KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Hardware supported Software and Control Flow Integrity

Ruan de Clercq

Supervisor:
Prof. dr. ir. I. Verbauwhede

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD)

November 2017

Hardware supported Software and Control Flow Integrity

Ruan DE CLERCQ

Examination committee:
Prof. dr. ir. Omer Van der Biest, chair
Prof. dr. ir. I. Verbauwhede, supervisor
Prof. dr. ir. F. Piessens
Prof. dr. ir. B. Preneel
Prof. dr. Aurélien Francillon
(EURECOM, France)
Prof. dr. ir. Bjorn de Sutter
(University of Ghent, Belgium)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD)

November 2017

Acknowledgements

First and foremost, I would like to thank my promoter Ingrid Verbauwhede for offering me the opportunity to pursue a PhD at COSIC. I am deeply grateful for her guidance, time, tips, and trust to allow me to freely conduct my research.

I would like to thank my assessors Prof. Bart Preneel and Prof. Frank Piessens for their valuable contributions throughout the doctoral program. I would further like to thank the additional members of the jury Prof. Aurélien Francillon and Prof. Bjorn de Sutter for the time and effort that they invested in this dissertation, and Prof. Patrick Wollants for chairing the jury.

Thanks to all my co-authors for the fruitful discussions and collaborations. I learned a lot from you, and would like to continue writing papers with you.

I would like to thank all my colleagues in COSIC for contributing to a great research environment, and also for the lunches, karting, table tennis, coffee breaks, COSIC weekends, Friday beers, and climbing sessions. I have thoroughly enjoyed my time here and I can highly recommend working at COSIC.

I am forever grateful to all my friends in Leuven for their friendship, support, and patience that made my time in Leuven an unforgettable experience. Special thanks to my climbing friends for the many amazing sessions in Freyr, Mozet, Berdorf, Fontainebleau and further afield.

I want to thank my partner and family for their ongoing support, encouragement, and for always being there when I need them. Finally, I am eternally grateful to my mom and my partner's parents for their continued support.

Ruan de Clercq
Leuven, November 2017

Abstract

Bugs are prevalent in a large amount of deployed software. These bugs often introduce vulnerabilities that can be exploited by attackers to make programs misbehave. Many devices rely on software that needs security, such as medical implants, sensor networks, RFID tags, automotive controllers. Software should do what it is asked to do, and should not misbehave; e.g., by delivering the wrong drug dosages, by stealing information, by spying on the user, by disabling the brakes on a car, or by attacking other computers.

The central topic of this thesis is the development of hardware-based mechanisms that prevent software from misbehaving. We focus on enhancing the security of microprocessors to detect runtime attacks, prevent malicious modification of software, and develop support for isolating software from malware.

The main contributions of this thesis are two-fold. First, we analyse existing hardware-based Control Flow Integrity (CFI) architectures. This includes a detailed description and comparison of each architecture's policies, security, hardware cost, performance, and suitability for widespread deployment.

Second, we design several new hardware-based security architectures. This includes developing the first known CFI architecture based on instruction-set randomisation, that also enforces software integrity through modifications to a processor. We further design the first known hardware-based software integrity architecture that is realised as a standalone Intellectual Property (IP) core that connects to the bus via standard interfaces. Finally, we develop an architectural feature which provides interrupt support for a program counter-based Protected Module Architectures (PMAs) by means of processor modifications. All the architectures developed in this thesis are evaluated on Field Programmable Gate Array (FPGA), which allows us to accurately determine the hardware cost and the performance overhead of running the software on the architecture.

Beknopte samenvatting

Veel geïnstalleerde software bevat fouten. Dit is problematisch omdat die fouten de oorzaak zijn van zwakheden die misbruikt kunnen worden om applicaties zich te doen misdragen. Software die beveiligd moet worden wordt gebruikt op vele apparaten, zoals medische implantaten, netwerken van sensoren, RFID tags en de regelaars in voertuigen. Deze software moet doen wat het gevraagd word en mag zich niet misdragen door bijvoorbeeld de verkeerde dosis medicijnen toe te dienen, informatie te stelen, de gebruiker te bespioneren, de remmen van een auto onbruikbaar te maken of door andere computers aan te vallen.

Het centrale onderwerp van deze thesis is het ontwerp van hardware-gebaseerde mechanismen die verzekeren dat software zich niet kan misdragen. We concentreren ons op het verbeteren van de beveiliging van microprocessoren om runtime aanvallen te detecteren.

De voornaamste bijdragen van deze thesis zijn tweeledig. Ten eerste analyseren we bestaande hardware-gebaseerde architecturen die de integriteit van de programmastroom beschermen. Dit omvat een gedetailleerde beschrijving van hun richtlijnen, samen met een evaluatie van hun beveiliging, hardware kost, performantie en geschiktheid voor wijdverspreide toepassing.

Ten tweede ontwikkelen we verschillende nieuwe hardware-gebaseerde architecturen. Dit omvat de ontwikkeling van de eerste gekende architectuur die de integriteit van de programmastroom beschermt aan de hand van instructieset randomisatie. Bovendien verzekert deze architectuur de integriteit van de applicatie door middel van aanpassingen aan de processor. We hebben daarnaast ook de eerste gekende hardware-gebaseerde architectuur ontwikkeld om de integriteit van software te beschermen tijdens de uitvoering ervan aan de hand van een *IP core* die verbindt met de bus via een gestandaardiseerde interface. Tot slot ontwikkelen we een architecturale functie op basis van aanpassingen aan de processor die ondersteuning voor *interrupts* toevoegt aan architecturen die softwaremodules beschermen door geheugentoegangcontrole op basis van de

programmateller. Alle architecturen die ontwikkeld worden in deze thesis zijn geëvalueerd op *FPGA*, waardoor we accuraat de hardware kost en de impact op de performantie voor het uitvoeren van software op de architectuur kunnen bepalen.

Contents

List of Figures

List of Tables

List of Abbreviations

ASIC Application-Specific Integrated Circuit

BL Branch Limitation

BR Branch Regulation

CAM Content-Addressable Memory

CFG Control Flow Graph

CFI Control Flow Integrity

CPI Code Pointer Integrity

CRA Code Reuse Attack

ELF Executable and Linkable Format

EXE Execute

FIFO First-In, First-Out

FPGA Field Programmable Gate Array

FSM Finite State Machine

ID Instruction Decode

IF Instruction Fetch

IP Intellectual Property

ISA Instruction Set Architecture

ISRAND Instruction Set Randomization

ISR Interrupt Service Routine

IVT Interrupt Vector Table

JOP Jump-Oriented Programming

LBR Last Branch Register

LUT Look-Up Table

MA Memory Access

MAC Message Authentication Code

MG Monitoring Graph

MMIO Memory-Mapped IO

OF Operand Fetch

OS Operating System

PC Program Counter

PL Programmable Logic

PMA Protected Module Architecture

PS Processing System

PT Processor Trace

ROP Return-Oriented Programming

SCM Secure Code Memory

SCS Shadow Call Stack

SI Software Integrity

SM Signature Modeling

SoC System-on-Chip

SR Status Register

TCB Trusted Computing Base

WB Write Back

XCP Exception

Chapter 1

Introduction

Computers play an important role in today's society and will continue to play an even greater role in the future. It is important to ensure that the software that runs on these computers perform the correct computations. Software should do what it is asked to do, and should not misbehave; e.g., by stealing information, by spying on the user, by delivering the wrong drug dosages, or by attacking other computers. Therefore, to ensure that computers behave as expected, security mechanisms are required by all classes of microprocessors: from small microcontrollers to large cloud-based servers.

Software programs are frequently deployed with bugs which makes them vulnerable to attack. A root cause of this problem can be attributed to the use of unsafe programming languages, which can introduce memory errors into programs. A *memory error* is a software bug caused by invalid pointer operations, use of uninitialised variables, and memory leaks. Memory errors occur due to the use of low-level languages, such as C and C++, which trade type safety and memory safety for performance. In contrast, *memory safe* languages aim to prevent arbitrary pointer arithmetic and provides runtime array bounds checks.

Memory errors are present in a surprisingly large amount of software, since memory unsafe languages are used by many systems, such as web browsers, embedded software, firmware, libraries, and Operating System (OS) kernels. In addition, this problem also exists in some unexpected places: (1) memory safe languages often rely on OS kernels and libraries written in memory unsafe languages, and (2) memory safe languages frequently use an interpreter which is written in a memory unsafe language.

1.1 Defending against runtime attacks

In this thesis, we use the term *system* to refer to the collection of software and hardware components that are used inside a computing platform. We assume that the computing platform consists of a microprocessor, storage, and software that runs on the microprocessor.

Defending against the exploitation of existing software bugs is a difficult problem. Even though a significant effort has been made to design defences, some attack classes remain extremely difficult to defend against. Current security mechanisms are built into OSs, compilers, programming languages, and the underlying (hardware) architectures. However, the introduction of each new defence mechanism usually leads to the development of a new attack which circumvents it. This has led to an arms-race between attackers and defenders.

Page-based protection, such as $\mathrm{W} \oplus \mathrm{X}$, is a strong defence against code injection and code tampering, and is supported by most modern processors and OSs [1]. However, $\mathrm{W} \oplus \mathrm{X}$ can be circumvented by *Code Reuse Attacks (CRAs)*, which do not require any code to be injected, but instead uses existing software for malicious purposes. To defend against CRAs, *code randomization* re-arranges the address space positions of key data and code areas of a process. This makes it more difficult to launch a CRA, since the location of the code is unknown to the attacker. However, this defence can be bypassed by a number of different approaches, including brute force or an information disclosure that allows for calculating the address of a randomised memory block. Therefore, CRAs still remain an important threat that is difficult to protect against.

Control flow is a term used to describe the order in which instructions are executed inside a program. The instructions in a program are executed sequentially, unless the processor runs into an instruction that changes the control flow, such as a branch instruction.

A *Control Flow Graph (CFG)*, is a graph of the valid control flow inside a program, and is commonly used as a model of the valid control flow inside a well-behaved program. Each node in the CFG represents a *basic block*, which is a group of instructions where control flows sequentially from the first instruction to the last. Therefore, control can only flow into a basic block at the first instruction (e.g., through a branch targeting the first instruction in a basic block.), and control can only flow out of the basic block at the last instruction. This implies that only the last instruction in a basic block may induce a control flow change. In a CFG, *forward edges* are caused by jumps and calls, while *backward edges* are caused by returns. The CFG is typically generated by statically analysing the source code or the binary of a program.

Control Flow Integrity (CFI) is a security policy that prevents attackers from tampering with the control flow of a program. The observation is that a large number of attacks rely on hijacking the control flow in order to succeed. Therefore, by enforcing a strict control flow policy, CFI can prevent these control flow hijacking attacks. CFI architectures typically assume that the software being protected contains vulnerabilities which can be exploited by an attacker, and that it is the responsibility of the CFI architecture to detect abnormal control flow.

Software Integrity (SI) is a security policy that prevents the execution of tampered software on the processor. The goal is to prevent an attacker from executing malicious code (code injection / tampering). One approach is to verify the authenticity of code before execution. Another approach is to make use of access control to isolate software. This mechanism can also exist independently of an operating system.

A *Protected Module Architecture (PMA)* is a security mechanism that allows for the secure execution of sensitive code in an area that is isolated from other processes. It operates independently of the operating system, which allows it to provide strong isolation guarantees, even when the system is infected with malware. In addition, some PMAs provide support for remote attestation, which allows a third party to determine if a device is in a trusted state.

1.2 Thesis objectives

The main objective of this thesis is to analyse and design security mechanisms that ensure that the software running on a computer processor is behaving correctly. We focus on developing new architectural security features for microprocessors. In other words, we aim to provide security from the processor itself, instead of doing so through the software. We study the modifications required to make the processor architecture support different security requirements. The focus is on small embedded processors, since they are easier to understand, use, and modify.

All the new security solutions proposed in this thesis are evaluated on an FPGA. This allows us to accurately determine the hardware cost, the performance overhead for interfacing with the hardware, the performance overhead of running the software on the modified processor, and the hardware area overhead. In contrast, many previous works rely on simulation-based evaluations, which only provides a functional evaluation, together with a rough estimate of the performance costs.

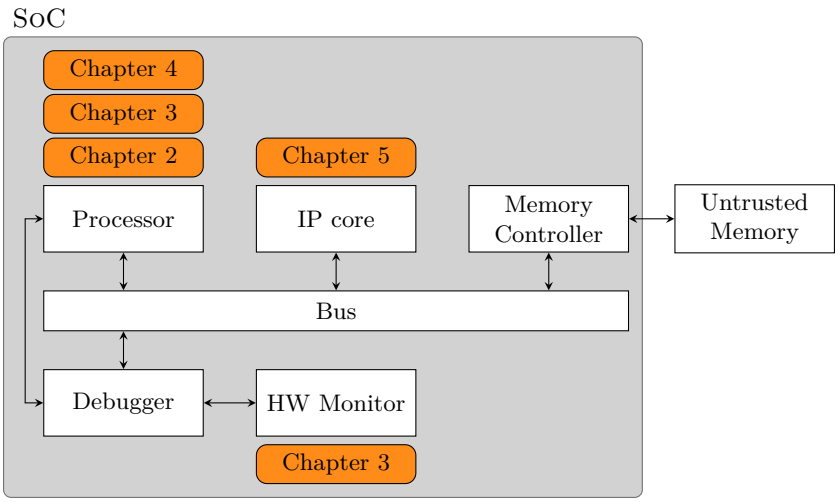

Figure 1.1: A System-on-Chip (SoC) showing the security features proposed and analysed in this thesis.

1.3 Summary of Contributions

Figure 1.1 illustrates the contributions of this thesis by highlighting the location of the security mechanisms inside a typical System-on-Chip (SoC) architecture. In summary, the contributions of this thesis are as follows:

- An architecture to provide interrupt support for Protected Module Architectures (PMAs), which requires modifying the processor (Chapter 2).
- An analysis of existing hardware-based Control Flow Integrity (CFI) policies and architectures proposed by industry and academia, which require modifying either the processor or a hardware monitor connected to the processor's debug interface (Chapter 3).
- A novel hardware-based CFI architecture, called SOFIA, which uses cryptographic techniques to enforce CFI and SI. It is implemented as a processor core modification (Chapter 4).
- A novel hardware-based architecture, called Secure Code Memory (SCM), which enforces SI. SCM is a lightweight alternative to SOFIA with reduced functionality and is implemented as an IP core (Chapter 5).

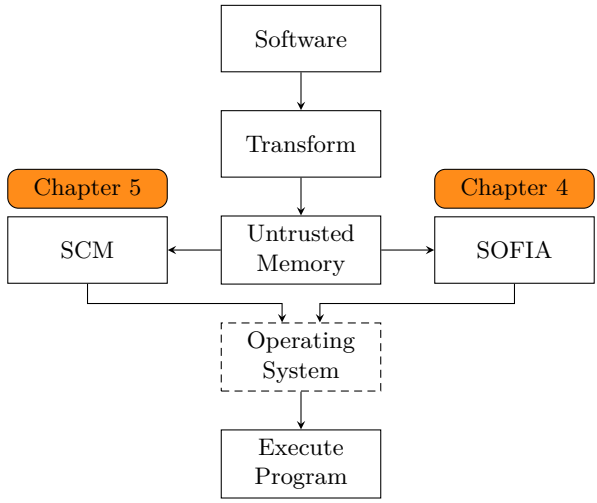

Figure 1.2: A software flowchart for SOFIA and SCM.

For SOFIA and SCM, we show a software flowchart in Figure 1.2. As a first step, a software transformation step is required, after which the transformed program is stored in untrusted memory. For SOFIA, we developed a toolchain to take care of the transformations, while for SCM we developed a scripted solution. All benchmarks were executed in baremetal, and as a future work, operating system support can be developed for these architectures.

1.4 Thesis Structure

This section provides an outline of the thesis structure.

Chapter 2 - Towards Secure Interrupts on Low-End Microcontrollers In Chapter 2 we focus on the problem of providing interrupt support to a light-weight PMA. Three methods of securely handling interrupts are proposed, each exploring a different trade-off between hardware cost, software complexity, and interrupt latency.

The content of this chapter is based on [37].

Chapter 3 - Control Flow Integrity (CFI) Policies. In Chapter 3 we introduce the concept of Control Flow Integrity (CFI), which aims to detect abnormal program behaviour. We outline the recent history of attacks and countermeasures. We provide a detailed analysis and comparison of the security

policies used by 21 state-of-the-art hardware-based CFI architectures. The security policies and architectures are compared in terms of the security properties that they provide.

The content of this chapter is part of an article that is currently under review.

Chapter 4 - Instruction-Set Randomization as a CFI policy. In Chapter 4 we introduce a security architecture called SOFIA, which is the only known architecture that enforces a CFI policy based on instruction-set randomization. The architecture is capable of defending against a large number of attacks, including code injection, code reuse, and fault-based attacks on the program counter. The architecture was evaluated on an FPGA, and a custom compiler toolchain was developed to perform the extensive software transformations required by the architecture.

The content of this chapter is based on [35] which is an extended version of [33]

Chapter 5 - A light-weight Software Integrity policy. In Chapter 5 we present an architecture that addresses the issue of protecting the integrity of code and read-only data that is stored in memory. The architecture works as a standalone IP core inside a System-on-Chip (SoC), which is a novel approach to enforce a hardware-based security policy. The architecture is also flexible to select the parts of the software to be protected, which eases the integration of our solution with existing software.

The content of this chapter is based on [34].

Chapter 6 - Conclusion. Finally, Chapter 6 we summaries the main findings of this thesis together with opportunities for future research on related topics.

1.5 Other Publications

In addition to the work published in this thesis, we worked on the following research publications which were not included in this thesis:

- In [38] we presented techniques to efficiently implement elliptic curve cryptography (ECC) on the ultra-low power ARM Cortex M0+. The paper proposed an improvement to the Lopez-Dahab field multiplication algorithm which reduces the number of memory accesses. This led to the fastest known ECC implementation on any ARM Cortex-M platform, together with the lowest energy requirement of any published microcontroller implementation with similar security parameters.

[38] de Clercq, R., Uhsadel, L., Van Herrewege, A., and Verbauwhede, I. Ultra Low-Power Implementation of ECC on the ARM Cortex-M0+. In *Proceedings of the Design Automation Conference* (2014), DAC '14, ACM, pp. 112:1–112:6

- In [36] we presented techniques to make an efficient software implementation of a post-quantum secure public-key encryption scheme based on the ring-LWE problem on the ARM Cortex-M4F. We proposed optimization techniques for fast discrete Gaussian sampling and efficient polynomial multiplication. The implementation was faster than any other published ring-LWE implementation by a factor of 7, and faster than any known ECC implementation by at least one order of magnitude.

  [36] de Clercq, R., Roy, S. S., Vercauteren, F., and Verbauwhede, I. Efficient Software Implementation of ring-LWE Encryption. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition* (2015), DATE '15, ACM, pp. 339–344

- The work in [54] proposed Soteria, which is an extension to Sancus, which provides protection of the intellectual property of code and data against powerful software attackers. The extension uses a toolchain to encrypt software IP, while at runtime a loader module decrypts the encrypted software into a software module. This ensures that code cannot leak to the outside world, while providing all the mechanisms to package and execute code.

  [54] Götzfried, J., Müller, T., de Clercq, R., Maene, P., Freiling, F., and Verbauwhede, I. Soteria: Offline Software Protection Within Low-cost Embedded Devices. In *Proceedings of the Annual Computer Security Applications Conference* (2015), ACSAC 2015, ACM, pp. 241–250

- The work in [91, 92] proposed new masking schemes to protect ring-LWE decryption against first-order side-channel attacks.

  [91] Reparaz, O., de Clercq, R., Roy, S. S., Vercauteren, F., and Verbauwhede, I. Additively homomorphic ring-LWE masking. In *International Workshop on Post-Quantum Cryptography* (2016), Springer, pp. 233–244

  [92] Reparaz, O., Roy, S. S., de Clercq, R., Vercauteren, F., and Verbauwhede, I. Masking ring-LWE. *Journal of Cryptographic Engineering 6*, 2 (2016), 139–153

- The work in [104] presents a coprocessor designed to offer hardware acceleration for a software-based VPN application. The open-source SigmaVPN application is used as the base solution, and a coprocessor is designed for the parts of Networking and Cryptography library (NaCl) which is used by SigmaVPN. The hardware-software codesign of this work is implemented on a Zynq-7000 SoC, which showed a 94% reduction in execution time for decrypting a 1024-byte Ethernet frame.

  [104] Turan, F., De Clercq, R., Maene, P., Reparaz, O., and Verbauwhede, I. Hardware Acceleration of a Software-based VPN. In *International Conference on Field Programmable Logic and Applications (FPL)* (2016), IEEE, pp. 1–9

- The work in [78] presents an analysis of the current state-of-the-art in hardware-based trusted computing architectures that provide isolation and attestation. It includes a definition of the common security properties offered by trusted computing architectures, and provides a detailed description of twelve hardware-based attestation and isolation architectures from academia and industry. In addition, the analysed architectures are compared with respect to their security properties and architectural features.

  [78] Maene, P., Götzfried, J., de Clercq, R., Muller, T., Freiling, F., and Verbauwhede, I. Hardware-Based Trusted Computing Architectures for Isolation and Attestation. *IEEE Transactions on Computers PP(99)* (2017)

Chapter 2

Towards Secure Interrupts on Low-End Microcontrollers

Content Sources

De Clercq, R., Schellekens, D., Piessens, F., and Verbauwhede, I. Secure Interrupts on Low-End Microcontrollers. In *International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (2014), IEEE, pp. 147–152
Contribution: Principle author together with Dries Schellekens. Responsible for hardware and software designs.

2.1 Introduction

Most modern trusted computing platforms ensure the secure execution of security-critical software by *isolating* an application from the rest of the processes running on the system. This allows for the correct execution of an application, even when the system is infected with malware. For an overview of hardware-based trusted computing platforms for isolation and attestation, which includes PMAs, the reader is referred to [78].

Currently, isolation is provided by two technologies found in high-end commercial processors, namely ARM TrustZone [10] and Intel SGX [84]. A TrustZone-capable processor contains two virtual processors, each with different privileges and a strictly controlled communication interface. Intel SGX processors provide

multiple protected domains, called enclaves, in which software can operate, free from external observation or modification of the code and data.

A number of works proposed hardware changes to enforce isolation on low-end processors. Kumar et al. designed a system that provides a number of protection domains within the address space [72]. Strackx et al. proposed a simple program counter based memory access control system to isolate software modules [98]. The SMART security architecture supports dynamic remote attestation with a software routine stored in immutable ROM [46]. The Sancus security architecture supports strong process isolation and hardware based remote attestation [87].

When we started this research, the related works applicable to low-end MCUs required their security functionality to execute uninterruptedly. For SMART this is a strict requirement, otherwise an attacker can move malware around during attestation to avoid detection [49]. Concurrent to our research, the TrustLite security architecture was developed, which provides hardware-enforced isolation of software modules with support for secure exception handling, and communication between protected modules [71]. At a later stage, Van Bulck et al. proposed extensions to a PMA to provide availability and real-time support for small microprocessors, which included an interrupt mechanism with a deterministic interrupt latency [106].

In this chapter, we provide a mechanism for handling interrupts for a program counter-based PMA which maintains the confidentiality of the protected module data. The proposed architecture allows *ISRs* to be located in either the secure or the non-secure domain of the processor. This makes it possible to use the processor for real-time processing, secure scheduling, and secure I/O, as tasks running in any security domain can be interrupted. We present a generic solution and compare the results of three implementation options.

The chapter is structured as follows. First, we present the general architecture of our security-enhanced processor, which is based on design principles of the related work [10, 49, 98]. Next, we describe a general scheme that can be used to allow secure interrupts. Subsequently, we discuss three implementations with different design trade-offs. Finally, we discuss our implementation results, and present the conclusion.

2.2 Architecture

This section discusses the security architecture of the system. We first describe the attacker's capabilities, followed by a description of the security enhancements

for domain isolation and security domain switching.

2.2.1 Attacker model

For the attacker model we assume that the adversary is capable of obtaining full control of the state of the software and data. This has the following implications. First, the attacker is capable of modifying any writable code, e.g., with a buffer overflow attack. Second, the attacker can read, and write to any memory region that is not explicitly protected by the processor. Third, the attacker may have compromised the underlying layer of software, e.g., the OS.

We also assume that the attacker is not capable of performing any hardware-based attacks, including placing probes on the memory bus, performing a hard reset of the system, and inducing hardware faults.

2.2.2 Domain isolation

The system is partitioned into two different domains, like in [10, 46]: (1) the *non-secure domain* where regular activities occur, and (2) the *secure domain* where all processing of sensitive data occurs. If an operating system is present on the embedded device, it will typically reside in the non-secure domain; this system for instance contains a network stack or a real-time scheduler. Both the program memory and the data memory are partitioned into their respective secure and non-secure parts. For the sake of simplicity, we only consider a single secure domain, but our scheme can easily be extended to multiple secure domains [72, 87, 98].

The processor makes a distinction between the two domains depending on the Program Counter (PC). When the PC is in the address range of the secure program memory, the system is considered to be inside the secure domain. When the PC is in the address range of the normal program memory, then the system is considered to be in the normal domain.

We assume a low-end processor without a memory management unit (MMU) and hence no support for virtual memory. Instead a basic *memory protection unit* (MPU) is inserted between the processor and the memory. This unit enforces (1) *program counter based access control* [98] on both the data memory and the program memory, and (2) guards the entry into the secure domain by allowing only a single point of entry.

