. 14, 20).

[Ole16b] Dmytro Oleksiuk. *Exploring and exploiting Lenovo firmware secrets*. June 2016. URL: http://blog.cr4.sh/2016/06/exploring-and-exploiting-lenovo.html (cit. on pp. 14, 20).

[Puj16] Bruno Pujos. *SMM unchecked pointer vulnerability*. May 2016. URL: http://esec-lab.sogeti.com/posts/2016/05/30/smm-unchecked-pointer-vulnerability.html (cit. on pp. 14, 20).

[Rei16] Alastair Reid. "Trustworthy Specifications of ARM v8-A and v8-M System Level Architecture." In: *FMCAD'16*. 2016 (cit. on p. 58).

[Rei+16] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. "End-to-End Verification of Processors with ISA-Formal." In: *Computer Aided Verification*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Cham: Springer International Publishing, 2016, pp. 42–58. ISBN: 978-3-319-41540-6 (cit. on p. 58).

[Son+16] 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. 40).

[BAL15b] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. "A Survey on Hypervisor-Based Monitoring: Approaches, Applications, and Evolutions." In: *ACM Comput. Surv.* 48.1 (Aug. 2015). ISSN: 0360-0300. DOI: 10.1145/2775111 (cit. on pp. 12, 75, 76).

[Baz+15] 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 p. 14).

[Dha+15] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight Jr., Benjamin C. Pierce, and Andre DeHon. "Architectural Support for Software-Defined Metadata Processing." In: *SIGARCH Comput. Archit. News* 43.1 (Mar. 2015), pp. 487–502. ISSN: 0163-5964. DOI: 10.1145/2786763.2694383 (cit. on p. 23).

[Dr 15] Dr. Web. *Linux.Encoder.1*. Nov. 2015. URL: https://vms.drweb.com/virus/?i=7703983 (cit. on p. 50).

[Heo15] Tejun Heo. *Control Group v2*. Oct. 2015. URL: https://www.kernel.org/doc/Documentation/cgroup-v2.txt (cit. on pp. 40, 48).

[Int15] Intel. *Intel Trusted Execution Technology (Intel TXT)*. July 2015 (cit. on p. 57).

[KK15] Xeno Kovah and Corey Kallenberg. "How Many Million BIOSes Would you Like to Infect?" (Vancouver, B.C., Canada). CanSecWest. 2015 (cit. on p. 14).

[Kov+15] Xeno Kovah, Corey Kallenberg, John Butterworth, and Sam Cornwell. "SENTER Sandman: Using Intel TXT to Attack Bioses." In: *Hack in the Box* (2015) (cit. on p. 57).

[Lee+15a] Jinyong Lee, Ingoo Heo, Yongje Lee, and Yunheung Paek. "Efficient Dynamic Information Flow Tracking on a Processor with Core Debug Interface." In: *DAC '15*. 2015 (cit. on p. 12).

[Lee+15b] Jinyong Lee, Yongje Lee, Hyungon Moon, Ingoo Heo, and Yunheung Paek. "Extrax: Security Extension to Extract Cache Resident Information for Snoop-based External Monitors." In: *DATE '15*. 2015 (cit. on p. 12).

[Lee+15c] 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*. HASP '15. Portland, Oregon: Association for Computing Machinery, 2015. ISBN: 9781450334839. DOI: 10.1145/2768566.2768569 (cit. on p. 12).

[PaX15] PaX Team. "RAP: RIP ROP." H2HC. Oct. 2015 (cit. on p. 16).

[Wil15] Dick Wilkins. "UEFI Firmware – Securing SMM." UEFI PlugFest. May 2015 (cit. on p. 19).

[YZ15] 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. 19, 39).

[YZF15] Jiewen Yao, Vincent J. Zimmer, and Matt Flemming. *A Tour Beyond BIOS Memory Practices in UEFI*. Tech. rep. Intel, June 2015 (cit. on p. 19).

[Aza+14] 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 p. 40).