The program counter based access control feature ensures that only the normal program memory and data memory is accessible while the system is in the

Table 2.1: Access rights enforced by the memory protection unit.

	Program Memory		**Data Memory**	
	non-secure	**secure**	**non-secure**	**secure**
Secure domain	`rwx`	`rwx`	`rw-`	`rw-`
Non-secure domain	`rwx`	`r-x`*	`rw-`	`--`

*Execute access only available on the single point of entry

normal domain. However, when the system is inside the secure domain, both the secure program memory and secure data memory also becomes accessible to the processor.

The single point of entry into the secure domain, from hereon referred to as the *single entry point*, is a mechanism that ensures that the secure domain can only be entered at a single address which is located in the secure program memory. Once the program counter has entered the secure program memory at this address, it is allowed to transition to any other address inside either the secure, or non-secure program memory. However, once the program counter points to an address that lies outside of the secure program memory, the secure domain can only be entered again via the single entry point. This feature ensures that secure code cannot be (ab)used to extract secure data from the secure domain. In Return-Oriented Programming (ROP) an attacker selectively executes chunks of program memory, which makes the program misbehave [95]. This could lead to unintended information leaks from the secure domain. The hardware enforcement of the single entry point guards against ROP attacks launched from outside a protected module.

The enforced access control is shown in Table 2.1. The secure data memory is inaccessible when the processor is in the non-secure domain. Furthermore, the non-secure domain only has read access to the secure program memory, except for its first address. This memory address has execute permission, and acts as the only entry point to the secure domain.

In order to expose multiple functions from the secure domain to the non-secure domain, a *jump table* is used, as also proposed in [72, 87]. The identifier of a specific function is stored in a register before jumping to the single entry point. The code at the single entry point then jumps to the correct function based on the identifier passed inside the register.

2.2.3 Context switching between domains

We define a *domain switch* as a transition from one security domain to another. Context switches within a domain (e.g., multithreading, user/kernel mode switching) are not considered in this work.

Two type of context switches can be distinguished. The first type are instructions that alter the program counter, including the `call` instruction, a return from a call with the `ret` instruction, a return from interrupt (`reti`), or with the `jmp` instruction. The second type are hardware events such as interrupts, processor exceptions or a reset.

The memory protection unit enforces the isolated memory regions of the two security domains. However, these domains still share the same set of registers. Therefore, special care is needed such that information does not leak through registers. Consequently, the general-purpose registers need to be cleared before a domain switch to the non-secure domain.

A stack frame is typically used to pass parameters, store return addresses, and for local data storage. A processor maintains a *stack pointer* that points to the top of the stack. In our solution, each domain has its own stack, which is located in its data memory address space, and a dedicated stack pointer register. The processor switches between the two registers depending on the security domain. We chose this option for the sake of simplifying domain switches, and providing better performance. An alternative method, which is used in [87], is to perform stack pointer switching in software by storing pointers to the top of each stack in fixed data memory addresses; this solution requires only a single hardware register.

2.3 Secure Interrupts

This section discusses the scheme for handling secure interrupts. We first describe a typical interrupt mechanism of low-end processors and then present a modified scheme for supporting interrupts with multiple security domains.

2.3.1 Standard interrupt mechanism

An interrupt is a signal generated by hardware or software to indicate to the processor an event that needs immediate attention (e.g., timer, peripheral device, etc.).

Each interrupt can have its own unique *Interrupt Service Routine (ISR)*. The addresses of the ISRs are stored in the *Interrupt Vector Table (IVT)*, which is located at a specific program memory address.

When an interrupt occurs, the following steps are generally performed: (1) the currently executing instruction is completed, (2) the PC that points to the next instruction (which we call the *resume point*), is stored on the stack, (3) the Status Register (SR) (which contains the status flag bits, e.g., zero, carry, and overflow) is pushed on the stack, (4) the interrupt with the highest priority is selected if multiple interrupts occurred during the last instruction, (5) the SR is cleared and further interrupts are disabled, and (6) the address stored in the IVT is loaded into the PC, causing a jump to the ISR. When the ISR is finished, it resumes the interrupted task with the `reti` instruction. This instruction restores the SR and PC from the stack to continue execution at the point where it was interrupted.

Some processor architectures support *nested interrupt*. In this case, interrupts can be re-enabled inside an ISR, causing any interrupt that occurs inside this ISR to interrupt the routine, regardless of its priority.

2.3.2 Domain isolation support

We have extended the processor with the notion of isolated protection domains. As mentioned above, it is crucial that no information leakage occurs during a domain switch from the secure to non-secure domain. With instruction based domain switching, the content of registers is cleared in software just before the transition occurs. However, this strategy cannot easily be applied with an interrupt based domain switch.

There are three main design challenges. First, a hardware interrupt can occur at any point during execution. This implies that there are four possible scenarios: (1) a non-secure task is interrupted by a non-secure ISR, (2) a secure task by a secure ISR, (3) a non-secure task by a secure ISR, and (4) a secure task by a non-secure ISR. The first two are trivial to handle, as no domain switch is required; however, the latter two require a domain switch.

Second, it should be possible for the software to choose whether to resume the interrupted task, or to execute another task. This allows for the scheduling of secure/non-secure tasks, as will be explained in Section 2.3.5.

Finally, the scheme must still comply with the hardware restricted entry into the secure domain. The ISR is unaware whether it is interrupting a secure or non-secure task. Normally it will resume execution with the `reti` instruction.

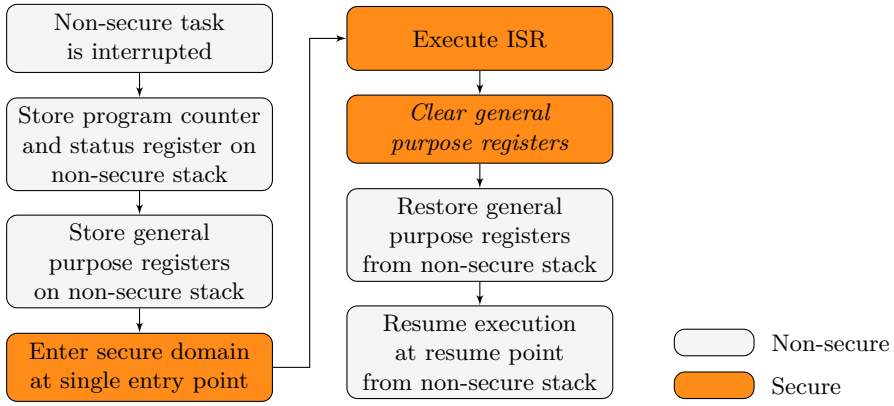

Figure 2.1: The steps for invoking secure Interrupt Service Routines (ISRs) from the non-secure domain.

However, this instruction cannot be used to directly perform a domain switch into the secure domain, as it would invalidate the single entry policy. If this restriction was not in place, then it could lead to an attacker circumventing the single entry policy by pushing a secure domain address onto the stack, followed by issuing a `reti`.

2.3.3 Interrupting non-secure task with secure ISR

In this scenario, an ISR that resides in secure program memory, is invoked from within the non-secure domain. Here we propose to handle this scenario with the scheme shown in Figure 2.1. The secure domain is entered at the single entry point. We propose to solve the problem of invoking the secure ISR from the non-secure domain by adding an entry to the single entry jump table (Section 2.2.2) for each secure ISR. Each entry is responsible for invoking a different ISR. The general-purpose registers should be cleared before a domain switch from the secure domain to the non-secure domain. However, the transition into the secure domain does not require the general-purpose registers to be cleared, because they do not contain any secrets.

2.3.4 Interrupting secure task with non-secure ISR

In this scenario, an ISR that resides in non-secure program memory, is invoked from the secure domain. Here we propose to handle this scenario with the

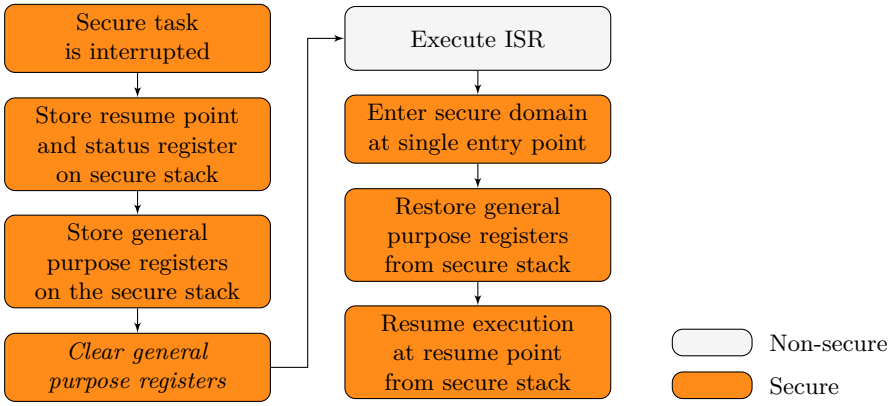

Figure 2.2: The required steps for invoking non-secure Interrupt Service Routines (ISRs) from the secure domain.

scheme shown in Figure 2.2. A domain switch from the secure domain to the non-secure domain is required. Therefore, the general-purpose registers need to be cleared before switching to the non-secure domain. We propose to solve the problem of resuming the interrupted secure task from the non-secure domain, by adding an entry into the single entry jump table (Section 2.2.2) to resume execution at the resume point.

2.3.5 Scheduling

Processors often have real-time operating requirements where scheduling of tasks are essential. When using a preemptive scheduler, interrupts are generated with a hardware timer, to transfer control back to the scheduler. The scheduler then selects the next task to execute, or resume, based on a ranking system. The mechanism that allows for interrupting a secure task with a non-secure ISR also enables preemptive scheduling.

A secure scheduler, as described in [83], enables the on-schedule execution of critical tasks that are running on a partially compromised system. This type of scheduler prevents components under the attacker's control from changing the execution times of other applications. The scheduler is kept isolated from the rest of the software by placing it inside the secure domain. When an application is preempted or an exception occurs, control is transferred to the scheduler which resumes execution of the pending applications. Therefore, the mechanism that allows for interrupting a non-secure task with a secure ISR also enables secure scheduling.

2.4 Implementation

This section presents the implementations that were made in order to demonstrate the feasibility of our proposed scheme. First, we describe the architecture of the processor that we extended with security features. Afterwards, we present three prototype implementations with different design goals, and different design trade-offs in terms of cycles, area, and code size.

2.4.1 MSP430

Our implementation is based on the low-cost, low-power TI MSP430 microcontroller. It features a 16-bit von Neumann processor, and a single 16-bit address space for program and data memory. It has no external memory bus, and the amount of on-chip memory is limited to 16 kB RAM and 256 kB flash memory. It has eleven general-purpose registers (R4-R15), with R0-R3 serving as a program counter, stack pointer, status register, and constant generator.

Most interrupts on the MSP430 architecture are maskable, and can therefore only cause an interrupt when they are enabled, and if the general interrupt enable bit is set inside the status register.

A multiplexer is used to select between the normal stack pointer and the secure stack pointer, depending primarily on the value of the program counter.

One of the design problems we faced, was that ISRs have a return control flow that depends on the domain that the ISR was invoked from. We decided that each ISR should use the same return mechanism, regardless of the domain it is invoked from. We solve this by invoking all ISRs that require a domain switch in a special manner, which we call an *emulated interrupt*. Instructions are used to emulate what the hardware does when an interrupt occurs, by pushing the SR and the address of a *return trampoline* onto the stack, followed by a jump to the ISR. The invoked ISR executes, and returns with a `reti`. Since the address of the return trampoline is still on the stack, the `reti` will invoke the return trampoline.

2.4.2 Software-based implementation

The goal of the software-based implementation is to use the minimum amount of hardware features. We opted to make use of a different IVT for each security domain. Since the ISRs have control flows that depend on the current security

domain, the idea is that each IVT will serve as the starting point for each of these control flows.

An IVT is normally populated with the addresses of ISRs, each associated with a different interrupt. As we now have an IVT for each security domain, we populate it with (1) the addresses of ISRs that exist in the same security domain as the IVT, and (2) the addresses of software routines that will initiate the control flow to invoke ISRs located in the other security domain.

The second IVT, which we refer to as the *secure IVT*, is stored at a fixed address in secure program memory. A hardware feature selects between IVTs, depending on the current security domain.

Upon entering the secure domain at the single point of entry, the value stored in R15 is used to determine which function in the jump table to execute.

Interrupting a non-secure task with a secure ISR

The steps for interrupting a non-secure task with a secure ISR is shown in Figure 2.3. When an interrupt occurs, the resume point is stored on the non-secure stack. Afterwards, the general-purpose registers are stored on the non-secure stack, and a `call` is made to the entry in the single entry jump table that corresponds to the current ISR. Next, the secure ISR is invoked with an emulated interrupt. The `reti` instruction inside the ISR returns from the emulated interrupt, after which the general-purpose registers are cleared, followed by returning (`ret`) from the `call`, which causes a transition back to the normal domain. Next, the registers are restored from the non-secure stack, and the resume point is used to jump to the point where execution was interrupted.

Interrupting a secure task with a non-secure ISR

The steps for interrupting a secure task with a non-secure ISR is shown in Figure 2.4. When an interrupt occurs, the resume point is stored on the secure stack. Afterwards, the general-purpose registers are stored on the secure stack, the registers are cleared, and the non-secure ISR is invoked by means of an emulated interrupt. The `reti` instruction inside the ISR returns from the emulated interrupt, after which a zero is stored in R15, and a jump is made to the single entry point. The value of zero in R15 corresponds to a jump table entry that is responsible for restoring all registers from the secure stack, and using the resume point to jump to the point where execution was interrupted.

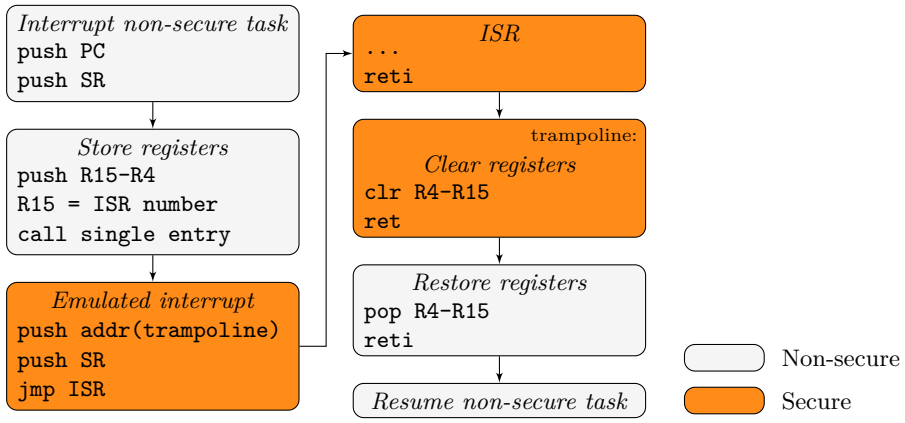

Figure 2.3: Software-based flowchart for invoking a secure ISR from the non-secure domain.

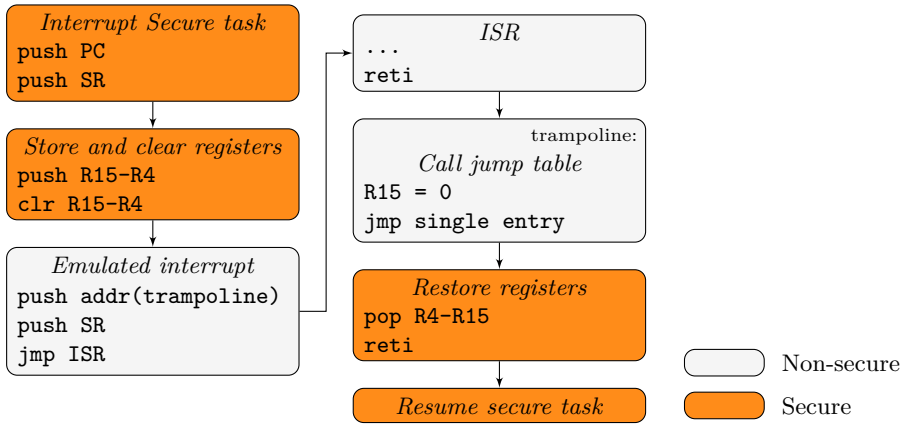

Figure 2.4: Software-based flowchart for invoking a non-secure ISR from the secure domain.

2.4.3 Hardware-based implementation

The goal for the hardware-based implementation is to minimize interrupt latency by adding more functionality to hardware. As a further goal we try to minimize the amount of additional hardware. The architecture of this implementation is very similar to the software-based implementation (Section 2.4.2), with the exception of the following: (1) a single IVT is used, and (2) the clearing, storing, and restoring of general-purpose registers is now done in hardware.

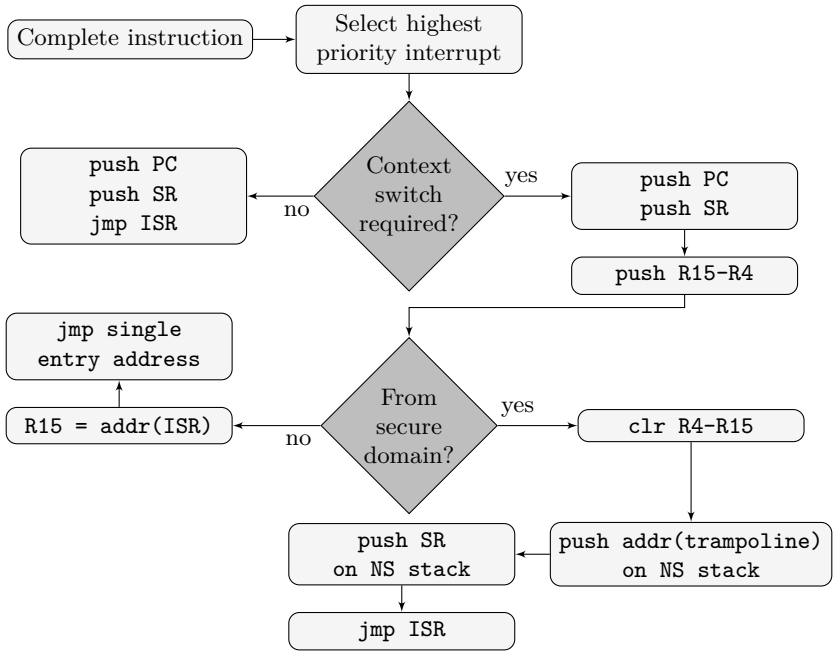

Figure 2.5: The modified hardware-based interrupt logic.

A single IVT is used to store the addresses of the ISRs that can be located in either domain. The extended interrupt logic, shown in Figure 2.5, takes care of any additional processing that needs to be done if a domain switch is required to invoke an ISR.

Figure 2.1 and Figure 2.2 show that both schemes require: (1) saving the general-purpose registers to the stack, and (2) restoring the general-purpose registers from the stack after domain switching back to the interrupted task's domain. We propose to solve (1) by further extending the interrupt logic to save the general-purpose registers on the stack before performing the domain switch to the other domain. We further propose to solve (2) by introducing a new instruction that restores all registers from the current stack.

2.4.4 Hidden registers optimization

We also propose a performance optimization to improve interrupt latency. For this optimization, the number of visible registers inside the ISR is restricted to n. This improves interrupt latency because less registers need to be cleared, saved

Table 2.2: A summary of the hardware costs for the different designs.

Design	LUTs	Registers
Unmodified	2231	1185
Software-based	2241	1187
Hardware-based	2417	1219
HW-based Hidden registers	2420	1220

and restored from the stack, but without the hardware cost of using shadow registers. To ensure that the remaining $11 - n$ general-purpose registers do not leak any information, all read and write operations on the remaining registers will be blocked. The compiler/programmer needs to ensure that only the visible n registers are used in the ISR, as the remaining registers are unusable.

This optimization can be done on either the hardware-based architecture (Section 2.4.3), or the software-based architecture (Section 2.4.2), and will require the following features to be activated when inside a domain switched ISR: (1) the saving and restoring of the general-purpose registers on the stack is limited to only n registers, and (2) the register file is modified to disable read and write access on the remaining registers.

2.5 Evaluation

This section presents our results, together with a discussion on the limitations of this work.

2.5.1 Results

We extended the openMSP430 [2] softcore to create our prototype implementations. This softcore is fully compatible with the TI MSP430 microcont¸roller, and executes code generated by any MSP430 toolchain in a near cycle-accurate way. The openMSP430 softcore was configured to use the following settings: a 10 MHz clock, 4 kB of data memory, 8 kB of program memory, a hardware multiplier, and a single timer. We used the Digilent Atlys, Spartan-6 LX45 based FPGA development board to test our prototype.

Table 2.2 shows the number of 6-input Look-Up Tables (LUTs), and registers for the unmodified openMSP430, and the three different designs. The synthesis optimization goal was set to “area”.

Table 2.3: Context switching cycle times for an interrupted task. The number of visible registers are indicated with n.

Context switch	SW	HW
Interrupt S to NS	71	9+n
Return NS to S	43	14+n
Interrupt NS to S	54	13+n
Return S to NS	52	10+n
Interrupt *	6	6
Return *	5	5

* No domain switch occurs here.

Table 2.3 compares the number of cycles required to perform an interrupt-based context switch. An interrupt that does not require a domain switch needs 6 cycles to pass control to the ISR, whereas 5 cycles are required to resume an interrupted task with the `reti` instruction. Pushing a register onto the stack requires 3 cycles, whereas popping a value from the stack requires 2 cycles. For the cycle times it is assumed that jump table logic requires only 4 cycles.

The results show that the software oriented technique has the slowest context switching time, and further requires the least amount of hardware modifications. Our hardware-based design has a moderately good context switching time with a slightly bigger hardware cost. The hidden register method can provide the fastest context switching time, at the cost of a small amount of additional hardware, and a reduced number of available registers in the ISR.

2.5.2 Limitations

It is important to note that this work only addresses the issue of maintaining the confidentiality of the protected data. However, there are additional security aspects which need to be considered when providing preemption support for PMAs. One aspect is interrupt authentication, where the architecture ensures that secure world interrupts are only invoked by a legitimate interrupt. The problem is that when interrupts can be spoofed by an attacker, it can be used make the system misbehave by tricking it into believing that an interrupt has been fired. An additional security aspect that needs to be considered is availability. Here, it would be important to ensure that a malicious interrupt service routine cannot hijack the control flow by executing code which never returns. In general, availability is an extremely difficult property to guarantee on most systems. Another limitation of this work is that it only supports two

domains. To support multiple secure domains will likely introduce additional challenges, such as storing the stack pointer register for each domain.

2.6 Conclusion

In this chapter, we proposed an architecture to provide preemption support for program counter-based PMAs. We proposed three implementations, each with different design trade-offs, and made the prototypes by extending the openMSP430 softcore. In all three cases, both hardware, as well as software techniques are required to make the prototypes.

Our results show that preemption support for PMAs on low-end microcontrollers are feasible, since the hardware cost is minimal, while the cycle overhead for domain switches is acceptable.

Chapter 3

Control Flow Integrity

The previous chapter was related to Protected Module Architectures (PMAs), which provide runtime protection through isolation and attestation. This chapter presents a detailed analysis of the current state-of-the-art in hardware-based *Control Flow Integrity (CFI)* architectures. This addresses one of the thesis goals, namely to analyse existing hardware-based security mechanisms for microprocessors.

Content Sources

The content of this chapter is currently under review at the ACM Computing Surveys journal.
de Clercq, R., and Verbauwhede, I. A Survey of Hardware-based Control Flow Integrity. In *ArXiv CoRR* (2017), abs/1706.07257
Contribution: Main author.

There are three important differences between Protected Module Architectures (PMAs) and CFI. First, their respective use cases differ. PMAs place small amounts of code inside a protected module to protect it from attacks originating from outside the protected module. The assumption here is that the protected software is free from vulnerabilities. In contrast, CFI is used to protect larger quantities of software, which is assumed to contain vulnerabilities, by monitoring the behaviour of the executing software.

Second, the security properties provided by each differ. PMAs enforce access control to protect its state, prevent leakage and further provides mechanisms to attest its state. However, Code Reuse Attack (CRA) protection is limited to calls from outside a module which have to respect the single entry point, while

no CRA protection is provided for the code stored inside the protected module. It is imperative that code inside a protected module is free from vulnerabilities, otherwise a CRA can be launched from inside the protected module. In contrast, CFI does not offer any access control protection, but aims to provide strong protection against CRAs for all protected code.

Third, the assumed attacker capabilities differ. Most PMAs assume a powerful attacker that is in control of all memory as well as the OS. However, CFI assumes a range of different attackers capabilities, which includes being in control of the data memory, being in control of the code memory, and being capable of inducing fault attacks on the control flow.

3.1 Introduction

Control Flow Integrity (CFI) is a term used for computer security techniques which prevent CRAs by monitoring a program's flow of execution (*control flow*). CFI techniques do not aim to prevent the sources of attacks, but instead rely on monitoring a program at runtime to catch deviations from the normal behaviour. CFI can detect a wide range of attacks, since many attacks rely on control flow hijacking in order make a program misbehave.

CFI has received a lot of attention by the research community, but has not yet been widely adopted by industry. This could be due to the practical challenges and limitations of enforcing CFI, such as requiring complex binary analysis or transformations, introducing unwanted overhead, or offering incomplete protection. In this chapter, we present an analysis of the security policies enforced by 21 state-of-the-art *hardware-based CFI* architectures. The primary focus is on CFI policies enforced in hardware by the underlying architecture, and therefore software-based solutions are only briefly discussed. For an overview of software-based CFI, the reader is referred to [20]. We identified a total of 13 security policies, and discuss each policy in terms of its security, limitations, hardware cost, and practicality for widespread deployment.

The remainder of this chapter is structured as follows. First, we motivate the need for CFI by introducing the recent history of attacks and countermeasures. After that, we provide some background on the enforcement of CFI, the need for hardware-based CFI, together with a description of two methods for interfacing the CFI hardware monitor with the processor. Next, we introduce the three different kinds of attackers assumed by most CFI architectures. Subsequently, we present the classical approach for enforcing CFI together with its challenges and limitations. Afterward, we present the CFI policies used by the 21 hardware-based CFI architectures evaluated in this chapter, followed by a discussion on

CFI enforcement via the processor's debug interface. Finally, we provide a detailed comparison of the architectures and their hardware costs/overhead, followed by a conclusion.

3.2 Attacks and countermeasures: an arms-race

This section provides a brief history of attacks and countermeasures.

We define a *memory error* as a software bug which is caused by invalid pointer operations, use of uninitialised variables, and memory leaks [102]. Of particular importance is memory corruption, which typically occurs when a program unintentionally overwrites the contents of a memory location. Memory errors are produced by memory unsafe languages, such as C and C++, which trade type safety and memory safety for performance. These errors are prevalent in a large amount of deployed software: (1) memory unsafe languages are frequently used to write OS kernels and libraries which are also used by memory safe languages, while (2) some memory safe languages, such as Java, rely on an interpreter which is written in a memory unsafe language. Attackers exploit these memory errors to intentionally overwrite a memory location. Defending against the exploitation of existing software bugs has led to an arms-race between attackers and defenders, where each new defence leads to an attack that circumvents it.

An important example of exploiting a memory error is the *buffer overflow*, where more data is written to a buffer than the allocated size, leading to the overwrite of adjacent memory locations. Attacks frequently make use of a buffer overflow by providing input data that is larger than the size of the buffer. This causes other items to be overwritten, such as local variables, pointer addresses, return addresses, and other data structures.

Many *code injection attacks* rely on a stack-based buffer overflow to inject shellcode onto the stack and overwrite the return address. By overwriting the return address, the attacker can change the control flow to any location during a function return. The attacker uses this to divert control flow to the injected shellcode, thereby allowing him to execute arbitrary code with the same privileges as that of the program. To defend against return address tampering, *stack canaries* involve placing a canary value between the return address and the local function variables [16, 29]. The canary value is verified by a code fragment before returning from the function. However, canaries can be circumvented and further require the insertion and execution of additional instructions at the end of each function call [8]. Another attack vector, known as *format string vulnerabilities*, allows overwriting arbitrary addresses. This vector can be used to overwrite return addresses without changing the canary value [86].

An effective defence against code injection attacks is non-executable (NX) memory, a.k.a W⊕X (Write XOR eXecute), a.k.a Data Execution Prevention [1]. An NX bit is assigned to each page to mark it as either readable and executable, or non-executable but writable. Most high-end modern processors have architectural support for W ⊕ X, while most low-end processors do not. This protection mechanism was circumvented by the invention of *code reuse attacks (CRAs)*, which do not require any code to be injected, but instead uses the existing software for malicious purposes. An example of this is the *return-to-libc* attack, where the attacker updates the return address to force the currently executing function to return into an attacker-chosen routine. In addition, the attacker places function arguments on the stack, thereby providing him with the ability to supply attacker chosen arguments to a function. While the attacker could return anywhere, libc is convenient since it is included in most C programs. A popular libc attack target is to spawn a shell by returning into `system("/bin/sh")`, or to disable W ⊕ X by returning into `mprotect()/VirtualProtect()`.

Return-Oriented Programming (ROP) [95] is a powerful CRA which is Turing-complete [1]. ROP makes use of code *gadgets* present inside the existing program. Each gadget consists of a code fragment that ends with a return instruction. The attacker overwrites the stack with a carefully constructed sequence of gadget arguments and gadget addresses. The goal is to invoke a chain of gadgets, with each return instruction leading to the invocation of the next gadget. After the initial function return, the first gadget is executed, leading to the eventual execution of a return, which causes the next gadget to be executed.

Another type of attack, called *Jump-Oriented Programming (JOP)* [17], is also Turing-complete. The building blocks are also called gadgets, but here each gadget ends with an indirect branch instead of a `ret` instruction. A dispatch table is used to hold gadget addresses and data, while a processor register acts as a virtual program counter which points into the dispatch table. A special gadget, called a dispatcher gadget, is used to execute the gadgets inside the dispatch table in sequence. After invoking each functional gadget, the dispatcher gadget is invoked, which advances the virtual program counter and then launches the next functional gadget.

Code randomisation protects against CRAs by placing the base addresses of various segments (`.text, .data, .bss,` etc) at randomised memory addresses, which makes it difficult for attackers to predict target addresses. This policy is currently used in one form or another in most modern OSs. The security of this technique relies on the low probability of an attacker guessing the randomly

[1] A computation model is called Turing-complete when it has the same computational power as a Turing machine [105].

placed areas. Therefore, a larger search space means more effective security. The Linux PaX project introduced code randomisation with a patch for the linux kernel in July 2001. Code randomization suffers from two main problems. First, the effectiveness of this policy relies on the number of bits available for randomization. Therefore, this policy is particularly vulnerable to brute force attacks on 32-bit architectures, since only a small number of bits are available for randomisation [96]. Second, it is vulnerable to memory disclosure attacks, since only the base addresses of each segment is randomised. If an attacker gains knowledge of a single address, he could compute the segment base address, which causes the system to again become vulnerable to CRAs. One method to disclose a memory address is by exploiting a format string vulnerability.

To address the above problems, fine-grained code randomization randomly re-orders the memory blocks of a program when it is launched, thereby ensuring that every execution instance is unique [57, 108]. This makes it significantly harder to use brute force to launch a CRA, since the entropy is increased.

Non-control data attacks rely on corrupting data memory which is not directly used by control flow transfer instructions [25]. In the past, it was assumed that non-control data attacks were limited to data leakage (e.g., HeartBleed [43]) or data corruption. However, recently this attack vector was used to launch two different Turing-complete attacks which can circumvent CFI policies (see Section 3.5.1 and Section 3.5.3).

3.3 Background

3.3.1 Control Flow Integrity (CFI)

For a definition of control flow, Control Flow Integrity (CFI), Control Flow Graph (CFG), basic blocks, forward edges, and backward edges, the reader is referred to Section 1.1.

At runtime, the dynamic control flow changes are restricted to the static CFG. Many CFI architectures only validate control flow changes caused by indirect branches, such as calculated jumps, calculated calls, and returns. The assumption is that the software is immutable, which means that static branch targets do not need to be checked. This implies that this policy cannot be used for self-modifying code, or for code generated just-in time. Software-based CFI policies typically verify each indirect branch target before the execution of an indirect branch instruction.

We define *fine-grained* CFI as a policy which only allows control flow along valid edges of a CFG. In recent years, many works proposed architectures which relax the strict enforcement of the CFG, in order to gain performance improvements. We define *coarse-grained* CFI as a policy which does not enforce a strict CFG. These policies rely on enforcing simple rules, such as ensuring that return targets are preceded by a `call` instruction, and that indirect branches can only target the first address in a function. They offer less security than fine-grained CFI policies, since they allow control to flow along paths that do not exist inside the CFG, as demonstrated by recent attacks [32, 53].

3.3.2 The need for hardware-based CFI

Many software-based CFI solutions rely on instrumentation, where code is inserted into a program to perform CFI checks on indirect branches. This can be done as part of a compiler optimisation step, static binary rewriting, or through dynamic loading. When the compiler is unaware of the security aspects concerning the CFI checks, it might cause the optimisation step to spill registers holding sensitive CFI data to the stack. Static metadata is often protected with read-only pages. However, many software-based CFI architectures rely on runtime data structures which are stored in memory which is sometimes writable. Recent attacks exploited this problem by tampering with runtime data structures to circumvent the security of the system [28, 48]. However, runtime data structures can be protected through instruction-level isolation techniques, such as memory segmentation or Software-based Fault Isolation. Memory segmentation provides effective isolation on x86-32, but is not really useful on x86-64 since segment limits are not enforced. Software-based Fault Isolation executes additional guards to ensure that each store instruction can only write into a specific memory region. It is worth pointing out that instruction-level isolation techniques rely on the integrity of code. Hardware-based access control mechanisms can provide strong isolation for runtime data structures and metadata. In addition, it has little overhead, and does not rely on code integrity to protect sensitive information.

The *Trusted Computing Base (TCB)* is the set of hardware and software components that are critical to the security of the system. The careful design and implementation of these components are paramount to the overall security of the system. The components of the TCB are designed so that other parts of the system cannot make the device misbehave when they are exploited. Ideally, a TCB should be as small as possible in order to guarantee its correctness [78].

Software-based CFI uses instrumented code to monitor the behaviour of other software. To prevent tampering, the software is protected by page-based access

control (such as $W \oplus X$). However, this only temporarily makes the software immutable, and care needs to be taken to ensure that an attacker cannot disable page-based protection with a syscall to `mprotect()/VirtualProtect()`. In contrast, the underlying hardware architecture is immutable, and is therefore more trusted. To enforce a strong security policy, it is well known that the TCB should consist of as little as possible software, while placing as much as possible security-critical functionality in hardware.

Hardware-based mechanisms can protect against strong attackers, such as attackers that control both code and data memory, as well as attackers with physical access that can perform fault attacks. In addition, it allows for verifying not only indirect branches, but also for direct branches, unintended branches, and for control flow between instructions inside a basic block, which may be altered during a fault attack. Furthermore, the TCB size is usually smaller, since security-critical functionality is mostly implemented in hardware, with little or no trusted software. Moreover, hardware-based mechanisms provide better performance, since fewer processor cycles need to be spent on performing security checks.

An important design decision for security-critical systems is to select which components need to be placed inside the TCB. Many hardware-based CFI architectures need software components to communicate runtime information with the CFI hardware or configure the CFI hardware. Care needs to be taken to ensure: the correctness of the TCB components, that the hardware/software interface cannot be exploited, that sensitive data which is stored in main memory is protected, and that the hardware-based security policies cannot be circumvented.

3.3.3 Hardware monitor

Hardware-based CFI architectures use trusted hardware to monitor the behaviour of the software, which we call the *hardware monitor*. Providing the hardware monitor with access to the necessary runtime information and signals, such as control flow information, runtime data structures, and metadata is an important issue when designing the CFI architecture. To access runtime control flow information, the following two approaches can be used: (1) integrating the hardware monitor into a processor by modifying the instruction pipeline stages, and (2) interfacing the hardware monitor with the processor's hardware debugger. To access runtime data structures or metadata stored in main memory, the hardware monitor can be connected as a master to the processor's bus, thereby providing access to the processor's memory space.

Processor's pipeline stage

Many architectures integrate their hardware monitor into one or more of the processor's instruction pipeline stages, as shown in Figure 3.1. This allows all the processor's internal signals to be exploited for enforcing the CFI policy. For example, many CFI policies require runtime branch information such as the branch source address, target address, and branch type. To enforce CFI, the required signals can be forwarded from inside the appropriate pipeline stage to the hardware monitor, which uses this information to verify the validity of each branch.

Another approach is to add *special instructions* to the Instruction Set Architecture (ISA) to enforce a CFI policy. This also requires modification of pipeline stage(s), such as the Instruction Decode stage. In order to use the special instructions, a software toolchain is often used to insert these special instruction into the compiled software.

In Figure 3.1 the hardware monitor is integrated with the Instruction Decode stage. However, the monitor can be integrated into multiple different pipeline stages. The decision on where to integrate the monitor depends on the availability of information at a given stage.

See Section 3.6 for a presentation on the state-of-the-art in CFI architectures that are integrated into the pipeline stages of the processor.

The biggest disadvantage of integrating a monitor into the pipeline stages is that it does not follow the typical SoC design flow, as it requires modification of existing IP, such as the processor.

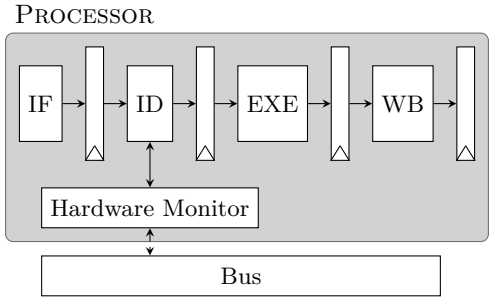

Figure 3.1: The hardware monitor is integrated into the instruction pipeline of the processor. The pipeline stages are abbreviated as follows: Instruction Fetch (IF), Instruction Decode (ID), Execute (EXE), and Write Back (WB).

Debug interface

Some recent works exploit the processor hardware debug interface to monitor a program's behaviour without modifying the processor or other pre-existing IP, as shown in Figure 3.2. This is a major advantage, as it complies with the design rules of SoCs. Modifying IP is expensive, as it requires modification by the IP vendor (expensive), or requires the IP vendor to release its hardware source files to the client (even more expensive). Monitors that are realised by hardware-based standalone IP are practical, because they do not require the modification of the target processor, but simply communicate with the target processor via existing interfaces, such as the processor's hardware debug port or the bus.

A major challenge with enforcing CFI through the debug interface, is to obtain access to all the required runtime information. Since the hardware debugger is designed for debugging purposes, it usually does not provide all the necessary information to enforce a given protection mechanism. A common approach to solve this is to provide supplementary metadata inside main memory, or to communicate the missing information by instrumenting the binary to write the required information to MMIO.

Some SoCs have an external debug interface, which facilitates the use of an external hardware monitor. In this case the monitor can be realised by an off-chip microprocessor or other dedicated hardware (FPGA or ASIC), which are connected to the external debug interface.

See Section 3.7 for a presentation on the state-of-the-art in CFI architectures that monitor the processor via the hardware debug interface.

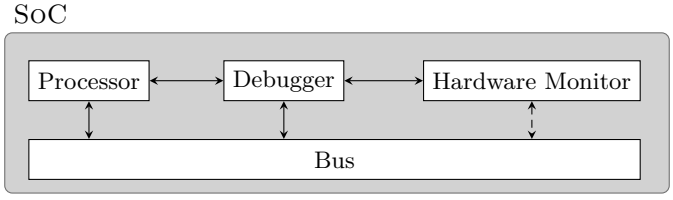

Figure 3.2: A CFI hardware monitor interfaces with the processor's debug port. In addition, a Memory-Mapped IO (MMIO) interface is nused by instrumented code to communicate with the hardware monitor.

3.4 Attacker model

The goal of the CFI architecture is to prevent attackers from launching a CRA. In contrast, the goal of the attacker is to use a CRA to perform arbitrary code execution, confined code execution, arbitrary computation, or information leakage.

In general, the architectures described in this chapter use one of three different attacker models. However, each of the three attacker models share the following assumptions.

- The memory contents and layout are readable by the attacker. The usual assumption is that code randomization can be circumvented, and that the location of all program segments are known.
- The hardware is trusted.
- Memory errors can be present inside the program. However, the usual assumption is that no other attack vectors or security holes exist that could directly allow an attacker to perform a privilege escalation.
- The attacker can arbitrarily write to memory by exploiting a memory error. The exact locations of writable memory depend on the used attacker model.
- Side-channel attacks are not possible.

The *first attacker model* assumes that the attacker is in control of data memory (which includes the stack and the heap), but not code memory. The usual assumption is that W$\oplus$X protection is enforced, which means that code injection and code tampering attacks cannot be performed. This further means that the OS is implicitly trusted, since OS support is required to set the page bits to enforce W $\oplus$ X. In order to launch an attack, the attacker has to exploit a memory error to overwrite data memory, such as code pointers or other data structures.

The *second attacker model* assumes that the attacker is in control of data memory as well as code memory. An attacker can therefore perform code injection and code tampering, since W $\oplus$ X is not enforced. However, architectures using this attacker model usually enforce a *Software Integrity (SI)* mechanism to prevent the execution of tampered/injected code. In this model, an attack can be performed by exploiting a memory error to overwrite any memory location.

The *third attacker model* also assumes an attacker in control of all memory. In addition, the attacker can perform non-invasive fault attacks that target the

program flow, such as glitching the clock. However, other types of fault attacks that do not target the program flow, such as glitching the bus of the processor, are not considered part of the attacker model. Therefore, in this scenario the attacker has two avenues for launching an attack: exploiting a memory error to overwrite any memory location, or performing a fault attack on the control flow.

3.5 Classical CFI

This section discusses Abadi et al.'s classical approach for enforcing CFI [6]. Even though this policy was originally enforced in software, many of its challenges and limitations are also applicable to hardware-based CFI. For an in-depth discussion on hardware-based CFI policies, see Section 3.6.

3.5.1 Labels

The label-based approach relies on inserting unique label identifiers at the beginning of each basic block. Before each indirect branch, the destination basic block's label identifier is compared to a label identifier which is stored inside the program. In other words, the correctness of the destination basic block's label identifier is verified before every indirect branch. Since unique label identifiers are used, control flow tampering causes the check to fail, since the destination label identifier will not match the label identifier stored inside the program. The control flow checks are performed using code checks which are inserted at the end of each basic block containing an indirect branch.

Limitations

A *static CFI* policy (such as the label-based approach of [6]), can only check that control flows along the CFG. It is *stateless*, since the the stack's runtime state is not considered when determining which control flow paths are valid. This means that it cannot be used to ensure that a function returns to its most recent call site. In other words, when enforcing a static CFG, each function can return along any valid backward edge inside the CFG. This is a problem for any function which is called from more than one location. During normal execution, the function might be called from one location, but after an attacker overwrites the return address, the function can return along another CFG edge to a different valid call site (of the attacker's choosing).

Carlini et al. demonstrated the severity of the stateless problem, by performing a non-trivial attack in the presence of a strict static CFI policy [21]. A *Control-Flow Bending* attack relies on *bending* control flow along valid CFG paths by overwriting return addresses. This allows an attacker to rapidly traverse the CFG in order to execute any attacker-chosen code.

3.5.2 Shadow Call Stack (SCS)

To address the stateless problem of static CFI and enforce a strict backward-edge policy, a *Shadow Call Stack (SCS)* can be used [88]. The goal of an SCS is to detect tampering with return addresses stored on the stack during a function call. To achieve this, an additional stack is used to store an extra copy of the return address for each function call. Therefore, every `call` requires that two copies of the return address are stored, one inside the stack frame, and the other inside the SCS. Before returning (`ret`) from a function call, the integrity of the return address is verified by comparing the return address stored on the two stacks. If there is a mismatch, an exception is raised. In essence, this means that the SCS ensures that every call to a function returns to its call site.

A major advantage of enforcing an SCS is that it provides an excellent defence against attacks on backward edges, which include ROP and return-into-libc attacks. In addition, it doesn't rely on a CFG, which is problematic to obtain in certain types of programs. These factors make hardware-based SCS suitable for widespread adoption.

An SCS exists independent of static CFI, and provides a complimentary set of security properties. When a stateful backward-edge policy, such as SCS, is used together with a forward-edge static CFI policy (such as the label-based approach), we call it *dynamic CFI*.

3.5.3 Challenges and limitations

Generating a precise CFG

Many CFI architectures rely on a CFG which was calculated through static analysis. However, generating a precise CFG through static analysis is unsolved for some program types. It is especially difficult to compute a precise CFG for certain programming constructs that rely on indirect branches, such as function pointers which are passed as between functions and C++ virtual methods. To overcome this, static analysis tools rely on over-approximation, which leads to an imprecise CFG that contains more edges than necessary [21, 70]. This

degrades the security of the CFI scheme relying on the CFG, since the security of CFI depends on an accurate CFG.

Static analysis tools can either work from source code or on binaries. It is generally accepted the accuracy of a CFG can be improved when relying on static source code analysis than on static binary analysis. For a classification of the precision of computing a CFG using different static analysis techniques, the reader is referred to [20].

It is worth mentioning that this problem can be avoided altogether by writing programs that are easy to analyse. However, this imposes many restrictions, such as disallowing the use of function pointers, and disallowing the use of C++ objects.

In general, security architectures which rely on a precise CFG cannot enforce a strict policy, since a precise CFG cannot be generated certain types of programs. However, if a program contains no indirect forward branches, then this is not a problem, since static analysis can be used to generate an accurate CFG for direct branches and function returns.

Unintended branches

Unintended branches can occur in architectures which use a variable length instruction encoding. To exploit it, the attacker deviates control flow to the middle of an instruction. Many coarse-grained policies cannot check for unintended branches. This is a serious concern, since the majority of gadgets in a program consists of unintended branches, e.g., 80% of all libc gadgets are due to unintended branches [66].

Limitations

This section discusses some advanced attacks that can circumvent the security of dynamic CFI. In general, most CFI architectures are extremely vulnerable to non-control data attacks. For example, let's consider a program which contains a for-loop. If an attacker tampers with the loop counter, CFI will not detect any abnormalities, since it can only detect control flow along invalid edges. Therefore, CFI can ensure only that executed edges are valid, but cannot ensure that the sequence of executed edges are correct. It is important to stress that this is a fundamental limitation of CFI.

A *printf-oriented programming* attack provides Turing-complete computation, even when enforcing a strict CFI policy together with a SCS [21]. Here,

the attacker exploits the standard library's printf function by controlling the format string, arguments, and destination buffer. The printf function allows for performing arbitrary reads and writes, and conditional execution. In addition, the attack is Turing-complete when a loop in the CFG can be exploited.

Counterfeit Object Oriented Programming demonstrated that C++ virtual functions can be chained together to create Turing-complete programs even when precise dynamic CFI is enforced [94]. The attack injects counterfeit C++ objects into the program address space followed by a manipulation of virtual table pointers. It is believed that this attack was possible because most CFI defences do not consider C++ semantics.

Data-oriented programming can achieve Turing-complete computation by utilising non-control data attacks in the presence of a precise and dynamic CFI policy [62]. Similar to ROP, these attacks utilise data-oriented gadgets, which consists of short instruction sequences that enable specific operations. In addition, a dispatcher gadget is used to chain together a set of gadgets to force a program to carry out a computation of an adversary's choice. The authors demonstrated the attack by disabling page-based protection in the presence of a software-based dynamic CFI policy.

3.6 Hardware-based CFI Policies

This section presents the CFI policies used by the hardware-based CFI architectures evaluated in this chapter. We discuss the benefits, challenges, and hardware cost for enforcing each policy in hardware.

3.6.1 Shadow Call Stack (SCS)

An SCS, as introduced in Section 3.5.2, is used by many hardware-based CFI solutions [12,27,30,50,60,63,66,67,74,75,76,81,101] to enforce a backward-edge policy, together with a range of different forward-edge policies. A hardware-based SCS typically requires the following components: a hardware buffer to store the most recently used entries, logic to manage the entries in the buffer, logic to interface with the main memory, and logic to handle exceptions (setjmp/longjmp).

Hardware buffers

When enforcing an SCS with a hardware monitor which has fast access to the on-chip main memory of a small embedded processor, it is possible to store the entire call stack without paying a performance penalty [50]. However, doing this for processors using off-chip main memory will incur a significant overhead since each main memory access can take a large number cycles. Therefore, to reduce the number of main memory accesses, a hardware buffer can be used to store the most recently used entries of the shadow stack. This can lead to a significant speedup, since accessing the hardware buffer can be done in a single cycle.

The next challenge is to select the buffer size. Since on-chip memory is expensive, the hardware buffer can only accommodate a limited number of entries. One approach is to design the buffer size around the maximum call depth of a program. However, Ozdoganoglu et al. found a program in their benchmark suite has a maximum call depth of 238 [88]. In addition, it is expected that the maximum number of shadow stack entries will further increase with multithreading. Therefore, in order to keep the hardware cost low by having only a small buffer, and further allow for deeply nested calls, the older entries in the shadow stack can be stored in main memory. One approach is to use an interrupt service routine to write the contents of the SCS to main memory when an overflow occurs [88]. Another approach is to use a hardware-based *shadow stack manager* to copy the oldest 50% of buffer elements to main memory upon detecing a buffer overflow [77]. Most hardware architectures seem to use a buffer that accommodates a call depth of 16 or 32 entries.

Protecting runtime data

A shadow stack stored in main memory allows for an almost unrestricted call depth. However, since we assume an attacker which controls data memory, a mechanism is needed to prevent tampering of runtime SCS data stored in main memory. IBMAC, which is implemented on a small microprocessor, uses an SCS memory region that can only be updated by `call` and `ret` instructions [50]. Intel CET uses special SCS pages such that regular store instructions cannot modify the contents of the shadow stack [63]. Lee et al. uses a hardware component that prevents the processor from accessing the SCS memory region by snooping traffic between the processor and the memory controller [74]. HCFI stores its entire call stack in a hardware buffer, which cannot be accessed by the processor [27]. However, this restrictive approach can only accommodate a limited call depth. Some hardware-based approaches rely on memory allocated in kernel space to ensure that the SCS is separated from the process's memory

space [30,65,66,101]. Here, the OS kernel maps a region of physical memory and reserves it as part of MMIO. The pages can be marked as write-protected while still being updatable from the SCS hardware. Kernel memory is not accessible from user space, thereby only allowing the hardware monitor and the kernel to access the memory.