[Bar14] Sean Barnum. "Standardizing cyber threat intelligence information with the Structured Threat Information eXpression (STIX)." Feb. 2014 (cit. on pp. 41, 46).

[Bul+14] Yuriy Bulygin, J Loucaides, Andrew Furtak, O Bazhaniuk, and A Matrosov. "Summary of Attacks Against BIOS and Secure Boot." In: *Proceedings of the DefCon* (2014) (cit. on p. 57).

[Int14] Intel. *Intel 64 and IA32 Architectures Software Developer Manual*. Oct. 2014 (cit. on p. 65).

[Jai+14] 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. 12, 75).

[Jan+14] 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 p. 19).

[NT14] 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 p. 16).

[Rua14] 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 p. 39).

[SCH14] 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 p. 41).

[Tic+14] 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 p. 16).

[Tim+14] Michaël Timbert, Jean-Luc Danger, Sylvain Guilley, Thibault Porteboeuf, and Florian Praden. "HCODE: Hardware-Enhanced Real-Time CFI." In: *PPREW at ACSAC 2014*. New Orleans, United States, Dec. 2014. DOI: 10.1145/2689702.2689708 (cit. on p. 12).

[YZZ14] 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 p. 13).

[Cor13] The MITRE Corporation. *CVE-2013-3582*. May 2013. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-3582 (cit. on pp. 14, 20).

[Gal13] Sean Gallagher. "Your USB cable, the spy: Inside the NSA's catalog of surveillance magic." In: *Ars Technica* (Dec. 31, 2013) (cit. on p. 13).

[Int13] Intel. *Desktop 4th Generation Intel Core Processor Family, Desktop Intel Pentium Processor Family, and Desktop Intel Celeron Processor Family*. Dec. 2013 (cit. on p. 65).

[Kal+13] Corey Kallenberg, John Butterworth, Xeno Kovah, and C Cornwell. "Defeating Signed BIOS Enforcement." EkoParty, Buenos Aires. 2013 (cit. on p. 20).

[Ker13] Michael Kerrisk. "Namespaces in operation, part 1: namespaces overview." In: *LWN* (Jan. 4, 2013) (cit. on p. 48).

[Lee+13a] 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: *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 511–526. ISBN: 978-1-931971-03-4 (cit. on p. 12).

[Lin13] 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/ (cit. on p. 13).

[Noo+13] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. "Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base." In: *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 479–498. ISBN: 978-1-931971-03-4 (cit. on p. 11).

[Owu+13] Emmanuel Owusu, Jorge Guajardo, Jonathan McCune, Jim Newsome, Adrian Perrig, and Amit Vasudevan. "OASIS: On Achieving a Sanctuary for Integrity

and Secrecy on Untrusted Platforms." In: CCS '13. Berlin, Germany: Association for Computing Machinery, 2013, pp. 13–24. ISBN: 9781450324779. DOI: 10.1145/2508859.2516678 (cit. on p. 11).

[RBM13] 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. 48).

[Fog12] Agner Fog. "The Microarchitecture of Intel, AMD and VIA CPUs: An Optimization Guide for Assembly Programmers and Compiler Makers." In: *Copenhagen University College of Engineering* (2012) (cit. on p. 55).

[Moo+12a] 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 2012 ACM Conference on Computer and Communications Security*. CCS '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 28–37. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382202 (cit. on p. 12).

[RSI12] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Part 2*. 6th ed. Microsoft Press, 2012. ISBN: 978-0735665873 (cit. on p. 40).

[Bin+11] 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. 19).

[Coo+11] 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 p. 13).

[Den+11] 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. 47).

[Int11] Intel Corporation. *bits-365*. Mar. 2011. URL: https://biosbits.org/news/bits-365/ (cit. on p. 21).

[Kir+11] Ivan Kirillov, Desiree Beck, Penny Chase, and Robert Martin. "Malware Attribute Enumeration and Characterization." 2011 (cit. on pp. 41, 43).

[Liv11] Ge Livian. *BIOS Threat is Showing up Again!* Sept. 2011. URL: https://www.symantec.com/connect/blogs/bios-threat-showing-again (cit. on p. 13).

[Ltd11] ARM Ltd. *CoreSight Program Flow Trace - Architecture Specification*. 2011 (cit. on p. 28).

[MC11] Scott Moore and Stephen Chong. "Static Analysis for Efficient Hybrid Information-Flow Control." In: *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*. IEEE Computer Society, 2011, pp. 146–160. DOI: 10.1109/CSF.2011.17 (cit. on p. 74).

[Tho11] 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. 54).

[Tru11] Trusted Computing Group. *TPM Main, Part 1 Design Principles*. Trusted Computing Group. Mar. 2011 (cit. on p. 13).

[Che+10b] H. Cheng, Y. Hwang, R. Chang, and C. Chen. "Trading Conditional Execution for More Registers on ARM Processors." In: *2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*. 2010, pp. 53–59 (cit. on p. 29).

[CS10] Michael R Clarkson and Fred B Schneider. "Hyperproperties." In: *Journal of Computer Security* 18.6 (2010), pp. 1157–1210 (cit. on pp. 64, 74).

[Khe+10] 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 p. 41).

[Kim+10] 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 p. 38).

[WSG10] Jiang Wang, Angelos Stavrou, and Anup Ghosh. "HyperCheck: A Hardware-Assisted Integrity Monitor." In: *Recent Advances in Intrusion Detection*. Ed. by Somesh Jha, Robin Sommer, and Christian Kreibich. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 158–177. ISBN: 978-3-642-15512-3 (cit. on p. 12).

[ZRM10] 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 p. 13).

[ARM09] ARM. *ARM Security Technology: Building a Secure System using TrustZone Technology*. ARM. Apr. 2009 (cit. on p. 14).

[Cor09] Jonathan Corbet. "Seccomp and sandboxing." In: *LWN* (May 13, 2009) (cit. on p. 48).

[Int09a] Intel. *Intel 5100 Memory Controller Hub Chipset*. July 2009 (cit. on p. 65).

[Int09b] Intel Corporation. "Introduction to the Intel Quickpath Interconnect." Jan. 2009 (cit. on p. 15).

[KDK09a] H. Kannan, M. Dalton, and C. Kozyrakis. "Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor." In: *2009 IEEE/IFIP International Conference on Dependable Systems Networks*. 2009, pp. 105–114. DOI: 10.1109/DSN.2009.5270347 (cit. on p. 12).

[KDK09b] H. Kannan, M. Dalton, and C. Kozyrakis. "Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor." In: *2009 IEEE/IFIP International Conference on Dependable Systems Networks*. June 2009. DOI: 10.1109/DSN.2009.5270347 (cit. on pp. 23, 24).

[Loi+09] Loic Duflot, Olivier Levillain, Benjamin Morin, and Olivier Grumelard. "Getting into the SMRAM: SMM Reloaded." In: (Mar. 2009). CanSecWest (cit. on pp. 57, 66).

[Pal+09] 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. 49).

[RJ09] Rafal Wojtczuk and Joanna Rutkowska. "Attacking SMM Memory via Intel CPU Cache Poisoning." In: (Mar. 2009) (cit. on p. 66).

[Roy+09] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. "Laminar: Practical Fine-grained Decentralized Information Flow Control." In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland: ACM,

2009, pp. 63–74. ISBN: 978-1-60558-392-1. DOI: 10.1145/1542476.1542484 (cit. on p. 22).

[WT09] Rafal Wojtczuk and Alexander Tereshkin. "Attacking Intel BIOS." Black Hat USA. July 2009 (cit. on p. 14).

[XJL09] 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. 38, 51).

[Ant08] 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. 46).

[BS08a] Yuriy Bulygin and David Samyde. "Chipset based approach to detect virtualization malware." Black Hat USA. 2008 (cit. on p. 12).

[Che+08a] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. "Flexible Hardware Acceleration for Instruction-Grain Program Monitoring." In: *SIGARCH Comput. Archit. News* (June 2008). ISSN: 0163-5964. DOI: 10.1145/1394608.1382153 (cit. on p. 23).