Support for setjmp/longjmp

Complex binaries sometimes have exceptions to the normal behaviour of `call`, `ret`, and `jmp`. One such case is `longjmp()`, which performs stack unwinding to restore the stack to the state it had during the previous `setjmp()` invocation. Therefore, after a `longjmp()`, when the subsequent return is made, the expected return address will not be on the top of the SCS, because it hasn't been unwinded yet. Smashguard proposed to remove elements from the top of the stack until the top SCS element matches the return address [88].

HCFI proposed to record the current SCS index when `setjmp()` is invoked, and unwinding the SCS to the recorded index for the subsequent `longjmp()` invocation [27]. To support multiple `setjmp/longjmp` pairs, a unique 8-bit identifier is assigned to each pair, and a 256-element hardware buffer is used to record the SCS indices for each unique identifier.

Das et al. noted that `longjmp()` sometimes uses an indirect `jmp` instruction instead of a `ret` [30]. This causes an exception for the next executed `ret`, since the expected `ret` did not execute, and the top SCS element was not evicted. In addition, the compiler sometimes uses a `pop` and `jmp` pair instead of a `ret` instruction. To address these exceptional cases, Das et al. proposed extended SCS rules [30]. Whenever an indirect `jmp` targets an address inside the SCS, the corresponding address is removed from the SCS. To improve the security of this approach, BBB-CFI limits `ret` and indirect `jmp` targets to the top 8 elements of the SCS [60]. This causes the matching element as well as the elements above it to be removed from the SCS. Special care needs to be taken when using this approach in combination with a fine-grained CFI policy, since an attacker could exploit this feature with a `ret` to any of the top 8 elements on the call stack.

Das et al. further noted that during software exceptions, the stack unwinding process is started, which frequently leads to a `ret` being used instead of a `jmp` to branch to an exception handler [30,60]. This causes a problem when using an SCS, since the target return address will not be on the SCS. To solve this, they proposed to allow `ret` to target any of the exception landing pad addresses. In x86 Executable and Linkable Format (ELF) binaries the exception information are stored in the read-only `eh_frame` and `gcc_except_table` sections.

Recursive functions

Recursive functions can be problematic, since invoking a function many times will require a large storage overhead for the SCS. HCFI supports recursive functions by assigning a one-bit recursion flag to each SCS element [27]. Before pushing a new return address onto the SCS, the address is first compared to the top element of the SCS. If the value is different, it is pushed. If the value is the same, then the top element's recursion flag is set. For each `ret`, the recursion flag remains set while the top element and the return addresses are the same. However, if there is a mismatch, the top element of the SCS is popped. This approach has the cost of storing one additional bit for every SCS element, which includes the hardware buffer and the storage in main memory. In [27], this was done using a 128*1-bit buffer, which restricts the call depth to 128.

Multiprogramming

Using a SCSs in a multithreading environment is problematic, since concurrency allows the `ret` of one thread to occur after the `call` of another thread, leading to SCS inconsistencies. Das et al. proposed to relax the CFI policy by allowing returns to target any address stored in the SCS [30]. This may reduce the security, since an attacker could re-order the sequence of returns. In hardware the top SCS elements can be stored in a Content-Addressable Memory (CAM) to accelerate searches. However, searching the SCS elements stored in main memory will incur a large performance penalty, since accessing main memory is usually slow. In the worst case the entire call stack will need to be searched.

To support multitasking, Das et al. proposed to store a process identifier in each SCS entry [30]. Each `ret` will only use the return address associated with its specific process identifier. This approach has the cost of storing the process identifier inside every element of the call stack, which is stored in the on-chip hardware buffer and the off-chip main memory. (A process identifier requires $\geq$ 16 bits on Linux.) In addition, the call stack will need to be searched for a specific process identifier during each return. Searches can be accelerated with a small CAM containing one element for each process identifier. If the process identifier is not found on the CAM, a huge performance overhead will be incurred, since the buffer and main memory will need to be searched.

Intel CET makes use of a Shadow Stack Pointer, which is a new architectural register that points to the top of the shadow stack. To address the problem of multiprogramming, it makes use of special instructions to switch the context of shadow stacks in both kernel-mode and user-mode. A detailed description of the shadow stack switching mechanism can be found in [63].

3.6.2 HAFIX: Shadow stack alternative

HAFIX proposed a stateful, fine-grained backward-edge policy which does not rely on an SCS [31]. To keep track of valid returns, a unique label is assigned to each function and a one-bit table entry is used for each label. During a function call the calling function's label is activated, while a return leads to the deactivation of the target function's label. During a return, if the target function's label is inactive, an exception is raised. To enforce this policy, HAFIX uses special instructions inserted in the binary to activate/deactivate the labels. The ISA was modified to only allow function returns to a special landing pad instruction which performs the check. Recursive function calls are supported by counting the number of times the function has been called. HAFIX improves upon an SCS by having a somewhat reduced memory storage overhead when compared to a SCS. Furthermore, since only one one bit of storage is required for each function in the program, the entire label state memory can be stored inside a hardware buffer.

HAFIX is integrated into the processor's pipeline stages. The major hardware components of the design consists of the CFI control logic and the label state memory, which consists of a hardware buffer with a one-bit entry for each function, as well as a counter used for suporting recursion.

3.6.3 Labels

HCFI [27] and Sullivan et al. [101] implemented Abadi's [5] label-based approach (see Section 3.5.1) in hardware through special instructions added to the ISA. Specifically, `SetLabel` is executed before each indirect forward branch to assign a new label identifier to the label register, while the `CheckLabel` instruction is executed as the first instruction after an indirect forward branch, and is responsible for verifying the label identifier stored in the label register [2]. Indirect branch targets are restricted to addresses containing the `CheckLabel` instruction.

It is important to highlight a security problem of the label-based policy. When multiple different indirect forward branches target the same basic block, the same label identifier will be used, since `CheckLabel` compares the label register identifier to a constant value. This means that all indirect forward branches targeting the same basic block need to assign the same label identifier before branching. To make this problem worse, an indirect branch can target many different basic blocks, which all need to use the same label identifier, since it

[2]For simplicity we used the instruction names `SetLabel` and `CheckLabel`, but HCFI named their instructions `SetPCLabel` and `CheckLabel`, while Sullivan et al. named their instructions `cfiprj/cfiprc` and `cfichk`.

is assigned before the branch. This causes imprecision in the enforced CFG, since the same label identifier is used for multiple basic blocks, thereby allowing many more edges than in the original CFG. This problem can be prevented by using a *trampoline* for each basic block targeted by multiple call sites [101]. The code is transformed so that each indirect branch targets a unique trampoline. Each trampoline executes `CheckLabel`, and then performs a direct jump to the original target. This ensures that the precision of the enforced CFG is not reduced by large equivalence classes.

The designs of [27] and [101] are both integrated into the processor's pipeline stages. The major hardware components used in their designs are an SCS module, a label register, and logic for each special instruction.

3.6.4 Table

The *table-based* approach uses a table of allowed branches (source addr, dest addr), with a single entry for each direct branch, and possibly multiple entries for each indirect branch. At runtime, each branch is verified by checking for the existence of an entry inside the branch table.

This approach enforces fine-grained, static CFI, and requires a CFG to generate the branch table. The branch table could have a large storage requirement, and searching the table could incur a high performance penalty.

CONVERSE enforced the table-based approach with a watchdog processor [56]. It populates the branch table using both direct and indirect branches, by means of static and dynamic analysis. The architecture uses a watchdog processor to check the validity of each branch at runtime, as explained in detail in Section 3.7.1. The branch table is protected from attacks on the target processor, since it is stored in the memory space of the watchdog processor.

Arora et al. [12] proposed an architecture to enforce CFI and SI at runtime. Intra-procedural control flow is protected using the table-based approach, while inter-procedural control flow is protected using the *Finite State Machine (FSM)* approach (see Section 3.6.5). A hash is used to verify the integrity of the instructions in each basic block at runtime, and the processor is stalled to allow for completing the hash calculation before moving on to the next block. A program is loaded by first verifying the program and integrity of the metadata. Next, the hardware monitor's FSMs and tables are populated using the metadata, followed by executing the program. The architecture can eliminate a wide range of software and physical attacks, including physical attacks on control flow and tampering with the contents of basic blocks.

The design of Arora et al. was integrated into the pipeline stages of the processor. The main hardware components are FSMs, lookup tables, buffers, a hash engine, and control logic. New FSMs and lookup tables are generated for each program, which means that each program has a different hardware area requirement. Inter-procedural checking can be implemented with a maximum area overhead of 1.2%, while the area increases to 5.17% when also adding intra-procedural checking, when compared to the die size of an ARM920T processor. When also using a hash engine, the area overhead increases to 9.1%.

3.6.5 Finite State Machine (FSM)

A FSMs can be used to enforce a valid sequence of events. While a program is executing, a state machine tracks the events, with each event representing a node in the FSM. As long as the events inside an executing program follow the correct sequence, the state transitions will be valid. However, if an invalid state transition occurs, an attack is assumed, and appropriate action is taken.

Arora et al. used an FSM to detect invalid control flow between functions [12]. Each node in the FSM represents a function, and each edge represents control flow between functions. A function call graph is extracted from a program, with each function call or return corresponding to a state transition. For more information on this architecture, see Section 3.6.4.

Rahmatian et al. used an FSM to detect invalid system call sequences [90]. The observation is that malicious code must invoke system calls to perform some of the necessary operations to launch an attack. If the sequence of system calls deviates from the FSM, an attack is assumed. A model of the legal sequences of syscalls is derived from a CFG, and the policy is enforced with an FPGA. To detect a syscall, the architecture made modifications to the instruction pipeline to detect a trap instruction (SPARCv8). The authors reported that the architecture has no performance overhead, and that illegal system call sequences can be detected within three processor cycles. The system relies on software to update the FSM stored on the FPGA at runtime. The major hardware component of the design is the syscall sequence FSM, which includes two 36Kb block RAM modules. A custom FSM is generated by static analysis, leading to different hardware requirements for different programs.

3.6.6 Heuristics

Heuristic-based approaches detect CRAs by deriving a CRA signature from the branching behaviour of a program under attack. A popular CRA signature is

the *number of executed instructions* between branches. The assumption is that typical ROP and JOP gadgets usually consist of a small number of instructions (around five instructions). Larger gadgets usually have side effects, such as updates to memory and registers, and are therefore avoided by attackers. A typical attack executes a number of short instruction sequences that each end with a branch, whereas a normal program executes larger instruction sequences between branches. Chen et al. [24] reported that ROP attacks require at least three gadgets, while Kayaalp et al. [67] reported that JOP attacks require at least six gadgets. This approach requires no CFG.

A major challenge is the proper selection of heuristic parameters (e.g., number of instructions between branches and chain length) that minimises the number of false positives. The number of false positives typically increases as the gadget length is incremented or when the chain length is reduced. In addition, the number of false positives can differ between different programs, while using the same parameter set.

Recent attacks demonstrated that heuristics which only assume short gadgets can be circumvented [22, 32, 52]. Detection can be avoided by placing an intermediate gadget (a gadget which doesn't do anything but is longer than the threshold of the heuristic) in the gadget chain. This is non-trivial, since the side effect(s) of the long gadget needs to either be tolerated, or subsequent gadgets should be used to undo the side effect.

To defend against intermediate gadgets, SCRAP proposed a *multi-threshold detector*, which tolerates longer gadgets in the gadget chain [67]. This is done by not advancing the gadget count when detecting an intermediate gadget. They demonstrated that they can protect the entire SPEC 2006 benchmark suite without any false positives. In addition, SCRAP limits false positives by allowing the chain length and instruction count to be configured at runtime. SCRAP is integrated into the commit pipeline stage of the processor. The major hardware components of this architecture is an SCS module and the multi-threshold heuristic logic which consists of several state machine counters. A simulation using PTLsim/x86 showed a performance overhead results of around 2%.

The number of *consecutive indirect branches* can also be used as a CRA signature. The observation is that a well-behaved program executes a mix of direct branches as well as indirect branches, since a normal program usually contains many more direct branches than indirect branches. Therefore, to detect an attack, the heuristic counts the number of consecutive indirect branches. If more than γ indirect branches are consecutively executed, then the program is assumed to be under attack. While this rule can detect most CRAs, adversaries can circumvent it by using a *gadget gluing attack* [26]. Here, the attacker places a

special gadget containing a direct branch inside the gadget chain to thwart the heuristic. To address this issue, Lee et al. proposed to use a second threshold parameter δ, which represents the maximum number of direct branches that are allowed between indirect branches [77]. Therefore, an alarm is only raised if the branch trace contains more than γ indirect branches and at most δ direct branches.

Lee et al. proposed a *two-stage heuristic* policy, where a mixed hardware/-software approach is used for detecting CRAs in a resource-constrained environment [77]. In the first stage, a lightweight hardware monitor employs a multi-threshold heuristic based on the number of consecutive indirect branches. When the first stage detects anomalous behaviour, an interrupt is raised, and the second stage software performs an in-depth analysis on the occurred branching behaviour, in order to ensure that the detected operation is not a false alarm. The idea of the two-step approach is to increase the system performance by using an efficient and lightweight monitor on all branches, while performing an in-depth analysis only when exceptional program behaviour occurs. During the first stage, the hardware monitor stores a trace of all indirect/direct branches inside the branch history buffer. The second stage then uses the branch history buffer to enforce a number of rules, such as: (1) returns should always point to call-preceding instructions, (2) indirect calls target only function entries, and (3) the number of instructions between indirect branches are usually large. A hardware-based performance evaluation showed a performance overhead of $<1.5\%$ for $\gamma>4$ and $\delta<3$. The major hardware costs are: a 32-entry branch history buffer First-In, First-Out (FIFO) (which stores the source address, target address, and branch type for each entry), an FSM, and three counters. This design was implemented using the debug interface, and is described in Section 3.7.1.

3.6.7 Monitoring graph (MG)

Mao et al. proposed to derive an information stream from the executing program to detect attacks, which they call a Monitoring Graph (MG) [81]. The stream is derived from a combination of any of the following: the address pattern, the opcode pattern, the load/store pattern, the control flow pattern, or a hash of the opcode and instruction address. The expected program behaviour, called a monitoring graph, is generated through analysis and simulation, and is then stored in memory. At runtime, the derived stream is compared to the monitoring graph. If there is a mismatch, an attack is assumed and the processor is interrupted. The architecture can detect a stack smashing attack in one to ten cycles, depending on which information is used to derive the stream. The authors reported a monitoring graph size of 100 kB compared to an application

binary size of 5 MB. SI can be guaranteed when the derived stream is based on the opcode pattern. The design consists of a watchdog processor connected to the target processor via an unspecified interface.

3.6.8 Branch Regulation (BR)

Branch Regulation (BR) protects forward edges by enforcing a simple invariant rule for branch targets, by disallowing arbitrary branches between functions [66]. The enforced invariant rule states that, during normal program execution, a `call` always targets a function entry point, while an indirect `jmp` targets either a function entry or an address within the current function. The enforcement of this rule severely limits JOP attacks since most functions lack a dispatcher gadget (see Section 3.2), which is critical to launch a JOP attack. The advantages of BR is that it requires no CFG, and it's simple, efficient, lightweight, and severely reduces the set of available gadgets.

BR claims to detect all branches, including unintended branches [66]. However, it seems to provide only limited protection against unintended branches, since indirect branches into the current function are never checked. Kayaalp et al. reported that BR reduced the number of exploitable gadgets to 1% of available gadgets in the original binary [66]. This authors also argue that unless the attacker can find gadgets that execute system calls, the damage from any attack is limited to the compromised process. However, the C standard library contains many system calls, and it seems like nothing prevents the attacker from invoking a function which uses a syscall.

A challenge of using BR is to communicate the function boundaries to the hardware monitor. Kayaalp et al. proposed to communicate each function's bounds with an annotation placed at the first address of each function header [66]. Each `call` instruction can only target annotated addresses, while `jmp` instructions can target either annotated addresses or addresses inside the current function. After branching to the annotated address, the instruction pipeline processes the annotation and places the new function bounds inside a Function Bounds Stack. Lee et al. proposed to transform each function to start with annotated code which communicates the function bounds by writing to MMIO (see Section 3.7) [74, 76].

Kayaalp et al. integrated BR into the execution pipeline stage of a processor [66]. The major hardware components are an SCS, memory for storing the Function Bounds Stack, and logic for interpreting the annotations and detecting invalid branches. The performance evaluation was done in simulation using PTLsim/x86, and they found that an Function Bounds Stack of 8 entries (total size 96 bytes) leads to a performance loss of about 1%.

3.6.9 BB-CFI: Branch Regulation on Basic Blocks

BB-CFI [30] limits branch targets to basic block boundaries, instead of function boundaries (as in BR [66]). Specifically, branches are only allowed to target the first instruction inside a basic block. In addition, a `call` is only allowed to target the first basic block of a function. The authors argue that restricting `jmp` targets to the current function boundary (as was done in BR) is too restrictive, as it cannot support `longjmp()`. A CFG is used to generate metadata containing the start address of each basic block as well as the start of each function. BB-CFI protects against unintended branches, since control can only flow to the first instruction in each basic block. Unlike BR, BB-CFI does not rely on non-executable memory. However, the metadata needs to be stored in protected memory. For the evaluation, the authors stored the metadata in hardware on content addressable memory, but it can also be protected by storing it in kernel space, which would incur additional overhead. The authors found that, on average, >99% of gadgets were eliminated from various different benchmarks. The RIPE benchmark [110], which performs exploits based on code injection, return-into-libc and ROP (a large number of different CRA attacks), was used to evaluate BB-CFI. The authors found that all attacks were blocked (both when $W \oplus X$ is enabled and disabled). It remains to be seen whether a specifically crafted attack can circumvent this security policy.

BB-CFI is integrated into the processor's commit pipeline stage. The major hardware components of the design are an SCS module, the Basic Block Table (which contains the profiling metadata), a control unit, and a Target Address Buffer, which is a buffer containing control flow instructions that still need to be validated. The Basic Block Table is implemented as CAM, which can search for a target address in the entire Basic Block Table in a single cycle, and has a maximum memory requirement of 209 kB for one of the measured SPEC benchmarks. A Target Address Buffer of 1 kB has an estimated a performance overhead of <1%.

3.6.10 Branch Limitation (BL)

Recent works [60, 63] proposed to limit branch targets to basic block entries, which we call Branch Limitation (BL). This restriction ensures that control can only flow from the exit (last instruction) of one basic block to the first instruction of a basic block. When a branch targets the middle of a basic block it violates the basic block definition, implying that the system is under attack. The BL architectures discussed here only checks forward indirect branches, but can be combined with an SCS to check backward edges.

Both BL and BB-CFI enforce the semantics of basic blocks that are targeted by indirect branches. However, they use different mechanisms for defining the basic block entry points. In BB-CFI, the basic block entries are stored in metadata which was generated from a CFG, while in BL the entries are indicated by instructions in the binary. The advantage of this approach is that it requires no CFG.

A major challenge of BL is to communicate the first address of a basic block with the hardware monitor. BBB-CFI argues that confirming a basic block's exit is equivalent to confirming an entry point of the following basic block [60]. Since a branch can only be placed on the last instruction of a basic block, the following instruction is usually the entry point of another basic block. Therefore, they proposed to restrict indirect branch targets to addresses which are preceded by any branch instruction, including direct/indirect calls, jumps and returns. This ensures that indirect branches can only target the first instruction of a basic block. To support basic blocks which are not preceded by branches, such as fall-through edges of switch-like statements, a second rule is introduced. Here, the the fall-through addresses are stored in metadata (inside main memory). At runtime, if the first rule fails, then the second rule is evaluated. The enforcement of both rules requires performing additional reads: rule one requires fetching the instruction preceding the branch target, while rule two requires reading metadata from memory. The authors reported a 90% reduction in JOP gadgets, and a total gadget reduction of 98.68% when also enforcing an SCS with extended rules. A RIPE evaluation showed that the only successful attacks were return-into-libc attacks which hijack an indirect call to invoke an exported library function. These attacks avoid detection because the call targets a function entry, which seems like a normal control flow to the CFI architecture. The authors reported an average performance overhead of 0.1%.

BBB-CFI is integrated into the processor's pipeline stages [60]. The major hardware components are a 32 element SCS, a buffer containing the most recently verified basic block boundaries, and control logic to compare and fetch metadata from main memory. The fall-through metadata are stored in main memory, with a storage overhead of around 13% when compared to the original program size.

Intel Control-flow Enforcement Technology (CET) solves the challenge of communicating the basic block entry points with the hardware monitor with *indirect branch tracking* [63]. Here a new instruction, called `ENDBRANCH`, is placed at the entry of each basic block that can be invoked through an indirect forward branch. When an indirect forward branch occurs, the following instruction is expected to be an `ENDBRANCH`, otherwise an attack is assumed. Since this approach is so similar to BBB-CFI, which also enforces the semantics of basic blocks, it is expected to lead to the same gadget reduction count as BBB-CFI.

Intel CET is integrated into the pipeline stages of the processor. The major hardware components are an SCS module and a small indirect branch tracking FSM.

3.6.11 Instruction Set Randomisation (ISRAND)

SOFIA performs Instruction Set Randomization (ISRAND), where each instruction's bytes are encrypted using control flow information from a CFG, to enforce a fine-grained CFI policy [35]. At runtime, each instruction's bytes are decrypted using a combination of the current and previous program counters. Therefore, any invalid control flow between two instructions will lead to a decryption error. To detect decryption errors and also provide SI, a Message Authentication Code (MAC) is calculated over the instruction bytes in each basic block. A unique key is used for each device, which can only be accessed by the cryptographic hardware, and is only known by the software provider. It is therefore impossible for an attacker to inject code or tamper with the software since he doesn't know the key. SOFIA checks all branches, including unintended branches and fault attacks on the control flow. SOFIA enforces a static CFI policy (no SCS). However, it would be difficult to craft a ROP attack, since all software is stored encrypted, making it near impossible to identify gadgets. SOFIA has an average performance overhead of 106%.

An interesting feature of SOFIA is that it does not only check the validity of control flow between basic blocks, but also verifies the control flow between every two instructions. This is important in scenarios where one needs protection against fault attacks on the control flow, or where one cannot rely on the OS to protect the integrity of instructions.

SOFIA is integrated into a processor's pipeline stages [35]. The major hardware components are a MAC component (which consists of a two cycle block cipher), as well as control logic. The MAC component has a critical path which is longer than that of the processor, leading to a clock speed reduction of 23.3%.

3.6.12 Signature Modeling (SM)

Signature Modeling (SM) is a technique used in fault-tolerant computing to detect control flow violations. A checksum is periodically calculated on the executed instructions for comparison with stored reference values. In its simplest form, the comparison is done at the end of each basic block. The reference values are computed offline and stored inside the binary.

In *Continuous Signature Monitoring*, a signature of each executing instruction is verified at runtime [111]. The signature is updated (e.g., with a XOR) and verified before the execution of each instruction, which facilitates the enforcement of SI. The signature depends on both the current instruction, as well as previously executed instructions, and it also captures errors in control flow, since a tampered control flow change will eventually lead to different instructions being executed. To ensure that the signature size does not become too large, only a small amount of signature bits are stored and checked for each instruction. Since the error propagates within the signature register, almost all control flow errors can be detected within 3 checks for a signature check size of only 4 bits. The advantage of this approach is that it can detect control flow changes at a fine granularity, such as when a fault attack is used to skip a single instruction inside a basic block. The disadvantage is that a signature value for every instruction is necessary.

Generalized Path Signature Analysis relies on signature updates in the code to ensure that a signature value at a given location is always the same, no matter which (valid) path was taken through the CFG [80]. First, the CFG is divided into a number of path sets, with each path set starting and ending at the same node. The goal is then to have a single signature value for each path set no matter which control flow path was taken. To achieve this, justifying signatures are inserted on some of the paths of each path set. When control flow arrives at the end of each path set, the signature is compared to the reference value. This lowers the storage requirement for reference values, as integrity checks occur less frequently. However, this method increases the latency between fault and detection.

Werner et al. use Generalized Path Signature Analysis together with Continuous Signature Monitoring to protect software running on an embedded processor against fault-based control flow attacks [109]. For Continuous Signature Monitoring, the processor's fetch unit is modified to also fetch the reference signature. Only when the reference signature matches the current signature will the instruction be forwarded to the decode stage. A post-processing tool is used to obtain a CFG, and calculate the derived signatures. In addition, binary instrumentation is used to send commands to the hardware to update/check the signature (by writing to a memory mapped register). All branches (direct and indirect) are checked, since any control flow change causes an update to the runtime signature. They reported a performance overhead of 9%.

Werner et al. integrated their hardware monitor into a commercial ARM Cortex-M3 processor. The signature monitor is implemented as a memory mapped device [109]. The major hardware components are a CRC-based signature monitor and logic to fetch reference values from main memory and compare them against the derived signatures.

3.6.13 Code Pointer Integrity (CPI)

Code Pointer Integrity (CPI) policies aim to prevent control flow hijacking by protecting the integrity of code pointers at runtime.

A recent whitepaper discusses hardware-based CPI support on the new ARMv8.3-a ISA [4]. Here, short MAC tags, called Pointer Authentication Codes (PACs), are used to verify the integrity of pointers at runtime. The MAC is calculated over the pointer target together with a context, which is usually the pointer address, e.g., PAC = MAC_k(target,context). Whenever a code pointer is used, its integrity is first verified, and whenever a pointer target is updated, a new PAC is calculated and stored. This prevents pointer tampering, since an attacker will also have to forge the PAC. Relocating a PAC and pointer pair to a new address is infeasible, since the MAC computation includes the pointer address. Key management is done by privileged software (EL1, EL2, EL3), and it is expected that the higher privilege levels control the keys for the lower privilege levels, which includes assigning a unique key per each process or per each boot.

PAC exploits the fact that the available virtual address space in 64-bit architectures is less than 64-bits, e.g., ARM64 Linux uses a 40-bit address space by default, leaving 26 bits to be used for storing PAC values. The PAC size can vary between 3 to 31 bits, depending on the system's virtual address space configuration, and the PAC is placed in the unused upper bits of the pointer before being written to memory.

The core functionality of PAC is provided by two instructions types: `PAC*` computes and stores a PAC, while `AUT*` verifies a PAC and restores the pointer value. An integrity failure during `AUT*` leads to a pointer update to an illegal address, which causes an exception when the invalid pointer is dereferenced. The idea is that the compiler inserts these instructions inside the binary in order to update and check the integrity of the critical pointers at runtime. PAC is integrated into the pipeline stages of the processor, since it uses dedicated instructions. The major hardware components are a low latency MAC implementation which uses the QARMA block cipher, together with control logic to calculate and verify PACs.

It is important to note that PAC cannot guarantee the freshness of an authenticated pointer. This makes PAC susceptible to substitution attacks, where one authenticated pointer is replaced with another which was previously stored on the same address.

Software-based CPI enforces pointer integrity by storing sensitive pointers in an isolated memory region, and further uses runtime checks to verify the correctness

of each code pointer on each control transfer [73, 82, 102]. Software-based CPI which rely on information hiding have been demonstrated to be broken [48]. However, other instruction-level isolation mechanisms, such as software fault isolation [73], provide much stronger protection for storing pointers, and have not been demonstrated to be broken. PAC is unlikely to be susceptible to these attacks, since each stored pointer is protected by a cryptographic MAC [4].

3.7 CFI enforcement via the debug interface

A common approach to enforce CFI through the debug port is to configure the debugger to provide a trace for each control flow transfer. However, the traces often do not provide all the required branching information to enforce a CFI policy. A simple solution is to modify the debugger to include the missing information inside the debug trace [56, 60, 74]. However, this is not ideal, since the main reason for using the debug interface is to avoid having to modify existing IP (see Section 3.3.3). Another approach is to provide supplementary metadata inside main memory [75], or to communicate the missing information with the hardware by writing to memory-mapped addresses [76, 77].

3.7.1 Implementations

Lee et al. observed that the debug traces generated by an ARM CoreSight debugger contain only the branch target address for indirect branches, while for direct branches only the direction (taken/not taken) is available [75]. In order to enforce an SCS, the branch type and source address are also necessary. To this end, offline analysis on the binary is used to generate supplementary metadata. The metadata contains the branch type, source address, and target address for each branch instruction (at the end of each basic block). By observing the debug traces, the hardware monitor can determine which basic block is currently executing, thereby allowing accurate enforcement of the SCS policy. The metadata increased the binary size by 145.5%, while incurring a performance penalty of 2.4%, which is mostly due to bus contention since the main memory is shared between the monitor and the processor. The major hardware components are a 16 entry branch trace FIFO, a debug trace analyser, logic to read metadata from main memory, and an SCS module. The hardware monitor is implemented in reconfigurable fabric, which runs at a lower clock frequency than the ASIC processor. They reported an operating frequency of 90 MHz for the ROP monitor, and 200 MHz for the host processor.

Lee et al. proposed to supplement ARM CoreSight's trace information by exploiting the fact that the target address is only available for indirect branches [76, 77]. To do this, a trampoline is inserted for each `call` in the binary, together with replacing each `call` instruction with an indirect jump to the unique trampoline address associated with that call. Since each trampoline has a unique address, the branch type and source address can be derived by checking the target address of the direct jump. This approach has a code size increase of 16.6%. The major hardware components are a 32-entry branch trace FIFO, a debug trace decoder logic, a 32-entry MMIO FIFO, a branch trace analyser, a FIFO trace combiner, a CRA detector controller, and an SCS module. The branch trace FIFO acts as a buffer for receiving debug traces, while the MMIO FIFO acts as a buffer for receiving the function bounds via MMIO. On the Zynq platform, the CRA monitor runs at 60 MHz, while the ARM processor runs at 150 MHz. They reported a performance overhead of 3%, which is mostly due to memory contention.

Lee et al. enforced BR and an SCS through the Core Debug Interface [74]. The debug interface is modified to provide more information for the hardware monitor, and MMIO is used to receive function boundary information from the instrumented binary. They reported an average performance overhead of less than 2%. The major hardware components are a debug trace filter, an indirect branch bounds checker, an internal bus, main control logic, the memory region protector, a secure call stack, and a debug trace FIFO.

CONVERSE enforced a table-based policy through a commercial off-the-shelf watchdog processor that is connected to the target processor's Nexus 5001 debug port [56]. Each debug trace contains the source and target addresses. When the watchdog processor detects of invalid control flow, the debug `BREAK` feature is used to halt execution of the target processor. The authors reported that their implementation does not have any overhead.

BBB-CFI enforced an SCS and BL by utilising Intel's Last Branch Register (LBR) and Processor Trace (PT) features [60]. The LBR buffers the 16 most recent traces, with each trace consisting of the source and target address, while the PT is used to write out a trace to physical memory or to a dedicated port on a SoC. BBB-CFI modified the LBR to include a 2-bit branch type field.

3.7.2 Limitations

A common problem with enforcing a CFI policy via the hardware debugger is that the hardware monitor could drop traces given a sufficiently high branch rate [56, 77]. This happens when the hardware monitor requires more time to process a trace than the rate at which branches occur on the target processor.

Whenever a trace is dropped without being analysed, it introduces a security weakness, since an attacker could exploit this by launching an attack during a time period when branches are frequently occurring. This problem can be mitigated by using a FIFO to store all incoming debug traces before they are processed. However, given a sufficient number of branches over a short time period, the hardware monitor will eventually fall behind, leading to dropped traces. This is especially problematic if the hardware monitor operates at a much lower clock rate than the processor.

Another problem is that the hardware debugger can be used by an attacker to circumvent the security of the system. If the attacker can access the debug interface, he could use it to tamper with code and data memory, or even disable the hardware monitor by tampering with the tracing mechanism. Therefore, care needs to be taken to ensure that an attacker cannot obtain access to the debug interface. This could be done by ensuring that no external debug interface is present, and prohibiting software running on the processor from accessing the debugger.

3.8 Comparison of Architectures

Table 3.1 shows a comparison of all hardware-based CFI architectures analysed in this chapter. Most architectures enforce dynamic CFI which relies on an SCS for stateful backward edge protection, while using a range of different static forward-edge policies.

3.8.1 Protection provided

Most works assume attackers in control of data memory, while a smaller number of architectures assume an attacker that can control code memory or perform fault attacks on control flow. Whenever an attacker model wasn't specified, we made an estimate of the assumed attacker, which is indicated by "*". The following text discusses the protection provided in terms of the assumed attacker model.

A policy which prevents arbitrary branches imposes limits on the allowed branch targets of forward branch instructions. We use the *"Indirect Branch Protection"* and *"Direct Branch Protection"* columns to indicate whether an architecture can prevent arbitrary branches to respectively for indirect or direct instructions. Many architectures can prevent arbitrary indirect branches, while only some can prevent arbitrary direct branches. Indirect branch protection is important to

Table 3.1: Overview of hardware-based CFI architectures.

Architecture	Policies		Protection Provided						Requirements					
	Stateful policy	Static policy	Attacker Capabilities[1]	Indirect Branch[3]	Direct Branch[3]	Data structures[2]	Exploitable Gadgets[4] (%)	Software Integrity	CFG	SW in TCB	ISA changes	Debug Port	W ⊗ X	Academic
SmashGuard [88]	SCS	○	D	◐	○	○	–	○	○	○	●	○	●	●
IBMAC [50]	SCS	○	D	◐	○	●	–	○	○	○	●	○	○	●
Lee et al. [75]	SCS	○	D	◐	○	○	–	○	○	○	○	●	●	●
HAFIX [31]	●	○	D	◐	○	↯	–	○	○	●	●	○	●	●
HCFI [27]	SCS	Label	D	●	○	●	0*	○	●	●	●	●	○	●	●
Sullivan et al. [101]	SCS	Label	D	●	○	●	0*	○	●	●	●	●	○	●	●
Arora et al. [12]	SCS	Table + FSM	CDF*	●	●	◐	0*	●	●	●	○	●	○	○	●
CONVERSE [56]	○	Table	–	●	●	●	0*	○	●	●	○	●	●	○	●
Rahmatian et al. [90]	○	FSM	CD	◐	○	○	100*	○	○	●	●	●	○	○	●
SCRAP [67]	SCS	Heuristic	D*	◐	○	○	–	○	○	○	○	●	○	○	●
Lee et al. [77]	○	Heuristic	D	◐	◐	○	–	○	○	○	●	○	●	●	●
Mao et al. [81]	SCS	MG	CDF*	●	●	○	0*	●	●	●	○	○	○	○	●
Werner et al. [109]	SM	SM	CDF*	●	●	○	0*	●	●	●	●	●	○	○	●
BR [65, 66]	SCS	BR	D	◐	●	●	1	○	○	○	●	●	○	●	●
Lee et al. [76, 77]	SCS	BR	D	◐	●	○	1*	○	○	○	●	○	●	●	●
Lee et al. [74]	SCS	BR	D*	◐	●	●	1*	○	○	○	●	●	●	●	●
BB-CFI [30]	SCS	BB-CFI	CD	●	●	◐	1	○	○	●	○	●	○	○	●
BBB-CFI [60]	SCS	BL	D	●	○	○	1.3	○	○	○	○	–	●	●	●
Intel CET [63]	SCS	BL	D*	●	○	●	1.3*	○	○	○	●	●	○	●	○
SOFIA [35]	○	ISRAND	CDF	●	●	●	0*	●	●	●	○	●	○	○	●
ARMv8.3-a [4]	CPI	CPI	D	●	○	●	0*	○	●	○	●	●	○	●	○

● = Yes; ◐ = Partial; ○ = No; ↯ = Not Applicable; – =Unspecified; *= Estimated
[1]D = Data modifications; C = Code modifications; F = Fault control flow
[2]Metadata, SCS, or runtime data structures access protection.
[3]Prevent forward branches to arbitrary target addresses; [4]Exploitable Gadgets in the binary

protect against JOP and ROP attacks, since these attacks make use of gadgets consisting of indirect branches. Direct branch protection provides an additional protection layer, and is important in the following two scenarios. First, when the attacker controls code memory direct branch targets could be modified, since code is mutable. Second, when fault attacks on the control flow occur, direct branch targets could be tampered with. BR provides only partial indirect branch

protection, since indirect branches to any address within the current function are allowed. This leaves BR somewhat vulnerable to unintended branches, since it allows CRAs which do not cross function boundaries.

Many solutions store sensitive runtime data structures or metadata in data memory. However, less than half the architectures protect both their runtime (SCS) data and metadata which is stored in main memory. Since all attackers can control data memory, it is important to protect sensitive data stored in main memory in order to prevent attackers from circumventing the security policy by tampering with the stored data.

Coarse-grained policies cannot prevent all indirect branches from disobeying the CFG. Hence, their program binaries contain a number of gadgets which are usable by an attacker. To quantify the number of usable gadgets, a metric called gadget reduction or *Average Indirect target Reduction* is sometimes reported, which we summarised in the "*Exploitable Gadgets*" column. The metric reports the ratio between the number of reachable gadgets when CFI is enforced, compared to the reachable gadgets when CFI is not enforced (i.e all gadgets). Gadgets are identified under the assumption that each gadget ends with an indirect branch and has no side effects, which can be automated with a gadget finder tool, such as ROPGadget [93]. It is important to remember that a successful CRA requires only a handful of gadgets. Therefore, even with a large gadget reduction, attacks can still succeed, albeit with more effort from the attacker. In some cases the values reported in the "Exploitable gadgets" column are estimated from the reported values of other architectures with the same policies. E.g., the values for [63] were based on [60] which both enforced BL, while [74, 76] are based on [66], since they enforce the same policies. Heuristic-based approaches, such as SCRAP [67], cannot prevent branches, since they only detect abnormal branch behaviour, and therefore almost all gadgets are exploitable. Fine-grained policies prevent arbitrary branches, and therefore the exploitable gadgets are estimated to be close to zero.

Most works assume attackers in control of data memory, while not controlling code memory, since it is protected with page-based protection, such as $W \oplus X$. In addition, these works only protect indirect forward branches. However, architectures that assume attackers that also control code memory, typically prevent code tampering/injection through an SI mechanism, which verifies code integrity at runtime by means of a cryptographic hash/MAC.

3.8.2 Requirements

This section compares the architectural requirements to the protection provided.

We found that roughly a third of the architectures offer fine-grained protection which relies on a precise CFG, as indicated by the *"CFG Required"* column. Fine-grained policies supposedly provide stronger security guarantees, while the security of the policy depends on the accuracy of the CFG. However, their dependence on a precise CFG is an important limitation, since it is difficult to obtain a precise CFG from complex binaries (see Section 3.5.3). We found that most architectures which do not rely on a precise CFG are coarse-grained, and therefore have a non-zero amount of exploitable gadgets. The exception seems to be ARMv8.3-a which is the only architecture which provides fine-grained protection while not relying on a CFG. However, currently we cannot thoroughly evaluate the architecture, since official reference material has not been released yet.

More than half of the architectures rely on software placed inside the binary to implement its security policy, as indicated by the "*SW in TCB* " column. This is a reasonable requirement, provided that code integrity is guaranteed, e.g., through $\mathrm{W} \oplus \mathrm{X}$ or by enforcing SI. All architectures respected this requirement.

We found that most architectures are integrated into the instruction pipeline stages of the processor, as indicated by the *"ISA changes"* column. In contrast, some architectures exploit the debug interface to allow enforcing their policies, as indicated by the *"Debug Port"* column. The main advantage of using the debug port is that it allows for enforcing a policy without making changes to the processor or other existing IP. However, we found that some architectures enforced via the debug port also modified the ISA to enforce their policies.

We found only two non-academic architectures, namely Intel CET [63] and ARMv8.3-a [4], as indicated by the *"Academic"* column. At the time of writing, Qualcomm has only released a whitepaper describing the PAC architecture, while Intel has released a detailed "Technology Preview" document. Intel's solution, which performs a type of Branch Limitation (BL), is coarse-grained, but practical for widespread deployment, since it only requires the insertion of special landing pad instructions. ARM's solution, which relies on CPI , promises to offer fine-grained protection, and seems to be practical since it does not require complex binary transformations. Both solutions do not rely on CFGs, which make these architectures practical for widespread deployment.

3.8.3 Overhead

Table 3.2 compares the performance and area overhead of the CFI architectures which were implemented on a FPGA or Application-Specific Integrated Circuit (ASIC). The architectures which have no published performance or hardware

overheads at the time of writing were not listed in the table, namely Mao et al. [81], Intel CET [63], and ARMv8.3-a [4].

Szekeres et al. [102] found that security mechanisms with a performance overhead of more than 10% do not gain widespread adoption in production environments. In addition, many people believe that the average performance overhead should be less than 5% in order to obtain adoption from industry. The *"Performance Overhead"* column shows that most architectures fall below the 5% barrier.

The majority of the architectures were evaluated for both performance and area in hardware. However, some architectures [65, 66, 88] were only evaluated in simulation, while some others [12, 30, 60, 67] performed a simulation-based performance evaluation, and further made a separate hardware implementation to evaluate the hardware area overhead.

Many architectures performed an evaluation on FPGA, but failed to specify the target FPGA technology, which makes it impossible to compare their hardware overhead with other designs. Some designs report their area (LUTs, registers, block RAM, gates) as an increase in area compared to the original unmodified processor. This also makes it difficult to compare the overhead with other designs, since the area used by different processors differ greatly. In addition, large variations in area can be observed for the same processor under different configurations. Many architectures are integrated into the pipeline stages of the processor, and it is important to know if the design has an impact on the critical path of the processor. Therefore, it is best to report the maximum attainable clock speed of the modified processor compared to the stock processor.

3.9 Conclusion

In this chapter, we presented an analysis of the security policies used by 21 hardware-based CFI architectures. This included a detailed comparison of the used policies with respect to their security, limitations, hardware cost, performance, and practicality.

The primary goal of CFI architectures is to protect against control flow hijacking attacks. As such, a fundamental limitation of CFI is that it cannot protect against non-control data attacks, which do not lead to the execution of invalid edges, but instead tamper with the order of the executed paths. We further found that most architectures provide backward edge protection with a Shadow Call Stack (SCS), and a large body of work discusses the intricacies of enforcing an SCS. However, for forward edge protection a range of different policies are used, which has different security and practical limitations.

Table 3.2: Performance and hardware overheads of the CFI architectures. All reported percentages are relative to the baseline performance of the target processor, while the non-percentages are absolute values.

Architecture			Perf. Evaluation			FPGA					ASIC			
	Static policy	Target ISA	Perf. Overhead (%)	Simulation†	Technology	LUTs	Flip Flops	36K BRAM	Area (%)	Clock (MHz)	Gates (kGE)	Technology (nm)	Area (%)	Clock (GHz)
SmashGuard [88]	○	Alpha	2.8%	●	–	–	–	–	–	–	–	–	–	–
IBMAC [50]	○	ATMega103	–	○	Cyclone-2	215	0	0	9%	–	–	–	–	–
Lee et al. [75]	○	Cortex-A9	2.4%	○	Zynq	7362	–	5	–	90	86.7	45	–	–
HAFIX [31]	○	LEON3	2%	○	–	0.3%	3%	8% *	–	0%	–	–	–	–
HAFIX [31]	○	Siskiyou Peak	2%	○	–	<1%	2.5%	2	–	–	–	–	–	–
HCFI [27]	Label	LEON3	1%	○	–	2.6%	2.5%	–	–	0%	–	–	–	0%
Sullivan et al. [101]	Label	LEON3	1.8%	○	–	–	–	–	–	–	–	32/28	1.78%	3
Arora et al. [12]	Table + FSM	ARM920T	–	●	Virtex-2P	1839	–	–	–	57	–	130	9.07%	–
CONVERSE [56]	Table	–	0%	○	–	–	–	–	–	–	–	–	–	–
Rahmatian et al. [90]	FSM	LEON3	0%	○	Virtex-6	340	–	2	1.9%	0%	–	–	–	–
SCRAP [67]	Heuristic	x86	2%	●	Spartan-3	–	–	–	–	284	–	–	–	–
Lee et al. [77]	Heuristic	Cortex-A9	1.5%	○	Zynq	1795	–	2	–	60	–	–	–	–
Werner et al. [109]	SM	Cortex-M3	9%	○	–	–	–	–	–	–	–	130	6.4%	–
BR [65, 66]	BR	x86	2%	●	–	–	–	–	–	–	–	–	–	–
Lee et al. [76, 77]	BR	Cortex-A9	4.5%	○	Zynq	3172	–	3	–	60	–	–	–	–
Lee et al. [74]	BR	LEON3	2%	○	Virtex-5	22.7%	–	15%	–	–	244.2	45	13.79%	1
BB-CFI [30]	BB-CFI	x86	<1%	●	Virtex-5	86	–	4	–	313	–	65	0.02%	–
BBB-CFI [60]	BL	x86	0.1%	●	Virtex-6	3720	3404	–	–	231	–	–	–	–
Intel CET [63]	BL	x86	–	○	–	–	–	–	–	–	–	–	–	–
SOFIA [35]	ISRAND	LEON3	106%	○	Virtex-6	958	467	0	–	23.2%	–	–	–	–

† Performance evaluation done in simulation, and not on the hardware of the target ISA
– =Unspecified; * = distributed RAM

The general conclusion is that substantial progress has been made in the area of CFI. In particular, SCS provides excellent protection against ROP, and does not suffer from any practical limitations which prevent widespread adoption. However, static forward-edge CFI policies appear to face the following practical limitations:

- The security of fine-grained CFI relies on the precision of a CFG, which cannot be accurately generated for programs that contain indirect forward branches. ARMv8.3-A seems to be the only architecture that can provide fine-grained protection without the need for a CFG. However, it has not yet been publicly released, which means that the security community still has to evaluate its limitations.

- Coarse-grained CFI relies on a relaxation in the strictness of its policy by enforcing simple rules that do not require a CFG. However, this comes

at a reduction in the security provided, since it cannot detect all illegal branches.

- Heuristics provide coarse-grained and practical protection. However, it can be circumvented by an attack which is crafted to thwart the heuristic. It further suffers from false positives, and proper heuristic parameter selection remains unsolved.

These limitations could explain why industry has not yet publicly released a CFI-capable processor, and we believe that the problem of enforcing practical and fine-grained CFI still remains unsolved.

In the following chapter we present our novel CFI architecture, called SOFIA (first introduced in Section 3.6.11), which enforces CFI by means of Instruction Set Randomization (ISRAND). Then, in Chapter 5, we will present a detailed description and evaluation of a novel Software Integrity (SI) architecture which does not require modification of any existing IP.

Chapter 4

SOFIA: Software and Control Flow Integrity Architecture

Content Sources

de Clercq, R., De Keulenaer, R., Coppens, B., Yang, B., Maene, P., de Bosschere, K., Preneel, B., de Sutter, B., and Verbauwhede, I. SOFIA: Software and Control Flow Integrity Architecture. In *Proceedings of the Conference on Design, Automation & Test in Europe* (2016), DATE '16, IEEE, pp. 1172–1177
Contribution: Responsible for hardware designs. Some concepts are the result of brainstorming sessions with co-authors.

de Clercq, R., Götzfried, J., David, U., Maene, P., and Verbauwhede, I. SOFIA: Software and Control Flow Integrity Architecture. In *Computers & Security* (2017), vol. 68, pp. 16–35
Contribution: Responsible for hardware designs and benchmarks.

4.1 Introduction

Cyber physical systems enable the physical world to integrate with control systems, such as embedded devices or the Internet of things. Cyber physical systems rely on software algorithms running on embedded devices which use sensors to measure physical processes and actuators to control physical components, such as a valve or the brakes on a car. The software is responsible

for monitoring and controlling these physical components to ensure that they are operating correctly. Examples of cyber physical systems include industrial control systems, process control, autonomous vehicle control systems, and medical implants. The correct functioning of cyber physical systems is crucial, as its failure can lead to injury, damage to equipment, or environmental catastrophe. Therefore, to ensure their correct operation, we need to ensure that the software that runs on the computational components are not compromised.

In this chapter, we present a novel CFI architecture, called SOFIA, which is designed for security-critical scenarios, such as cyber physical systems. It protects software against runtime attacks, such as CRAs, code injection, software tampering, and fault attacks on control flow, while assuming a powerful attacker. It is deeply integrated in a processor's pipeline stages, and relies on cryptographic methods to protect the running software. In particular, SOFIA enforces CFI, Software Integrity (SI), and code secrecy.

The remainder of this chapter is structured as follows. First, the problem statement is provided including the threat model and system goals. Next, the proposed architecture is described. Afterwards, we describe the hardware implementation, followed by toolchain implementation. After that, we provide an evaluation of the security and performance of our solution, and finally discuss future research directions and conclusion.

4.2 Problem Statement

In this section we discuss the system model, composed of a threat model and a set of system requirements.

4.2.1 Threat Model

SOFIA considers an adversary that has the same capabilities as attacker number three in Section 3.4. Here is a brief summary of the assumed capabilities:

- Full control of program and data memory.
- Full control of external I/O pins of the processor.
- Capable of tampering with program flow by means of software-based runtime attacks, as well as through fault attacks on the control flow.
- Side-channel attacks are not possible.

4.2.2 System goals

In this work, a hardware-based security architecture performs run-time verification of the integrity and execution of software. To this end, the requirements of the system are as follows.

Software integrity: The attacker should not be able to execute tampered software on a SOFIA core. We consider the software to consist of instructions and read-only data.

Control flow integrity: The attacker should not be able to change the control flow of running software along an invalid path without this being detected. This includes software-based attacks based on code-reuse such as Jump-Oriented Programming (JOP) [17], Return-Oriented Programming (ROP) [95] and return-to-libc [103], and further includes hardware-based control flow attacks [107], such as instruction skips induced by glitching the clock.

Tampered code protection: No tampered code should be allowed to execute on the processor. We consider tampered code to include illegally modified instructions or any code resulting from an invalid control flow.

Code confidentiality: The attacker cannot read stored software in clear or execute stored software on other devices. This prevents the attacker from finding potential vulnerabilities and further prevents the extraction of confidential IP from the software.

Reverse engineering protection: The software that is stored on each device cannot be analysed or be copied and executed on other devices.

4.3 Architecture

In this chapter, we propose a number of architectural extensions to a microprocessor in order to enhance its security. The extensions consist of two major mechanisms. First, a Control Flow Integrity (CFI) mechanism guards against code injection and code reuse attacks. This mechanism uses a type of Instruction Set Randomization [68] where each instruction is encrypted with control flow dependent information. At runtime, the instructions are decrypted using the same control flow dependent information. Only when the correct control flow paths are taken the instructions will decrypt correctly.

Second, a Software Integrity (SI) mechanism ensures that tampered software never executes on the processor. Here, a Message Authentication Code (MAC) is used to verify the integrity of groups of instructions at run-time. If an integrity

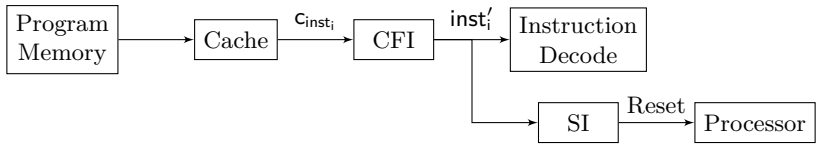

Figure 4.1: Overview of the design using Control Flow Integrity (CFI) and Software Integrity (SI).