[Che+08b] 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. 49).

[Hol+08] 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 p. 16).

[McC+08] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. "Flicker: An Execution Infrastructure for TCB Minimization." In: Eurosys '08. Glasgow, Scotland UK: Association for Computing Machinery, 2008, pp. 315–328. ISBN: 9781605580135. DOI: 10.1145/1352592.1352625 (cit. on p. 11).

[Ruw+08] Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. "Parallelizing Dynamic Information Flow Tracking." In: *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*. SPAA '08. Munich, Germany: ACM, 2008, pp. 35–45. ISBN: 978-1-59593-973-9 (cit. on p. 23).

[Ven+08] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. "FlexiTaint: A programmable accelerator for dynamic taint propagation." In: *2008 IEEE 14th HPCA*. Feb. 2008, pp. 173–184. DOI: 10.1109/HPCA.2008.4658637 (cit. on p. 24).

[Zel08] Nickolai Zeldovich. "Securing Untrustworthy Software Using Information Flow Control." AAI3292438. PhD thesis. Stanford, CA, USA, 2008. ISBN: 978-0-549-35732-2 (cit. on p. 22).

[DKK07] Michael Dalton, Hari Kannan, and Christos Kozyrakis. "Raksha: A Flexible Information Flow Architecture for Software Security." In: *SIGARCH Comput. Archit. News* 35.2 (June 2007), pp. 482–493. ISSN: 0163-5964 (cit. on p. 23).

[DCF07] H. Debar, D. Curry, and B. Feinstein. *The Intrusion Detection Message Exchange Format (IDMEF)*. RFC 4765 (Experimental). Internet Engineering Task Force, Mar. 2007 (cit. on p. 41).

[CCH06] 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 p. 35).

[Hsu+06] 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 p. 38).

[Koc06] Hans-Jürgen Koch. *The Userspace I/O HOWTO*. 2006. URL: https://www.kernel.org/doc/html/v4.13/driver-api/uio-howto.html (cit. on p. 31).

[Pet+06a] Nick L. Petroni, 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 Conference on USENIX Security Symposium - Volume 15*. USENIX-SS'06. Vancouver, B.C., Canada: USENIX Association, 2006 (cit. on p. 12).

[Qin+06] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. "LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks." In: *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 39. USA: IEEE Computer Society, 2006, pp. 135–148. ISBN: 0769527329. DOI: 10.1109/MICRO.2006.29 (cit. on p. 23).

[Bel05] 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. 19).

[Che+05] 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. 35).

[Efs+05] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. "Labels and Event Processes in the Asbestos Operating System." In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. SOSP '05. Brighton, United Kingdom: ACM, 2005, pp. 17–30. ISBN: 1-59593-079-5. DOI: 10.1145/1095810.1095813 (cit. on p. 22).

[Foo+05] 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. 38, 41).

[Goe+05] 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 p. 38).

[NS05] James Newsome and Dawn Song. "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software." In: *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS 2005)*. San Diego, CA, Feb. 2005 (cit. on p. 23).

[Avi+04] 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 p. 38).

[LA04] 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 p. 19).

[MA04] 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. 46, 47).

[Pet+04a] Nick L. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. "Copilot - a Coprocessor-Based Kernel Runtime Integrity Monitor." In: SSYM'04. San Diego, CA: USENIX Association, 2004, p. 13 (cit. on p. 12).

[Suh+04] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. "Secure Program Execution via Dynamic Information Flow Tracking." In: *SIGARCH Comput. Archit. News* 32.5 (Oct. 2004), pp. 85–96. ISSN: 0163-5964 (cit. on p. 23).

[KSS03] 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. 38).

[PC03] 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. 18).

[Mar+02] Debbie Marr, Frank Binns, D Hill, Glenn Hinton, D Koufaty, et al. "Hyper-Threading Technology in the Netburst® Microarchitecture." In: *14th Hot Chips* (2002) (cit. on p. 55).

[MMK02] Milena Milenkovic, Aleksandar Milenkovic, and Jeffrey Kulick. "Demystifying Intel Branch Predictors." In: *Workshop on Duplicating, Deconstructing and Debunking*. 2002 (cit. on p. 55).

[Zha+02a] 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*. EW 10. Saint-Emilion, France: Association for Computing Machinery, 2002, pp. 239–242. DOI: 10.1145/1133373.1133423 (cit. on pp. 11, 12).

[Dye+01] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, and S. W. Smith. "Building the IBM 4758 secure coprocessor." In: *Computer* 34.10 (2001), pp. 57–66. DOI: 10.1109/2.955100 (cit. on p. 11).

[Sch00a] Fred B. Schneider. "Enforceable Security Policies." In: *ACM Transactions on Information and System Security (TISSEC)* 3.1 (2000), pp. 30–50 (cit. on p. 63).

[Ell+97] 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 p. 38).

[Loi+95] Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, Saddek Bensalem, and David Probst. "Property Preserving Abstractions for the Verification of Concurrent Systems." In: *Formal methods in system design* 6.1 (1995), pp. 11–44 (cit. on p. 59).

[Lam85] Leslie Lamport. "Logical Foundation." In: *Distributed systems-methods and tools for specification* 190 (1985), pp. 119–130 (cit. on p. 63).

[GM82] Joseph A Goguen and José Meseguer. "Security Policies and Security Models." In: *Security and Privacy, 1982 IEEE Symposium on*. IEEE. 1982, pp. 11–11 (cit. on p. 63).

Publications

Conference proceedings

[1] **Camille Le Bon**, Erven Rohou, Frédéric Tronel, and **Guillaume Hiet**. "DAMAS: Control-Data Isolation at Runtime through Dynamic Binary Modification." In: *Workshop on the Security of Software / Hardware Interfaces (SILM 2021)*. digital event, France, Sept. 2021 (cit. on p. 52).

[2] **Ronny Chevalier**, David Plaquin, Chris Dalton, and **Guillaume Hiet**. "Survivor: A Fine-Grained Intrusion Response and Recovery Approach for Commodity Operating Systems." In: *ACSAC 2019 - 35th Annual Computer Security Applications Conference*. Vol. 2019. Proceedings of the 35th Annual Computer Security Applications Conference. San Juan, Puerto Rico, Dec. 2019. DOI: 10.1145/3359789.3359792 (cit. on p. 52).

[3] Jean-François Lalande, Valérie Viet Triem Tong, Pierre Graux, **Guillaume Hiet**, Wojciech Mazurczyk, Habiba Chaoui, and Pascal Berthomé. "Teaching Android Mobile Security." In: *SIGCSE '19 - 50th ACM Technical Symposium on Computer Science Education*. Proceedings of the 50th ACM Technical Symposium on Computer Science Education. Minneapolis, United States: ACM Press, Feb. 2019, pp. 232–238. DOI: 10.1145/3287324.3287406.

[4] **Kévin Le Bon**, Byron Hawkins, Erven Rohou, **Guillaume Hiet**, and Frédéric Tronel. "Plateforme de protection de binaires configurable et dynamiquement adaptative." In: *RESSI 2019 - Rendez-Vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information*. Erquy, France, May 2019, pp. 1–3.

[5] **Muhammad Abdul Wahab**, Pascal Cotret, **Mounir Nasr Allah**, **Guillaume Hiet**, Arnab Kumar Biswas, Vianney Lapotre, and Gogniat Guy. "A small and adaptive coprocessor for information flow tracking in ARM SoCs." In: *ReConFig 2018 - International Conference on Reconfigurable Computing and FPGAs*. Proceedings of the 2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig). Cancun, Mexico, Dec. 2018, pp. 1–17. DOI: 10.1109/reconfig.2018.8641695 (cit. on p. 35).