Table 4.1: Architectural features vs. system model criteria.

System goal	CFI	SI	CFI and SI
Software Integrity	No	Yes	Yes
Control Flow Integrity	Yes	No	Yes
Tampered code protection	No	Yes	Yes
Code confidentiality	Yes	No	Yes
Tampered control flow prevention	No	No	Yes

violation is detected, the processor is reset in order to prevent any tampered instructions from executing.

An overview of the architecture is shown in Figure 4.1. Encrypted instructions (c_{inst_i}) are fetched from program memory, placed in instruction cache and decrypted by the CFI feature. The decrypted instructions ($inst'_i$) are sent to the Instruction Decode stage of the processor. At the same time, the SI feature performs run-time integrity verification of the decrypted instructions. Upon detection of an integrity violation, execution is halted by resetting the processor, thereby preventing both tampered control flow and tampered instructions from executing. The processor should be able to reboot reliably fast, allowing the software to quickly reach a safe and expected state. Each processor is embedded with a set of unique keys that can only by accessed by the block cipher. The keys are known only by the device manufacturer and the software provider.

In the remainder of this section, the CFI and SI mechanisms will first be presented as standalone features. Afterwards, there will be a discussion on how to use the two mechanisms within a single system. As shown in Table 4.1, each standalone feature meets only part of the criteria of the system model. However, when the standalone features are combined into a single system (CFI and SI), they complement each other, thereby satisfying all the criteria in the system model.

ALGORITHM 1: Control flow dependent information is used to encrypt and decrypt the instructions of a program.

Input: Plaintext m_i, j-bit key k_1, number of plaintext blocks u, nonce ω
Result: Encryption produces r-bit ciphertext blocks c_0 , $\cdots$, c_u. Decryption recovers plaintext m.

Encryption: ;
for $i \leftarrow 1$ *to* u **do**
    $I_i = \{\omega \parallel \mathsf{prevPC}_i \parallel \mathsf{PC}_i\}$;
    $O_i \leftarrow E_{\mathsf{k}_1}(I_i)$;
    $t_i \leftarrow$ the r least-significant bits of O_i ;
    $\mathsf{c}_i \leftarrow \mathsf{m}_i \oplus t_i$;

Decryption: ;
for $i \leftarrow 1$ *to* u **do**
    $\mathsf{m}_i \leftarrow \mathsf{c}_i \oplus E_{\mathsf{k}_1}(I_i)$, where I_i, O_i, and t_i are computed as above.

4.3.1 Control Flow Integrity (CFI)

The main idea of the CFI mechanism is to perform instruction-set randomization by decrypting instruction opcodes based on control flow dependent information. This enforces CFI by ensuring that instructions are only decrypted correctly at runtime when valid control flow paths are followed. A binary that consists of encrypted instructions is created by performing a transformation operation at compile time. The instructions are encrypted based on the control flow paths present in a precise Control Flow Graph (CFG) of the whole program. Each encrypted instruction is decrypted at run-time using a combination of the current program counter and the address of the previously executed instruction.

Each instruction in the binary is encrypted using a block cipher in counter-mode, as shown in Algorithm 1. The *counter* value (I_i) contains the dynamic control flow between two instructions. This is expressed as the address of the currently executing instruction together with the address of the previously executed instruction. Encryption is performed with $\mathsf{c}_{\mathsf{inst_i}} = E_{\mathsf{k}_1}(I_i) \oplus \mathsf{inst_i}$, while decryption is performed with $\mathsf{inst}'_\mathsf{i} = E_{\mathsf{k}_1}(I_i) \oplus \mathsf{c}_{\mathsf{inst_i}}$, with I_i the counter value and k_1 the encryption key. The counter is $I_i = \{\omega \parallel \mathsf{prevPC}_i \parallel \mathsf{PC}_i\}$, with PC the current program counter or address of $\mathsf{inst_i}$, prevPC the previously executed program counter, and ω a nonce. The nonce ω needs to be unique across each version of every encrypted program, and is stored in a fixed address in the binary. This decryption process is illustrated in Figure 4.2.

Instructions are decrypted correctly as long as the control flow of a running program follows the paths of the original CFG. However, during a control flow hijacking attack, an attacker forces control to flow along a path which does

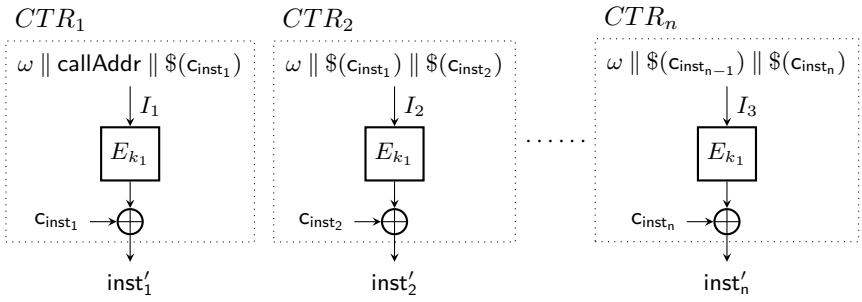

Figure 4.2: Encrypted instructions ($\mathsf{c_{inst_n}}$) are decrypted at runtime using dynamic control flow information consisting of the current and previous program counters (PC, and prevPC). Under the condition that control flow is untampered, $\mathsf{PC} = \text{addr}(\mathsf{c_{inst_i}})$, and $\mathsf{prevPC} = \text{addr}(\mathsf{c_{inst_{i-1}}})$, or $\mathsf{prevPC} = \mathsf{callAddr}$, with callAddr the call site.

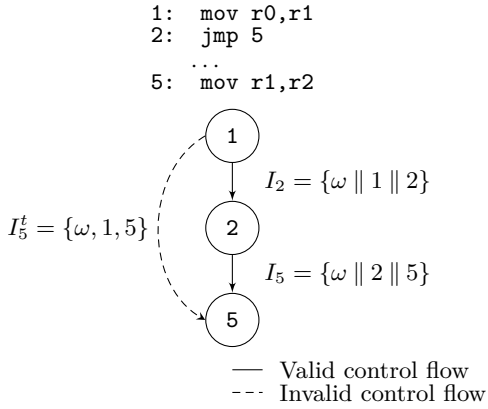

Figure 4.3: A CFG of a small program shows two different control flow paths from node 1 to node 5. If the valid control flow path is taken, all instructions are decrypted correctly. However, when the invalid control flow path is taken, instruction 5 is decrypted incorrectly.

not exist in the original CFG (see Section 3.2). This will cause at least one instruction to be decrypted incorrectly, as the counter I_i contains an invalid prevPC.

An example program listing with corresponding CFG is shown in Figure 4.3.

Each CFG node represents a single encrypted instruction, while the edges indicate control flow between instructions. The solid edges represent valid control flow, with the encryption counter I_i indicated next to each edge. The CFG shows that control flows from node 1 to 2; therefore, instruction 2 is decrypted with counter value $I_2 = \{\omega \parallel 1 \parallel 2\}$. A branch causes control to flow from node 2 to 5; therefore, instruction 5 is decrypted with counter value $I_5 = \{\omega \parallel 2 \parallel 5\}$. When an attacker causes invalid control flow to occur from, e.g., node 1 to node 5, instruction 5 is decrypted with counter $I_5^t = \{\omega \parallel 1 \parallel 5\}$, leading to a decryption error.

Function calls are supported in a similar way as direct branches. The function's entry point is encrypted with the call site, while the return point in the call site is encrypted with the address of the return instruction in the callee. Callees with multiple call sites or the targets of function pointers with multiple call sites correspond to nodes with multiple predecessors in the CFG, and cannot be handled with the scheme discussed so far. Section 4.3.4 discusses the necessary extensions.

The CFI mechanism presented in this section provides protection from attacks based on code injection and code reuse. However, a decryption error caused by tampered control flow might lead to a decrypted instruction ($\mathsf{inst}'_\mathsf{i}$) with a valid opcode. The instruction will execute on the processor, albeit leading to a different result than that of the original program. This is a serious problem, as the incorrectly decrypted instruction could lead to a malicious result. This problem can be solved by using the CFI mechanism in combination with the SI mechanism described in the following section.

4.3.2 Software Integrity (SI)

This section presents a mechanism which ensures, with very high probability, that only untampered instructions can execute on the processor. A Message Authentication Code (MAC) is *precomputed* on groups of instructions, and stored in instruction memory, as shown in Figure 4.4. At *run-time*, a MAC verification is performed on each group of instructions before they reach the end of the processor's pipeline. The run-time MAC is compared with the precomputed MAC to verify the integrity of all instructions in each group. If their verification fails, the processor is reset in order to prevent tampered instructions from executing.

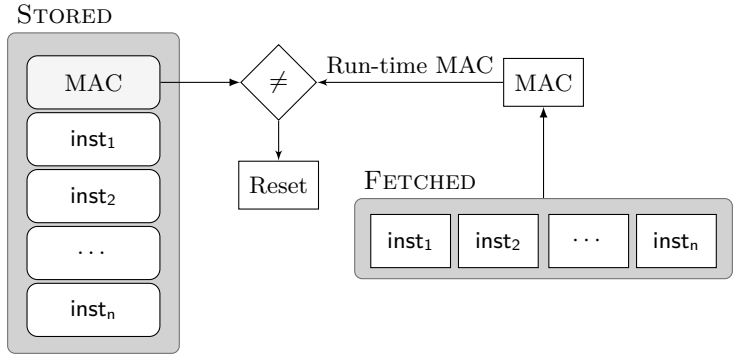

Figure 4.4: The integrity of a program's instructions is verified at runtime by comparing the precomputed MAC with the run-time calculated MAC. If verification fails, the processor is reset to prevent tampered instructions from executing.

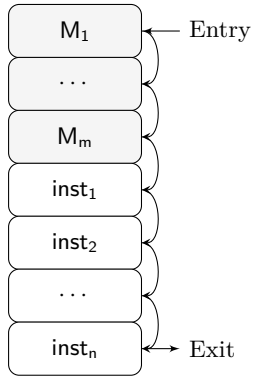

Figure 4.5: The *execution block* consists of an m-word precomputed MAC (M) and n instructions. Control flow can only enter at M, and can only exit at $\mathsf{inst_n}$. Inside a block the control flows through each consecutive word.

Design

An *execution block*, shown in Figure 4.5, consists of m MAC words (M_i) and n instructions ($\mathsf{inst_i}$). Control can only flow into an execution block at $\mathsf{M_1}$, and can only exit at $\mathsf{inst_n}$. Inside the execution block, control flow passes through each MAC word and then through each instruction.

The processor's Instruction Fetch (IF) pipeline stage is used to read instructions

and precomputed MAC words from memory. The MAC words are replaced with a `nop` before being sent to the decode stage. It is necessary that all words in an execution block are fetched every time it is executed, as all the instructions in a block are needed to compute the run-time MAC, and the precomputed MAC is required for verification.

In our design we use the Cipher Block Chaining-Message Authentication Code (CBC-MAC) algorithm [64] with a 64-bit MAC length. In the remainder of the text we will refer to the two 32-bit chunks of the MAC as M_1 and M_2. It is well known that the CBC-MAC algorithm is only secure for messages of a fixed length [59]. Care needs to be taken, as SOFIA computes a MAC on different message lengths due to the two block types that each consists of a different number of instructions (see Section 4.3.5). We propose to address this issue by using a different key for each type of block, thereby using one key for each message length. We further use a different key for the MAC and for encryption. Consequently, each device has a total of three different keys: k_1 is used for encryption, k_2 is used for CBC-MAC of execution blocks, and k_3 is used for CBC-MAC of multiplexer blocks.

Preventing tampered blocks from executing

One of the design criteria is to prevent the execution of instructions that are tampered with or occur after an illegal control flow. Here we discuss the techniques used to achieve this.

SOFIA is designed to work as an extension to any microprocessor. However, for the discussion in this chapter, we will base the design on the instruction pipeline of the SPARCv8-based [97] LEON3 [51] processor as an example. It uses a single-issue pipeline with seven stages:

1. **Instruction Fetch (IF)** requests instructions from cache or main memory.
2. **Instruction Decode (ID)** translates opcodes into instructions and generates call or branch target addresses.
3. **Operand Fetch (OF)** reads the operands from registers.
4. **Execute (EXE)** performs arithmetic logical unit, logical, and shift operations. For memory operations, the address is generated.
5. **Memory Access (MA)** performs read or write operations to/from memory.
6. **Exception (XCP)** resolves traps and interrupts.
7. **Write Back (WB)** stores the result of the arithmetic logical unit, logical, shift, or memory access operations to the register file.

A cyber physical systems's physical components commonly interface with

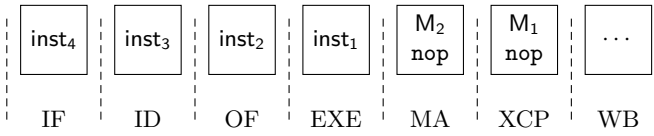

Figure 4.6: The instructions in a four instruction execution block fit in the pipeline stages before the Memory Access (MA) stage. This allows the architecture to verify the integrity of the block before a memory access has been performed.

a microcontroller through a physical connection, such as General Purpose Input/Output pins. The microcontroller's software controls the actuators by writing to the interface's memory mapped addresses with *store instructions*. Special care needs to be taken with store instructions, as they could used to send tampered commands to an actuator, which could have a catastrophic effect, e.g., disable the brakes on a car, or by emptying a patient's insulin tank.

SOFIA detects tampered instructions by verifying an execution block's integrity while the instructions are partially executed inside the processor's instruction pipeline. In this work we propose to verify the integrity of a block before any store instructions inside a given block has reached the Memory Access (MA) pipeline stage. Instructions other than stores cannot control actuators from cyber physical systems, and are therefore allowed to progress through all the procesor's pipeline stages. A simple approach, illustrated in Figure 4.6, is to make the execution blocks small enough to fit into the pipeline stages before the MA stage. This allows the run-time MAC to be computed before the instructions reach the MA stage. If verification fails, the instructions are prevented from moving further in the pipeline by resetting the processor, thereby discarding all instructions in the block before reaching the MA stage.

In the LEON3, *memory access* operations are performed in the fifth instruction pipeline stage. Therefore, a four-instruction execution block can fit in the pipeline stages before the MA stage. When a single-cycle MAC hardware component is used, tampering can be detected before the first instruction reaches the MA stage. To improve the performance of the system, the number of instructions in an execution block can be increased to six instructions if store instructions are not allowed to be located on inst_1 or inst_2, as illustrated in Figure 4.7. A *store violation* is generated when a store instruction is detected on inst_1 or inst_2. Store violations are handled in the same manner as MAC violations.

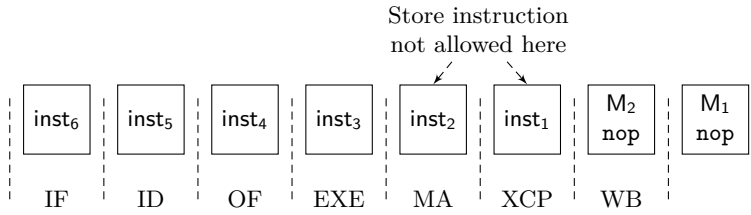

Figure 4.7: The size of an execution block can be increased to six instructions if store instructions are restricted from inst_1 and inst_2.

4.3.3 Control Flow Integrity with Software Integrity (CFI and SI)

By using both the CFI and SI mechanisms in a single processor, it is possible to detect tampered execution blocks as well as invalid control flow. By designing the system to rapidly detect tampering we are able to also prevent the execution of instructions resulting from tampered control flow. The CFI mechanism decrypts instructions based on the run-time control flow, but can not detect decryption errors. The SI mechanism performs integrity verification in order to detect tampered instructions, but cannot detect invalid control flow when used alone. Therefore, to detect tampered control flow and instructions, the SI mechanism verifies the integrity of a block only after the CFI mechanism has decrypted the encrypted instructions. Figure 4.8 shows the process of decrypting the words in an execution block using counter-mode, and then calculating the CBC-MAC on all the decrypted instructions (inst'_i). The figure assumes untampered control flow, i.e., each counter value I_i contains the valid values for PC_i and prevPC_i, such that $\mathsf{PC}_i = \text{addr}(\mathsf{c}_{\mathsf{inst}_i})$, and $\mathsf{prevPC}_i = \text{addr}(\mathsf{c}_{\mathsf{inst}_{i-1}})$ or $\mathsf{prevPC}_i = \mathsf{callAddr}$, with $\mathsf{callAddr}$ the address of the call site.

At run-time the CFI mechanism first decrypts the instructions using dynamic control flow information. Next, the SI mechanism calculates the run-time MAC on the decrypted instructions. If an invalid control flow path was taken, a decryption error occurs. When the SI mechanism calculates the run-time MAC with the incorrectly decrypted instruction and incorrect MAC is produced and the integrity verification fails. The processor is then reset to prevent the execution of instructions resulting from the tampered control flow.

The plaintext binary is transformed with the MAC-then-Encrypt construction [85]. For each execution block, a MAC M is first calculated on the plaintext instructions. Next, M is interleaved with the instructions to form execution blocks. Finally, the plaintext execution blocks are encrypted with Algorithm 1, and then stored in main memory.

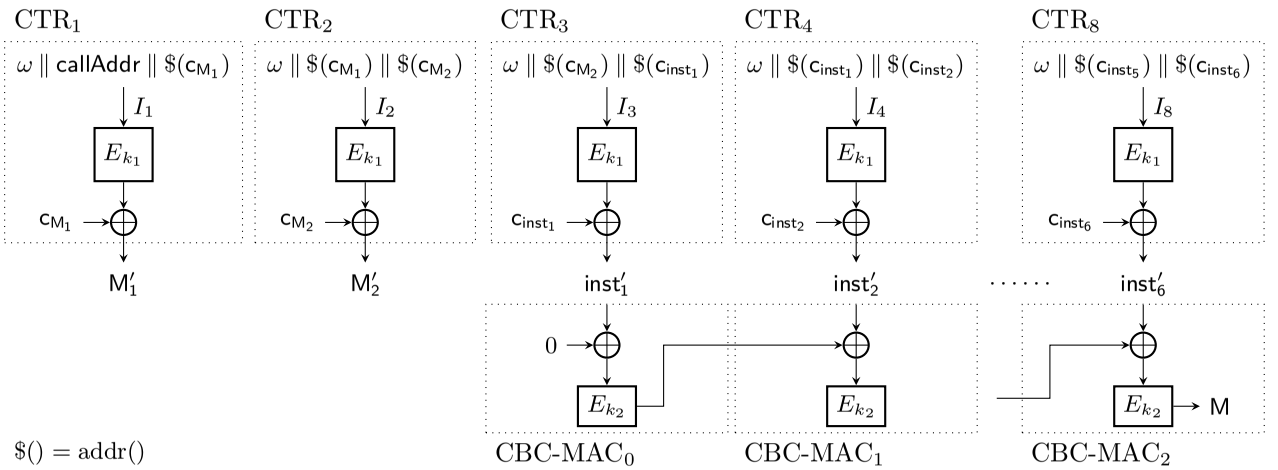

Figure 4.8: The CFI and SI architectural features are combined to detect tampered software and control flow. At runtime, the encrypted words in an execution block (c_{inst_i} and C_{M_i}) are first decrypted with counter-mode, and then a CBC-MAC is used to compute a MAC on the decrypted instructions ($inst'_i$).

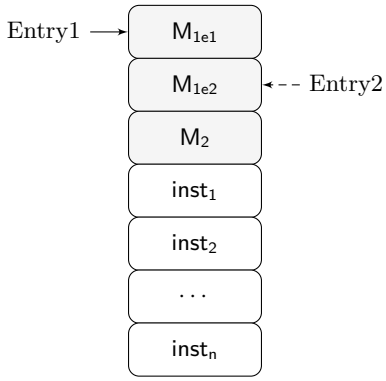

Figure 4.9: The plaintext multiplexer block uses two copies of the first MAC word M_1 as its two entry points, which are respectively called M_{1e1} and M_{1e2}.

4.3.4 Blocks with Multiple Predecessors

The CFI mechanism presented in Section 4.3.1 only supports nodes with a single predecessor, since execution blocks only have one entry point. This section introduces the *multiplexer block* which allows for two predecessors. This block uses both the CFI (Section 4.3.1) and SI (Section 4.3.2) mechanisms.

Just like for the execution block, a two-word MAC M is first calculated on the block's plaintext instructions $inst_i$. To support two predecessors, we propose to create two entry points by inserting two copies of the first MAC word M_1 at the beginning of the block, as shown in Figure 4.9. Each copy of M_1 is used as an entry point into the block, which we call M_{1e1} and M_{1e2}. Each of the two entry points are therefore encrypted using their respective call sites ($callAddr_1$ and $callAddr_2$), as illustrated by Figure 4.10. The two entry points are encrypted as follows: $c_{M_{1e1}} = E_{k_1}(I_1) \oplus M_1, I_1 = \{\omega \,\|\, callAddr_1 \,\|\, \mathrm{addr}(c_{M_{1e1}})\}$, and $c_{M_{1e2}} = E_{k_1}(I_2) \oplus M_1, I_2 = \{\omega \,\|\, callAddr_2 \,\|\, \mathrm{addr}(c_{M_{1e2}})\}$. In addition, two distinct control flow paths exist in the block. The first control flow path enters the multiplexer block at $c_{M_{1e1}}$, skips $c_{M_{1e2}}$, and flows to c_{M_2}, followed by all the encrypted instructions c_{inst_i} in the block. The second control flow path enters the block at $c_{M_{1e2}}$, flows to c_{M_2}, followed by all the encrypted instructions c_{inst_i} in the block. To decrypt c_{M_2}, the counter value $I_3 = \{\omega \,\|\, \mathrm{addr}(c_{M_{1e2}}) \,\|\, PC\}$ is used by the hardware, regardless of which control flow path is used.

If a node in the CFG requires more than two predecessors a tree of multiplexer blocks can be used. In Figure 4.11 a multiplexer tree allows a node to be called by four different call sites. The tree structure is used to handle entry points from call sites, function pointers, and branch targets. Therefore, the multiplexer

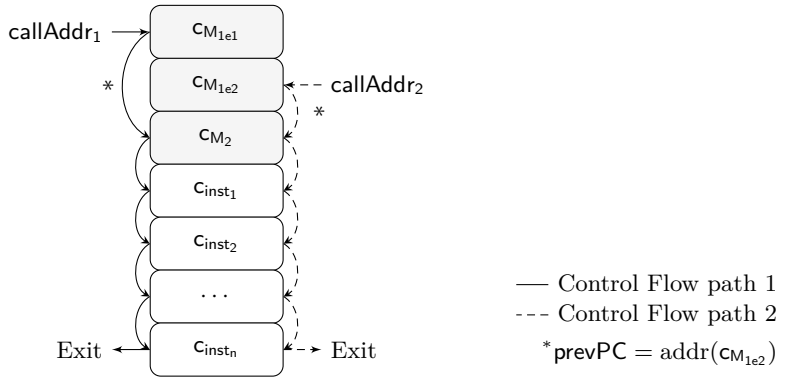

Figure 4.10: The encrypted multiplexer block supports two entry points and has two unique control flow paths through the block.

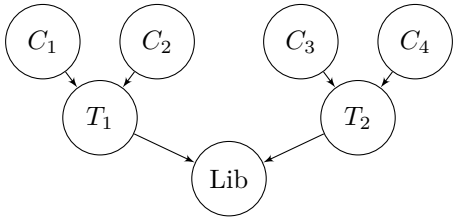

Figure 4.11: A tree of multiplexer nodes is used to increase the number of call sites (C_i) that can invoke a function.

tree structure needs to have an entry point for each call site that can reach a function through a branch or a function call.

4.3.5 Support for blocks with single and multiple predecessors

Most non-trivial programs consist of blocks with one entry point as well as blocks with multiple entry points. In the above text we outlined two different types of blocks, namely the execution block with a single entry point, and the multiplexer block which has two entry points. In order to create a meaningful program using these two blocks, we need to develop mechanisms to make them work together within the same system.

The software needs a mechanism to indicate to the hardware which type of block is about to execute. We propose to solve this by using the call site to inform the hardware of the block type. For an execution block, we select the block's first

word $\mathsf{c}_{\mathsf{M}_1}$ as the call site. Therefore all calls, branches, or fall-throughs to $\mathsf{c}_{\mathsf{M}_1}$ will indicate to the hardware that an execution block should be executed. For a multiplexer block we propose to use the second and third words, respectively $\mathsf{c}_{\mathsf{M}_{1e2}}$ and $\mathsf{c}_{\mathsf{M}_2}$, as the two call sites. Therefore, a branch or a call to $\mathsf{c}_{\mathsf{M}_{1e2}}$ or $\mathsf{c}_{\mathsf{M}_2}$ will indicate to the hardware that a multiplexer block should be executed. A branch/call to $\mathsf{c}_{\mathsf{M}_{1e2}}$ will cause the first control flow path to be followed, and similarly, a branch/call to $\mathsf{c}_{\mathsf{M}_2}$ will cause the second control flow path to be followed.

The size of both block types is chosen to be eight 32-bit words. Therefore, the execution block consists of 2 MAC words and 6 instructions, while a multiplexer block consists of 3 MAC words and 5 instructions.