[6] **Muhammad Abdul Wahab**, Pascal Cotret, **Mounir Nasr Allah**, **Guillaume Hiet**, Vianney Lapotre, Gogniat Guy, and Arnab Kumar Biswas. "A novel lightweight hardware-assisted static instrumentation approach for ARM SoC using debug components." In: *AsianHOST 2018 - Asian Hardware Oriented Security and Trust Symposium*. Proceedings of the 2018 Asian Hardware Oriented Security and Trust Symposium (AsianHOST). Hong Kong, China, Dec. 2018, pp. 1–13. DOI: 10.1109/asianhost.2018.8607177 (cit. on p. 35).

[7] **Ronny Chevalier**, David Plaquin, and **Guillaume Hiet**. "Intrusion Survivability for Commodity Operating Systems and Services: A Work in Progress." In: *RESSI 2018 - Rendez-Vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information*. Nancy / La Bresse, France, May 2018 (cit. on p. 52).

[8] **Oualid Koucham**, Stéphane Mocanu, **Guillaume Hiet**, Jean-Marc Thiriet, and Frédéric Majorczyk. "Efficient Mining of Temporal Safety Properties for Intrusion Detection in Industrial Control Systems." In: *SAFEPROCESS 2018 - 10th IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes*. Warsaw, Poland, Aug. 2018, pp. 1–8 (cit. on p. 9).

[9] **Thomas Letan**, Yann Régis-Gianas, Pierre Chifflier, and **Guillaume Hiet**. "Modular Verification of Programs with Effects and Effect Handlers in Coq." In: *FM 2018 - 22nd International Symposium on Formal Methods*. Vol. 10951. LNCS. Oxford, United Kingdom: Springer, July 2018, pp. 338–354. DOI: 10.1007/978-3-319-95582-7_20 (cit. on p. 73).

[10] **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: *ACSAC 2017 - 33rd Annual Computer Security Applications Conference*. Vol. 2017. Proceedings of the 33rd Annual Computer Security Applications Conference. Orlando, United States: ACM, Dec. 2017, pp. 399–411. DOI: 10.1145/3134600.3134622 (cit. on p. 22).

[11] **Muhammad Abdul Wahab**, Pascal Cotret, **Mounir Nasr Allah**, **Guillaume Hiet**, Vianney Lapotre, and Guy Gogniat. "ARMHEx: A hardware extension for DIFT on ARM-based SoCs." In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. Ghent, Belgium, Sept. 2017. DOI: 10.23919/fpl.2017.8056767 (cit. on p. 35).

[12] **Muhammad Abdul Wahab**, Pascal Cotret, **Mounir Nasr Allah**, **Guillaume Hiet**, Vianney Lapotre, and Guy Gogniat. "ARMHEx: embedded security through hardware-enhanced information flow tracking." In: *RESSI 2017 : Rendez-vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information*. Grenoble (Autrans), France, May 2017 (cit. on p. 35).

[13] **Muhammad Abdul Wahab**, Pascal Cotret, **Mounir Nasr Allah**, **Guillaume Hiet**, Vianney Lapotre, and Guy Gogniat. "A portable approach for SoC-based Dynamic Information Flow Tracking implementations." In: *11ème Colloque du GDR SoC/SiP*. Nantes, France, June 2016.

[14] **Muhammad Abdul Wahab**, Pascal Cotret, **Mounir Nasr Allah**, **Guillaume Hiet**, Vianney Lapotre, and Guy Gogniat. "Towards a hardware-assisted information flow tracking ecosystem for ARM processors." In: *26th International Conference on Field-Programmable Logic and Applications (FPL 2016)*. Lausanne, Switzerland, Aug. 2016. DOI: 10.1109/fpl.2016.7577396.

[15] **Oualid Koucham**, Stéphane Mocanu, **Guillaume Hiet**, Jean-Marc Thiriet, and Frédéric Majorczyk. "Classification des approches de détection d'intrusions dans les systèmes de contrôle industriels et axes d'amélioration." In: *Rendez-Vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information (RESSI 2016)*. Toulouse, France, May 2016.

[16] **Oualid Koucham**, Stéphane Mocanu, **Guillaume Hiet**, Jean-Marc Thiriet, and Frédéric Majorczyk. "Detecting Process-Aware Attacks in Sequential Control Systems." In: *21st Nordic Conference on Secure IT Systems (NordSec 2016)*. Oulu, Finland, Nov. 2016, p.20–36. DOI: 10.1007/978-3-319-47560-8_2 (cit. on p. 9).

[17] **Thomas Letan**, Pierre Chifflier, **Guillaume Hiet**, Pierre Néron, and Benjamin Morin. "SpecCert: Specifying and Verifying Hardware-based Software Enforcement." In: *21st International Symposium on Formal Methods (FM 2016)*. 21st International Symposium

on Formal Methods (FM 2016). Limassol, Cyprus: Springer, Nov. 2016. DOI: 10.1007/978-3-319-48989-6_30 (cit. on p. 73).

[18] **Deepak Subramanian**, **Guillaume Hiet**, and Christophe Bidan. "A self-correcting information flow control model for the web-browser." In: *FPS 2016 - The 9th International Symposium on Foundations & Practice of Security*. Vol. 10128. Lecture Notes in Computer Science. Québec City, Canada, Oct. 2016, pp. 285–301. DOI: 10.1007/978-3-319-51966-1_19 (cit. on p. 9).

[19] **Deepak Subramanian**, **Guillaume Hiet**, and Christophe Bidan. "Preventive Information Flow Control through a Mechanism of Split Addresses." In: *9th International Conference on Security of Information and Networks (SIN 2016)*. SIN '16 Proceedings of the 9th International Conference on Security of Information and Networks. Newark, United States, July 2016, p.1–8. DOI: 10.1145/2947626.2947645 (cit. on p. 9).

[20] **Guillaume Hiet**, Hervé Debar, Sélim Ménouar, and Vérène Houdebine. "Etude comparative des formats d'alertes." In: *C&ESAR (Computer & Electronics Security Applications Rendez-vous) 2015*. Actes de la conférence C&ESAR (Computer & Electronics Security Applications Rendez-vous) 2015. Rennes, France, Nov. 2015, pp. 125–148.

[21] **Georges Bossert**, Frédéric Guihéry, and **Guillaume Hiet**. "Towards Automated Protocol Reverse Engineering Using Semantic Information." In: *ASIA CCS '14*. 12 pages. Kyoto, Japan, June 2014, pp. 51–62. DOI: 10.1145/2590296.2590346 (cit. on p. 9).

[22] **Deepak Subramanian**, **Guillaume Hiet**, and Christophe Bidan. "Preventive Information Flow Control through a Mechanism of Split Addresses." In: *9ème Conférence sur la Sécurité des Architectures Réseaux et des Systèmes d'Information*. Saint-Germain-Au-Mont-d'Or, France, May 2014.

[23] **Georges Bossert**, Frédéric Guihéry, and **Guillaume Hiet**. "Netzob : un outil pour la rétro-conception de protocoles de communication." In: *SSTIC 2012*. Rennes, France, June 2012, p. 43.

[24] Emmanuelle Anceaume, Christophe Bidan, Sébastien Gambs, **Guillaume Hiet**, Michel Hurfin, Ludovic Mé, Guillaume Piolle, Nicolas Prigent, Eric Totel, Frédéric Tronel, and Valérie Viet Triem Tong. "From SSIR to CIDre: a New Security Research Group in Rennes." In: *1st SysSec Workshop*. Amsterdam, Netherlands, July 2011.

[25] **Georges Bossert**, **Guillaume Hiet**, and Thibaut Henin. "Modelling to Simulate Botnet Command and Control Protocols for the Evaluation of Network Intrusion Detection Systems." In: *SAR-SSI 2011*. La Rochelle, France, May 2011, pp. 1–8. DOI: 10.1109/SAR-SSI.2011.5931397.