4.3.6 MAC Chaining

This section discusses the security of decrypting parts of a MAC with different counter-mode operations, and proposes some changes to improve its security. The discussion here is limited to execution blocks, but the proposed modifications are also applicable to multiplexer blocks.

When using the architecture presented so far, the first encrypted MAC word ($\mathsf{c}_{\mathsf{M}_1}$) is decrypted with a counter value (I_1) that depends on the call site ($\mathsf{callAddr}$), while the second MAC word is decrypted with a counter value (I_2) that does not depend on the call site. Ideally all MAC words should be decrypted using a counter value that depends on $\mathsf{callAddr}$. However, since the current architecture only decrypts the first MAC word using a counter value that depends on $\mathsf{callAddr}$, the security of the system is reduced to 32 bits.

To solve this problem, we propose to use additional cryptographic operations to chain all the MAC words together, thereby ensuring that the decryption of the entire MAC relies on $\mathsf{callAddr}$. We propose two different solutions. For $\text{size}(\mathsf{M}) = b$, where b is the block size of the cipher, a single block cipher operation can be used to perform the chaining, and an ECB-mode encryption can be used. However, for $\text{size}(\mathsf{M}) > b$, multiple cryptographic operations are necessary, for which we recommend using CBC-mode encryption to chain the MAC words together. For both cases encrypting the binary works as follows. First, M is encrypted with either ECB or CBC mode, providing us with $\mathsf{c}_{\mathsf{M}_1}$, and $\mathsf{c}_{\mathsf{M}_2}$. Next, $\mathsf{c}_{\mathsf{M}_1}$ and $\mathsf{c}_{\mathsf{M}_2}$ are encrypted with counter-mode, i.e. $\mathsf{c}_{\mathsf{c}_{\mathsf{M}_1}} = E(I_1) \oplus \mathsf{c}_{\mathsf{M}_1}$, with $I_1 = \{\omega \,\|\, \mathsf{callAddr} \,\|\, \text{addr}(\mathsf{c}_{\mathsf{c}_{\mathsf{M}_1}})\}$, and $\mathsf{c}_{\mathsf{c}_{\mathsf{M}_2}} = E(I_2) \oplus \mathsf{c}_{\mathsf{M}_2}$, with $I_2 = \{\omega \,\|\, \text{addr}(\mathsf{c}_{\mathsf{c}_{\mathsf{M}_1}}) \,\|\, \text{addr}(\mathsf{c}_{\mathsf{c}_{\mathsf{M}_2}})\}$. At runtime, the reverse operations are performed. First, counter-mode decryption will be used: $\mathsf{c}'_{\mathsf{M}_1} = \mathsf{c}_{\mathsf{c}_{\mathsf{M}_1}} \oplus I_1$, and $\mathsf{c}'_{\mathsf{M}_2} = \mathsf{c}_{\mathsf{c}_{\mathsf{M}_2}} \oplus I_2$. Next, the partially decrypted MAC words ($\mathsf{c}'_{\mathsf{M}_1}$ and $\mathsf{c}'_{\mathsf{M}_1}$) will be de-chained using either ECB-mode or CBC-mode decryption. This approach

ensures that M'_2 can only be decrypted correctly if M'_1 was also decrypted correctly. In order for $\mathsf{M}_1 = \mathsf{M}'_1$, the correct call site (callAddr) is needed. This solves our problem, as the decryption of the entire MAC now relies on callAddr.

Chaining with ECB-mode: The plaintext MAC words are encrypted using ECB mode, i.e., $\{\mathsf{c}_{\mathsf{M}_1} \parallel \mathsf{c}_{\mathsf{M}_2}\} = E(\mathsf{M}_1 \parallel \mathsf{M}_2)$. The partially decrypted MAC words are dechained as follows: $\{\mathsf{M}'_1 \parallel \mathsf{M}'_2\} = D(\mathsf{c}'_{\mathsf{M}_1} \parallel \mathsf{c}'_{\mathsf{M}_2})$.

Chaining with CBC-mode: The plaintext MAC words are encrypted using CBC mode, i.e., $\mathsf{c}_{\mathsf{M}_1} = E(\mathsf{M}_1 \oplus IV)$, and $\mathsf{c}_{\mathsf{M}_2} = E(\mathsf{M}_2 \oplus \mathsf{c}_{\mathsf{M}_1})$, where the IV is an initialization value. The partially decrypted MAC words are dechained as follows: $\mathsf{M}'_1 = D(\mathsf{c}'_{\mathsf{M}_1}) \oplus IV$, and $\mathsf{M}'_2 = D_{\mathsf{k}_1}(\mathsf{c}'_{\mathsf{M}_2}) \oplus \mathsf{M}'_1$.

4.4 Hardware implementation

This section describes the SOFIA hardware implementation. First, we provide an overview of the modifications made to the LEON3 processor. Next, we discuss the choice of block cipher, followed by a description of the hardware design. Finally, we discuss the schedule employed by the block cipher.

4.4.1 Overview

SOFIA has been implemented on Gaisler's LEON3 v1.3.7-b4144 soft microprocessor. The processor was configured with the minimum number of peripherals, branch prediction support, double data rate (DDR) memory support, a small amount of cache, and a single vector trap table. The hardware design was evaluated on a Xilinx Virtex-6 XC6VLX240T FPGA.

The majority of the modifications to the processor were done in the seven-stage integer pipeline (`iu3.vhd`). The CFI component was integrated with the LEON3's cache controller (`ico`) and the ID pipeline stage. In addition, the logic to calculate the next program counter was modified in order to allow for the non-standard control flow through multiplexer blocks. A reset line was wired from `iu3.vhd` to the top level design (`leon3mp.vhd`) in order to halt execution of instructions when either an integrity violation is detected or a store instruction is detected on $\mathsf{inst}_1, \ldots, \mathsf{inst}_3$. No additional instruction pipeline stages were added to the processor.

Table 4.2: Hardware overhead of two block ciphers: RECTANGLE and PRINCE.

Design	Cycles	Slices	LUTs	Flip Flops	Clock (ns)
RECTANGLE	1	699	2,013	0	22.77
PRINCE	1	348	1,148	0	15.08
RECTANGLE	2	839	2,016	64	11.4
PRINCE	2	376	1,154	64	8

4.4.2 Block cipher

In order to prevent tampered instructions from executing, the MAC computation needs to complete in only a few cycles. A simple approach is to use a single cycle block cipher, thereby allowing the decryption of each instruction in only one cycle. This further allows for rapid MAC verification when using CBC-MAC, as each decrypted instruction ($\mathsf{inst}'_{\mathsf{i}}$) can be processed in a single cycle.

For our initial design we selected the RECTANGLE-80 [114] block cipher, which has a 64-bit block size, and an 80-bit key. The RECTANGLE-80 block cipher was unrolled to compute in one cycle [79]. Our initial experiments showed that placing a single-cycle implementation of RECTANGLE-80 in the instruction pipeline stages of the processor increased the design's critical path to a maximum clock frequency of around 28 MHz. Block ciphers are typically complex, leading to long critical paths when all of its operations are performed in a single cycle. Therefore, in order to improve the critical path of the SOFIA core, we decided to use a dual cycle cipher. In addition, we made a second implementation which uses a dual cycle version of PRINCE [18], which has a 64-bit block size and a 128-bit key. Table 4.2 compares the hardware overhead of each block cipher configured for running in either one or two cycles.

4.4.3 Hardware design

A block diagram of the hardware is shown in Figure 4.12. The two stages of the dual cycle block cipher are indicated by "Cipher part one" and "Cipher part two", with the pipeline register indicated with `r_pipeline`. As discussed in Section 4.3, the block cipher is used in three different modes of operation. First, to perform the counter-mode decryption the cipher uses the input $\{\omega \parallel NPC \parallel FPC\}$, where `NPC` indicates the address that will be fetched in the next IF slot, and `FPC` is the address in the current IF slot. The first execution block serves as the starting point of the program, and can be entered from anywhere. To this end, $I_1 = \{\omega \parallel FPC \parallel 0\}$ is used to decrypt the first

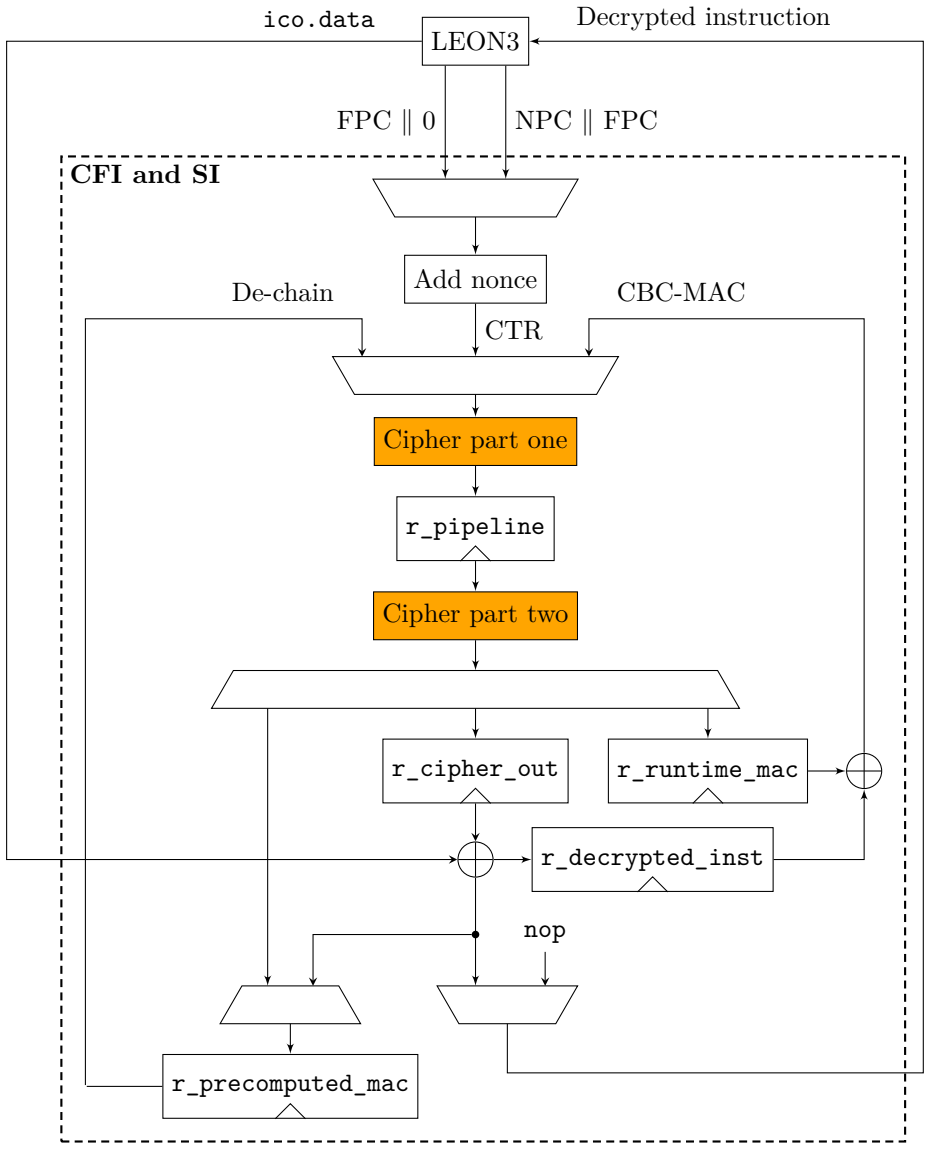

Figure 4.12: A hardware block diagram showing the SOFIA core integrated in the instruction pipeline stages of the LEON3.

MAC pair in the first execution block. For each counter-mode decryption, the output of the cipher is stored in the `r_cipher_out` register. In the following cycle, encrypted instructions arriving from the cache controller (`ico.data`) are decrypted, stored in the `r_decrypted_inst` register, and sent to the processor's decode stage. In the same cycle, encrypted MACs are decrypted, stored in the `r_precomputed_mac` register, and a `nop` is sent to the decode stage.

Second, for the MAC dechaining operation, the counter-mode decrypted MAC (C'_{M}), which was stored in `r_precomputed_mac` in the previous slot, is provided as an input to the cipher. The result of this operation (M') is finally stored inside the `r_precomputed_mac` register.

Third, for the CBC-MAC operation, the first cipher input consists of the decrypted instruction stored inside `r_decrypted_inst`, and the result is stored inside `r_runtime_mac`. For the second and last CBC-MAC operation, `r_decrypted_inst` is XORed with `r_runtime_mac` and are provided as an input to the cipher. When the final MAC result has been computed it is immediately compared with `r_precomputed_mac`.

The reset logic is generated from two different sources. First, the output of the cipher is compared to the value stored in `r_precomputed_mac` during the first cycle when the lowest six bits of `FPC` are equal to eight, which corresponds to the first cycle in which the MAC computation is finished. Second the first three decrypted instructions ($\mathsf{inst}_1, \ldots, \mathsf{inst}_3$) are decoded to detect STORE instructions. When the reset line is asserted the processor immediately resets, causing the annulation of all partially executed instructions, as well as the cancellation of all pending memory accesses.

While experimenting with the design, we found a large delay present in the FPC and NPC signals generated by the LEON3 together with our own logic which drives the input signals to the first block cipher stage. To allow the design to run at a high clock frequency, `r_pipeline` was not placed in the middle of the block cipher, but was rather placed after only a couple of rounds of computation. This leads to the critical path of the design being from `r_pipeline` to `r_precomputed_mac`. We further improved the timing of the design by partitioning the block cipher to a specific region of the FPGA.

4.4.4 Scheduling the Block Cipher

In our implementation, we use a single block cipher instance to perform the following three different operations. First, counter-mode decryption is performed on the encrypted instructions ($\mathsf{c}_{\mathsf{inst}_i}$) and encrypted MAC ($\mathsf{C}_{\mathsf{CM}}$). Second, an ECB decryption operation is performed on C'_{M} to de-chain the MAC, and finally

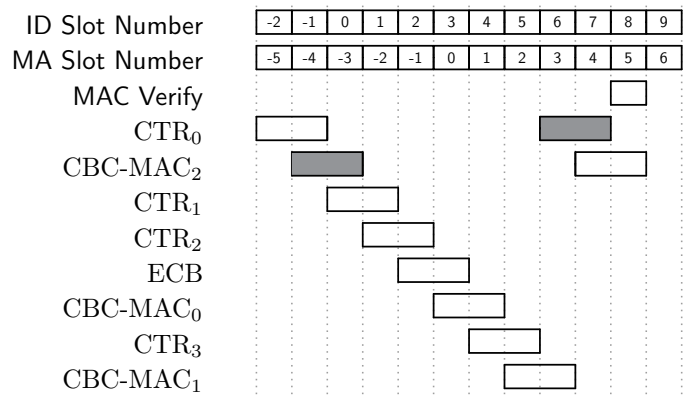

Figure 4.13: A timing diagram of the block cipher operations to process a single execution block. $\mathrm{CTR_n}$ indicates counter-mode decryption, ECB indicates MAC de-chaining, and CBC-MAC indicates part of the CBC-MAC computation. The execution block exists in slots zero to seven. Negative slot numbers indicate the previous block in the instruction pipeline. Gray blocks indicate cipher operations of the previous or next block.

obtain M'. Third, CBC-MAC is used to calculate the runtime MAC on the decrypted instructions inst'_i. Since our design's block cipher has 64-bit blocks, a single operation can process two 32-bit words. This means that a total of four counter-mode operations are required to decrypt all the words in a block, one operation is required for MAC de-chaining, and three CBC operations are required to calculate the MAC.

A timing diagram of the block cipher operations to process a single execution block is shown in Figure 4.13. Two different instruction pipeline stages of the LEON3 are indicated in the figure, namely the Instruction Decode (`ID`) slot number, and the Memory Access (`MA`) slot number. In this example, the execution block starts at slot number zero, and ends at slot number seven. The figure shows that *MAC verification* occurs at MA slot number five. Since a dual cycle block cipher is used, all crypto operations span two slots. In addition, the block cipher is pipelined and a new input can be fed to the cipher in every slot.

The four counter-mode operations are indicated with $CTR_0, \ldots, CTR_3$. The result of each counter-mode operation is used to decrypt an instruction in the ID stage. When scheduling the counter-mode operations, it is important to ensure that all instructions can be decrypted before reaching the ID stage, e.g., the computation of $E(I_0)$ needs to finish before slot zero reaches the ID stage.

The MAC de-chaining operation is indicated by ECB, and finishes in ID slot 2.

The CBC-MAC operations, indicated by CBC-MAC$_0$, ..., CBC-MAC$_2$, are used to calculate the MAC over the decrypted instructions. The final MAC operation (CBC-MAC$_2$) starts in ID slot 7, and finishes in slot 8, at which point the MAC verification is performed. To meet the requirement that tampered instructions doesn't execute, STORE instructions are disallowed in $\mathsf{inst}_1, \ldots, \mathsf{inst}_3$.

4.4.5 Limitations

This section discusses the current limitations of our SOFIA implementation.

SOFIA currently does not support interrupts. We don't believe that this is a fundamental limitation, and could be supported with additional hardware logic. One of the challenges we foresee is to ensure that memory blocks are decrypted correctly, both when servicing an interrupt as well as when returning from an interrupt. It's worth mentioning that the problem of providing interrupt-support for SOFIA is entirely different to the problem addressed in Chapter 2, in which the goal was to maintain the confidentiality of a protected module's data while enabling interrupt support.

One approach to provide interrupt support is to save the internal SOFIA registers on a protected stack before entering an Interrupt Service Routine (ISR), and then restoring the internal SOFIA registers when returning from an ISR. In addition, the hardware should allow the first block of each ISR to be entered from any predecessor. The simplest approach is to delay executing an ISR until after processing the last instruction in the currently executing block. This has the advantage that only a few internal SOFIA registers have to be saved/restored on the protected stack. However, the disadvantage is that the time required to service an interrupt will be increased.

4.5 Software Implementation

The SOFIA hardware extension imposes several constraints on software which should run on the modified processor. In particular, there are constraints regarding the control flow between blocks as well as the type and position of instructions within blocks. To be able to compile and run standard C code, we have designed a software toolchain consisting of several parts, pre-linkage as well as post-linkage. Because the complete control flow graph must be known to produce SOFIA-compatible machine code, all source files need to be passed to the toolchain for compilation. Our toolchain is able to compile C down to an

ELF binary satisfying all constraints with just the limitations given by SOFIA itself.

4.5.1 Toolchain Design

Our toolchain consists of several independent tools. We added optimization passes and changed the SPARC backend within the LLVM compiler infrastructure. An unmodified version of `clang` is used to compile source code to LLVM intermediate code. The intermediate representations for all source files are then linked together into a single file using `llvm-link` which is then passed to `opt`. The optimizer applies two custom optimization passes to the program to ensure a binary control flow graph within each function, i.e., at most two predecessors are allowed per node, and a binary call graph between functions, i.e., each function is only allowed to have two direct predecessors, before passing the result to `llc`. For `llc` we changed the SPARC backend in such a way that it emits SPARC assembler instructions respecting the constraints of SOFIA blocks instead of plain assembler instructions.

Within the second stage, the binutils provided by the Bare C Compiler from Aeroflex Gaisler are used to assemble and link the code emitted by `llc`. Furthermore, unnecessary sections such as comment or debug sections are stripped from the resulting binary using `objcopy`.

For the final stage, the binary is processed by custom standalone tools written in C++. These tools encrypt the SOFIA blocks within the binary and assist the programmer by verifying that the final binaries comply with the SOFIA constraints. An overview of how the independent parts of the toolchain work together is shown in Figure 4.14. Each step will be described further in the following sections.

Compiler Stage

The LLVM Compiler Infrastructure is used to transform C to assembly code as shown in Figure 4.15. The tools used to achieve this are `clang`, `llvm-link`, `opt`, and `llc`. Clang is the compiler frontend provided by the LLVM project. It takes C source code as input and emits LLVM intermediate code. No changes were necessary for the purpose of this project.

The intermediate code generated by `clang` is then linked using `llvm-link`. This step is needed, because building a binary call graph between functions requires global knowledge of the program. To ensure that each function is only called by two other functions, each function needs access to its callers.

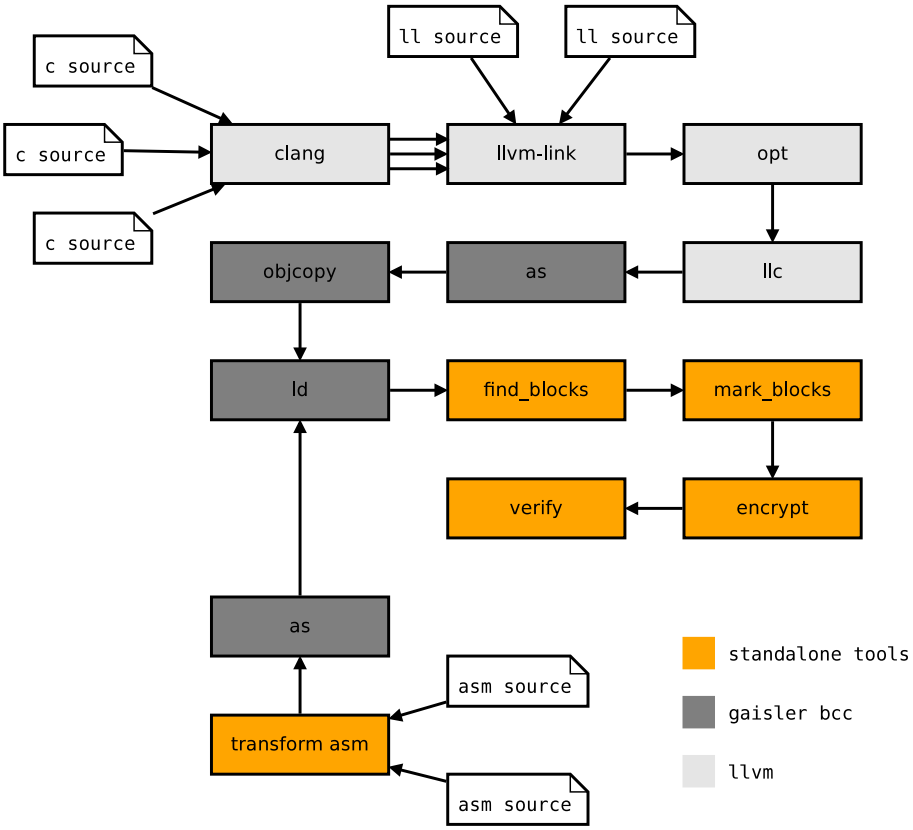

Figure 4.14: Overview of how the independent parts of the SOFIA toolchain work together.

Two optimization passes were written to ensure a binary control flow graph, i.e., each node in the graph is only allowed to have at most two predecessors. Each pass implemented the same algorithm (see Section 4.5.2) but operates at different scopes, namely intra- and inter-function. At this point, a single LLVM intermediate code file is produced. The last step of the compiler stage uses `llc` to compile the intermediate code to assembly code.

As SOFIA is currently not able to handle traps; since register window overflows and underflows would cause traps, the SPARC register window was disabled. To this end, we implemented the *flat* calling convention as the first backend patch for LLVM, which has the same effect as the `-mflat` option passed to older GCC versions. This leads to an increase of approximately 10% in both

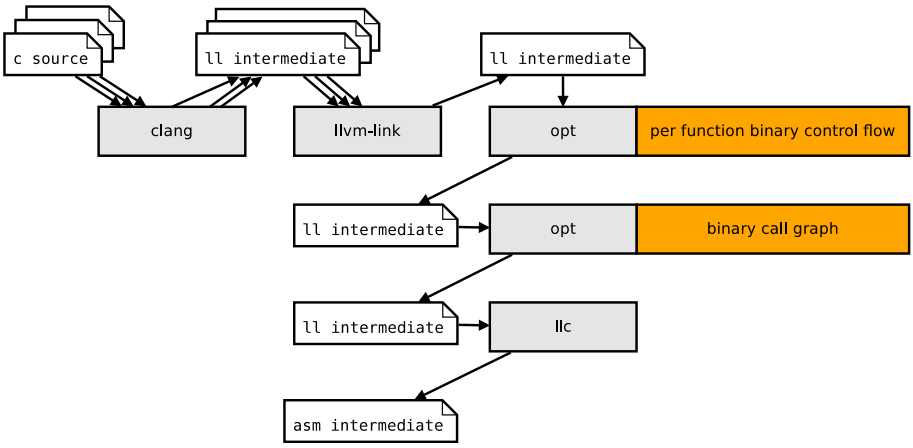

Figure 4.15: Compiler Stage of the toolchain transforming C code to SPARC assembler code.

code size and execution time [3].

The second patch for the SPARC backend was written to satisfy the requirements imposed by SOFIA. It consists of two LLVM machine function passes. One transforms the program into block form and marks the beginning of each block. The other ensures that multiplexer blocks can only be reached by explicit branching instructions, but never by falling through.

Assembler and Linker Stage

Aeroflex Gaisler's Bare C Cross Compiler is used to assemble the output of the compiler stage and link the resulting object files to a single ELF binary. The assembly file is passed to `as`, producing an object file. This object file is then stripped of its `comment`, `note.GNU-stack` and `eh_frame` sections using `objcopy`. The resulting object file is linked with the run-time environment using `ld`.