[26] **Guillaume Hiet**, Frédéric Guihéry, Goulven Guiheux, David Pichardie, and Christian Brunette. "Sécurité de la plate-forme d'exécution Java : limites et propositions d'améliorations." In: *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*. Rennes, France, 2010.

[27] **Guillaume Hiet**, Valérie Viet Triem Tong, Ludovic Mé, and Benjamin Morin. "Policy-Based Intrusion Detection in Web Applications by Monitoring Java Information Flows." In: *CRiSIS 2008*. Tozeur, Tunisia, Oct. 2008, 8 pages.

[28] **Guillaume Hiet**, Ludovic Mé, Benjamin Morin, and Valérie Viet Triem Tong. "Monitoring both OS and program level information flows to detect intrusions against network servers." In: *IEEE Workshop on Monitoring, Attack Detection and Mitigation*. Toulouse, France, Nov. 2007, unknown.

[29] **Guillaume Hiet**, Ludovic Mé, Jacob Zimmermann, Christophe Bidan, Benjamin Morin, and Valérie Viet Triem Tong. "Détection fiable et pertinente de flux d'information illé-

gaux." In: *Sixth Conference on Security and Network Architectures (SARSSI)*. France, June 2007, unknown.

Journals

[30] **Ronny Chevalier**, David Plaquin, Chris Dalton, and **Guillaume Hiet**. "Intrusion Survivability for Commodity Operating Systems." In: *Digital Threats: Research and Practice* 1.4 (Dec. 2020), pp. 1–30. DOI: 10.1145/3419471 (cit. on p. 52).

[31] **Thomas Letan**, Yann Régis-Gianas, Pierre Chifflier, and **Guillaume Hiet**. "Modular verification of programs with effects and effects handlers." In: *Formal Aspects of Computing* (Dec. 2020). DOI: 10.1007/s00165-020-00523-2 (cit. on p. 73).

[32] Mathieu Jaume, Valérie Viet Triem Tong, and **Guillaume Hiet**. "Spécification et mécanisme de détection de flots d'information illégaux." In: *Revue des Sciences et Technologies de l'Information - Série TSI : Technique et Science Informatiques* 31.6 (2012), pp. 713–742. DOI: 10.3166/tsi.31.713-742.

[33] **Guillaume Hiet**, Valérie Viet Triem Tong, Ludovic Mé, and Benjamin Morin. "Policy-based intrusion detection in web applications by monitoring Java information flows." In: *International Journal of Information and Computer Security* Vol.3.N°3/4 (2009), 15 pages.

[34] Christophe Bidan, **Guillaume Hiet**, Ludovic Mé, Benjamin Morin, and Jacob Zimmermann. "Vers une détection d'intrusions à fiabilité et pertinence prouvables." In: *La Revue de l'Electricité et de l'Electronique* 9 (Oct. 2006), 13 pages.

Posters

[35] **Muhammad Abdul Wahab**, Pascal Cotret, **Mounir Nasr Allah**, **Guillaume Hiet**, Vianney Lapotre, and Guy Gogniat. *ARMHEx: a framework for efficient DIFT in real-world SoCs*. Field Programmable Logic (FPL). Poster. Sept. 2017.

[36] Pascal Cotret, **Guillaume Hiet**, and Guy Gogniat. *HardBlare: an efficient hardware-assisted DIFC for non-modified embedded processors*. HiPEAC. Poster. Jan. 2016.

[37] **Mounir Nasr Allah**, **Guillaume Hiet**, **Muhammad Abdul Wahab**, Pascal Cotret, Guy Gogniat, and Vianney Lapotre. *HardBlare: a Hardware-Assisted Approach for Dynamic Information Flow Tracking*. Séminaire des doctorantes et doctorants en informatique de la Société Informatique de France. Poster. Apr. 2016.

[38] Pascal Cotret, **Guillaume Hiet**, Guy Gogniat, and Vianney Lapotre. *HardBlare: an efficient hardware-assisted DIFC for non-modified embedded processors*. CHES 2015 - Workshop on Cryptographic Hardware and Embedded Systems. Poster. Sept. 2015.