The run-time environment is implemented partly in C and assembler. The low-level assembler parts are responsible for clearing the BSS segment, providing access to the debug console, and passing control to the main routine. We wrote a small tool which transforms assembler code to SOFIA block form such that the startup code does not need to be transformed manually. This is necessary, because LLVM provides no frontend for SPARC assembler and thus, our optimization passes cannot be used for low-level assembler routines.

The tool is written in C++ and provides similar functionality compared to the optimization passes for LLVM. In particular, a binary control flow graph, block form, and header markers are applied to human readable assembly code. The final result, which is still readable assembly code, is then assembled and linked together with the actual program code.

After this stage, the binary is in block form with marked headers and has a binary control flow graph. All non-cryptographic constraints are satisfied, except the exact offsets for jumps to multiplexer blocks. The adjustment of these offsets, MAC calculation, and encryption of the actual blocks is done after linking, because then all locations are resolved and the offsets necessary for encryption can be obtained.

Post-linkage Stage

The post-linkage stage has been implemented as a number of standalone C++ applications as shown in Figure 4.16. Those applications are responsible for encrypting the binary, applying the jump offsets to multiplexer blocks, and providing verification routines.

The linked ELF binary from the assembler stage is first passed to `find_blocks` which finds all possible control flow paths and identifies reachable blocks. The list of reachable blocks is later needed by the encryption tool, as the cipher is parametrized with the current and previous program counter. The control flow graph is generated statically by examining the jump targets at the end of each block.

Secondly, with `mark_blocks`, jumps to multiplexer blocks are adjusted to jump to the correct instruction, i.e., an offset is added. Furthermore, blocks are marked as execution or multiplexer block and all necessary information for encryption is prepared and added to the header of each block.

Finally, the encryption tool `encrypt` encrypts all blocks, with either RECTANGLE-80 or PRINCE, and replaces the block headers with MACs. It outputs the final encrypted ELF binary, but also supports the output of a plain version which contains all instructions in clear and places `nop`s instead of the MAC words within the header. The plain version runs on an unmodified LEON3 processor to simplify debugging.

In addition to the plain version, verification tools are provided that statically check the constraints imposed by SOFIA. The non-cryptographic constraints are verified declaratively in several python scripts, while the correctness of the encrypted data and MAC words are verified by a simple simulator implemented in C++.

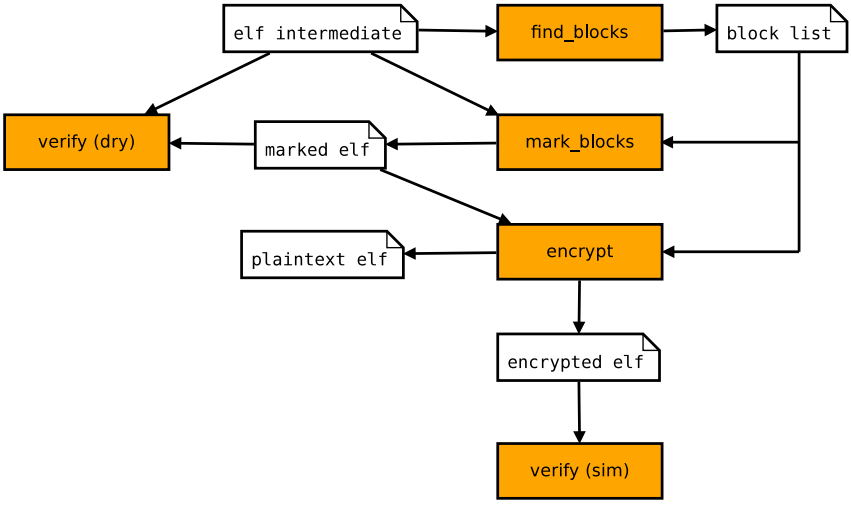

Figure 4.16: The post-linkage part of the toolchain is responsible for identifying blocks, adjusting offsets, and finally encrypting each reachable block.

4.5.2 Toolchain Implementation

The different stages of our toolchain produce separate result files in such a way that the following stage uses the output file of the previous stage as input. To simplify the overall compilation process, *CMake* is used to connect the tools and resolve dependencies. In this section we will describe some implementation aspects of our software toolchain in more detail. The constraints for SOFIA are satisfied at different levels. The binary control flow graph, binary call graph, and a single return per function are satisfied with the help of optimization passes, while the fixed block length, single branch instruction per block, position of branching instructions, the position of store instructions are satisfied by backend patches. All remaining constraints, such as encryption and MACs are then satisfied during the post-linkage stage.

Optimization Passes

Optimization passes in LLVM are shared objects that are dynamically loaded and executed by `opt`. They exclusively work on LLVM intermediate representation and can perform any transformations that does not depend on a particular target machine. Using optimization passes has the advantage that the compiler

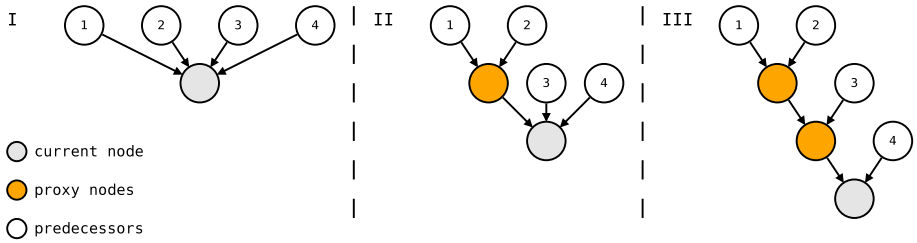

Figure 4.17: Example of the iterative transformation ensuring a binary control flow graph. Proxy nodes are added until every node has at most two predecessors.

does not need to be patched and that they can easily be loaded on demand by the optimizer. Furthermore, optimization passes are the best place to implement control flow-related transformations, as LLVM provides an interface to access the different nodes within the control flow graph, and it is possible to rearrange or replace nodes at this level. We used two optimization passes to implement the transformation ensuring a binary control flow graph, i.e., at most two predecessors are allowed per node, within functions and between functions, respectively. This transformation is implemented using an iterative algorithm which operates locally on a node inside a tree and creates proxies to bundle predecessors, until there are at most two predecessors left.

An example run of our algorithm is shown in Figure 4.17. The current node is shown in blue and it has four predecessors before the transformation (I). In the first iteration (II) the first two predecessors, i.e. 1 and 2, are removed and a proxy node (orange) is created. The current node now still has three predecessors, thus in the second iteration (III) the newly created proxy node and the third original predecessor 3 are chosen and another final proxy node is created for those nodes.

Currently, the choice of the nodes to be bundled in the proxy is arbitrary, as long as they are distinct. However, this choice determines the shape of the resulting tree and provides future optimization opportunities regarding the overall run-time performance.

Backend Changes

The LLVM intermediate representation offers a convenient way of describing transformations on basic blocks and functions. However, it is agnostic to the target processor's instruction set and can therefore not be used to satisfy low-level constraints such as the exact position or type of a single assembler instruction.

To gain control of architecture-specific instructions, some constraints were implemented in the SPARC backend of LLVM. The backend has access to the SPARC instruction set and offers a lower level view. LLVM intermediate functions are transformed to machine functions and intermediate basic blocks are converted to machine basic blocks. However, the drawback of implementing backend changes is that they cannot be implemented as separate passes, but instead the compiler itself needs to be patched.

Two machine function passes were written to satisfy the low-level instruction constrains of SOFIA. First, the multiplexer fall-through pass ensures that no multiplexer block is reached without an explicit jump. Next, the basic block inflator lays out the code in SOFIA block form.

The transformation carried out by the basic block inflator is strictly local to basic blocks. It takes a sequence of SPARC instructions and inserts `nop`s between them until the sequence fits the desired form. The algorithm used to reach this form consists of two phases. In the first phase, all instructions are scanned and an abstract record of blocks is built. This record, named *SofiaBlockSequence*, consists of a sequence of structures called *SofiaBlocks*. Each SofiaBlock holds a part of the input instruction sequence, the block type (execution or multiplexer), and the padding required between the instructions.

The algorithm walks through the sequence of instructions and tries to insert them into the current SofiaBlock. The insert operation takes into account the padding required to move the instruction forward and if it has to be at or beyond a specific position inside a block. All instructions that alter control flow, like calls, `ret`s and branches, must be moved to position 7. The instructions that write to memory are moved beyond position 4. If the instruction cannot be placed in the current block, the block is inserted into the SofiaBlockSequence and a new block is created. This may happen either because the block is completely filled or because the required position is already occupied. In any case, each block which is inserted into the SofiaBlockSequence is padded to a fixed size of eight instructions.

This first phase is complete when all instructions are placed in the SofiaBlockSequence. At this point all headers and the neccessary padding before and after instructions are known. The second phase walks over all instructions again and inserts padding and header markers into the actual instruction sequence. The header markers are `unimp` instructions and are always the first instruction of a block. Execution blocks are marked with `unimp 0x50F1A`, while multiplexer blocks are marked with `unimp 0x50F1B`. Padding is realized by inserting `nop` instructions. An example of how the basic block inflator works is shown in Figure 4.18. It is assumed that the first store instruction is reachable from two blocks, i.e., the first SOFIA block is a multiplexer block.

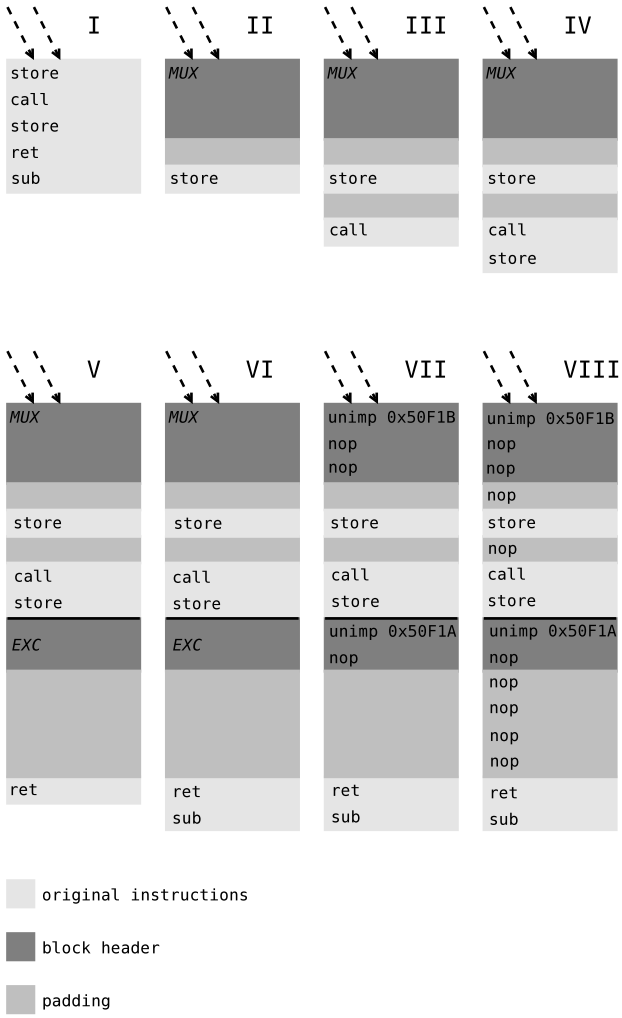

Figure 4.18: Example of how the SOFIA basic block inflator transforms a sequence of assembler instructions to satisfy all low-level constraints. SPARC uses delayed branching, which means that the instruction after a branch is executed before the branch takes effect. Therefore, the `ret` instruction is placed on the second-to-last element in the memory block.

Cryptographic Operations

The last stage of the toolchain is implemented as several standalone tools written in C++. The tools take the compiled and linked binary emitted by Gaisler's

Bare C Compiler as input. This binary is then encrypted as required by the current implementation of the SOFIA processor. The user can choose between the ciphers RECTANGLE-80 and PRINCE to generate the MAC and encrypt the block. In addition to satisfying the cryptographic constraints, jumps to multiplexer blocks are adjusted and the markers introduced by our backend machine passes are replaced by the correct MAC words. This step has been postponed until after linking, because the offset calculation requires absolute jump targets.

4.5.3 Limitations

This section discusses the limitations of the current implementation of the SOFIA toolchain.

SOFIA relies on a precise CFG to perform the necessary software transformations. Polymorphism and calculated jumps are currently not supported because they make it difficult to build a precise CFG. The problem of building a CFG from calculated jumps is often approached by over-approximation [70], which leads to losing some security guarantees since an attacker might take paths which do not exist within the real program. Thus, if we would have tool support for creating a precise CFG which allows the jump targets of each calculated jump to be known, it would be possible to support both calculated jumps and polymorphism.

SPARC register windows are currently not supported because they trigger a window overflow or underflow trap when the current window pointer coincides with an invalid window. Therefore, in order to be able to support register windows, the SOFIA hardware needs to be updated to support interrupts (see Section 4.4.5).

In our evaluation we only considered baremetal applications. However, this is not a fundamental limitation. To provide *microkernel* support would be challenging, since this kernel type consists of several separate processes, with each running in its own address space. SOFIA requires its software to be inside a single address space, since a single transformation needs to be performed for each process's software. This is especially problematic for shared libraries, which have a unique address space inside each used process. This can be solved (in part) by using a *monolithic kernel*, which uses a single large process with one address space. User-mode applications can be supported by disabling shared libraries. To enable task scheduling, interrupt support can be added (see Section 4.4.5).

Our toolchain currently does not support software-based floating point

operations. This can be fixed by implementing optimization passes which replace floating point operations by calls to library functions and ensure compatibility to existing optimization passes.

The current toolchain does not support any compiler optimizations (only `-O0` is supported). A major problem is that some optimization passes on the LLVM intermediate representation leads to the binary control flow imposed by the SOFIA-specific LLVM passes to become undone.

4.6 Evaluation

In this section we evaluate the security of the architecture, followed by an evaluation of hardware and performance overheads of our implementation.

4.6.1 Security Evaluation

Software Integrity (SI)

The SI property is considered equivalent to forging a MAC. An attack is successful if an adversary alters an instruction and MAC pair so that the integrity verification succeeds.

The bit length of a MAC is directly related to the number of trials that need to be performed before a forged message and MAC pair is accepted. For an n-bit MAC, an adversary has to perform an average of 2^{n-1} random online MAC verifications before this strategy will succeed [59]. Consider that a 64-bit MAC is used, and that an attacker requires at least 8 cycles to verify a forging attempt of a single execution block on the target platform. Then, a successful forgery will require at least 33,001 years to succeed on a 70.6 MHz SOFIA core.

Control Flow Integrity (CFI)

The CFI property is also considered equivalent to forging a MAC. An attack is successful if an adversary deviates control flow from the valid CFG so that the integrity verification succeeds.

An attack on the control flow requires two steps. First, the adversary has to divert control flow (e.g., through ROP), from one memory block to another. Second, the adversary has to forge the MAC on the decrypted instructions of the second memory block. Executing the first block will require 8 cycles, while

the MAC verification of the tampered block will require an additional 8 cycles, leading to a minimum of 16 cycles per attempt. Therefore, an online brute force attack on a 64-bit MAC will require at least 66,002 years on a 70.6 MHz SOFIA core.

Tampered code protection

The tampered code protection provided by SOFIA relies on both the CFI and SI properties. At run-time, any code tampering will be detected when executing an execution/multiplexer block. Not only will the tampering be detected, but the detection will happen before write operations occur. This allows the system to ensure that a tampered block, or a block resuling from tampered control flow, will not have a chance to execute a bad instruction that can lead to a tampered write operation. It is especially important to protect write operations in the context of cyber physical systems where physical components interface with the processor via a port or memory mapped interface.

Code confidentiality

The CFI mechanism decrypts instructions at run-time using a key that is unique to each device. This key is only known by the device manufacturer and by the software provider. Since the software is encrypted for a single device, it can therefore only be decrypted by that device. This prevents an attacker from copying the encrypted software and running it on another device. It is also not possible for an attacker to extract plaintext instructions from a device, as the software is stored encrypted in the cache. By ensuring that code remains confidential, it prevents the reverse engineering of the software to find potential exploits or to obtain a vendor's software IP.

Fault attack protection

In SOFIA, instructions and MAC tags are stored encrypted in cache and main memory, and are only decrypted before they are requested by the IF pipeline stage. This allows SOFIA to detect tampered instructions/MAC tags due to fault attacks on the main memory, the instruction cache, or due to control flow glitches, as explained below.

A well-known fault-based attack on control flow is to glitch the external clock line [15]. This typically involves temporarily reducing the clock period, thereby causing the processor to skip instruction(s). In our implementation the block

Table 4.3: The hardware overhead of SOFIA.

Design	Slices	LUTs	Flip Flops	Clock (ns)
LEON3	6,052	14,222	11,135	92.3 MHz
SOFIA w/ RECTANGLE	7,208	15,909	11,714	60 MHz
SOFIA w/ PRINCE	6,728	15,180	11,602	70.6 MHz

cipher is in the critical path of the processor. Therefore, when the clock period is reduced, the block cipher will be the first component to fail, as its path is the longest in the entire design, meaning that it requires the most amount of time in each clock cycle to perform a computation. In our implementation of SOFIA the block cipher is utilized in all pipeline slots. This means that when the clock period is reduced by a sufficiently large amount (i.e. by glitching the clock), the block cipher will not have had enough time to finish the computation. When the clock is glitched for as little as one clock cycle, the cipher's operation will be incorrect, leading to a MAC failure. The reason for this is that the MAC can only be calculated if all the cipher operations of a block are computed correctly.

Another well-known fault-attack mechanism is to glitch the power line of a processor [13]. If the external power supply voltage deviates by more than 10% it could cause problems with the functionality of the IC. This could lead to wrong computation result of the processor. SOFIA cannot make any guarantees about the correct execution of the instructions inside the instruction pipeline. However, since the block cipher is in the critical path of the processor, it is highly likely to be the first computation to fail due to a power glitch. If a block cipher operation fails to compute, a MAC failure occurs, and the processor is reset.

4.6.2 Hardware Evaluation

Table 4.3 shows the hardware overhead of the two different SOFIA implementations, with each respectively using RECTANGLE or PRINCE, compared to a LEON3 core. We found that the LUTs increased by 12.9%, while the clock speed reduced by 23.2% when compared to an unmodified LEON3 core. The clock speed reduction is due to the block cipher being in the critical path of the design.

4.6.3 Performance Evaluation

To evaluate the performance of our SOFIA implementations we selected the following software benchmarks applicable to small embedded processors:

- MiBench [58]:
  - `ADPCM`: This Adaptive Differential Pulse Code Manipulation (ADPCM) implementation converts an audio file of 16-bit PCM samples into 4-bit samples, thereby yielding a 4× compression rate.
  - `qsort_small`: The *quicksort* algorithm is used to sort an array of strings into ascending order.
  - `patricia`: Practical Algorithm to Retrieve Information Coded as Alphanumeric (Patricia) is an algorithm used for routing table lookups. The input data consists of a list of IP traffic from a web server.
  - `bitcount`: Performs bit manipulation by counting the bits in an array of integers using five different methods.
  - `crc32`: A Cyclic Redundancy Check (CRC) is an operation that is commonly used to detect transmission errors.
- `EEMBC CoreMark [47]`. This benchmark performs list processing (find and sort), matrix manipulation, state machine, and CRC computation.
- `AES`: This benchmark performs ECB-mode encryptions using AES-128.

All benchmarks were executed baremetal on three different cores: a LEON3 core clocked at 92.3 MHz, a RECTANGLE-80-based SOFIA core clocked at 60 MHz, and a PRINCE-based SOFIA core clocked at 70.6 MHz. The LEON3 code was compiled with LLVM, while SOFIA code was compile with the SOFIA toolchain, with both using the compiler flags: `-O0 -target sparc -m32 -S -emit-llvm -mcpu=v8`. Figure 4.19 shows the cycle overhead of executing the benchmarks on the two SOFIA cores compared to the LEON3 processor. For the RECTANGLE-80 implementation we measured an average cycle overhead of 149%, with AES having the smallest overhead of 36%, and CoreMark having the largest overhead of 438%. For the PRINCE-based SOFIA implementation we measured an average cycle overhead of 141%, with AES having the smallest overhead of 44%, and CoreMark having the largest overhead of 373%. Figure 4.20 shows the execution time overhead of executing the benchmarks on the two SOFIA cores compared to executing the benchmarks on the stock LEON3 processor. For the RECTANGLE-80 based SOFIA implementation we measured an average total execution time overhead of 106%, with `crc32` having the

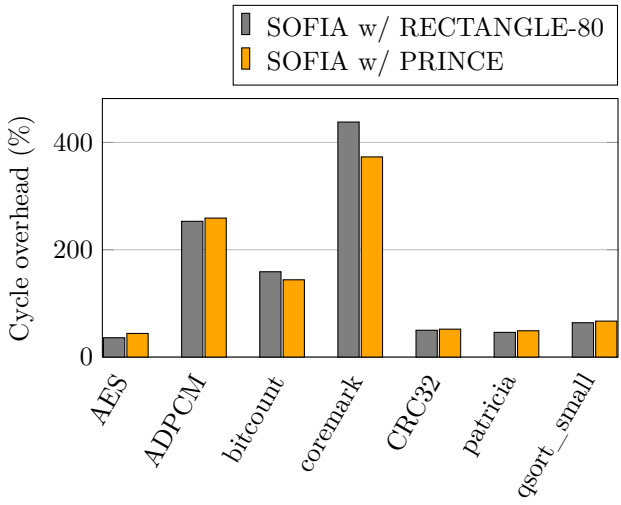

Figure 4.19: A comparison of the cycle overhead of benchmarks running on a SOFIA core compared to a stock LEON3 processor clocked at 92.3 MHz.

Table 4.4: A comparison of the code size of the benchmarks compiled for both a SOFIA core and for a stock LEON3 processor.

Benchmark	**LEON3 code (bytes)**	**SOFIA code (bytes)**	**Overhead (%)**
ADPCM	4,080	12,480	205
AES	7,184	38,624	437
Bitcount	12,272	25,834	110
CoreMark	22,576	68,640	204
CRC32	8,736	23,488	169
Patricia	8,496	23,136	172
qsort_small	10,896	23,420	123

smallest overhead of 0.53%, and CoreMark having the largest overhead of 726%. For the PRINCE based SOFIA implementation we measured an average total execution time overhead of 137%, with `crc32` having the smallest overhead of 30%, and CoreMark having the largest overhead of 516%. A comparison of the code size is shown in Table 4.4, and we found that the code size increases by an average of 203%.

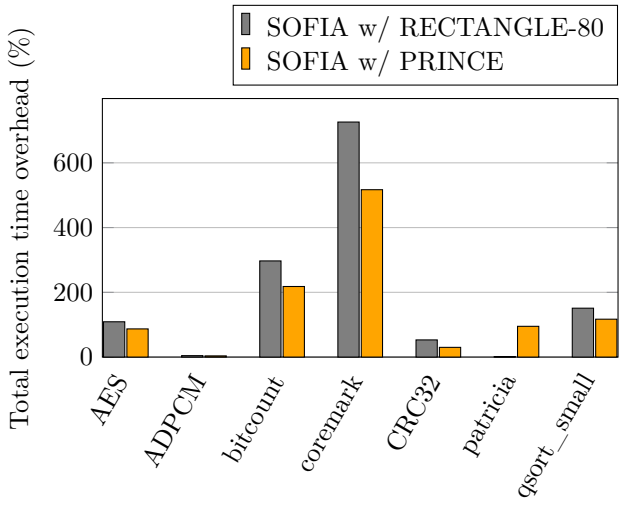

Figure 4.20: A comparison of the total execution time overhead of benchmarks running on a SOFIA core compared to a stock LEON3 processor clocked at 92.3 MHz.

4.6.4 Practical feasibility in time constrained cyber physical systems

SOFIA's performance evaluation, reported in Section 4.6.3, show significant overhead in terms of memory, clock delay, and execution time. Cyber physical systems typically make use of real-time programs to guarantee a timely response to critical events. Longer execution times increase the difficulty of meeting deadlines. This raises concerns about the practicality of using SOFIA in a time-constrained cyber physical system application. First, the clock delay overhead increases execution time for any program running on a SOFIA core. However, we feel that a clock speed reduction of only 23.2% does not significantly limit the practical feasibility of using SOFIA in a time constrained cyber physical system application.

Second, the memory overhead further increases execution time, since more reads are required from main memory. To mitigate this the processor's cache size can be increased.

Third, the increase in executed instructions, such as `nop`s and jumps, increase the overall execution time. However, the effect of `nop`s are minimal because they execute in one cycle, i.e., they do not cause pipeline stalls due to data

dependencies or branch mispredictions. Consequently, the increase in executed instructions is dominated by multiplexer blocks. Large multiplexer trees are especially prevalent at the entry points for frequently used functions. These large trees contain long paths requiring several jumps to reach the intended callee. It is therefore desirable to keep the number of multiplexer blocks inside critical software to a minimum. One approach to achieve this is to write code that ensures that critical functions are called by the minimum number of callers. In addition, the software inside a critical function should contain the minimum number of jumps and calls. Another approach is to modify the toolchain to force it to assign short paths for time-critical functions. Yet another approach would be to duplicate frequently used critical functions.

4.7 Conclusion

In this chapter, we presented SOFIA, a novel architecture which enforces CFI, SI, code confidentiality, and reverse engineering protection. SOFIA is the first known architecture to enforce CFI through instruction-set randomization, where the instructions are decrypted at runtime with control flow dependent information. The architecture's security policies are enforced in hardware through modifications to the processor's instruction pipeline. To evaluate the design, we integrated SOFIA with a LEON3 core, and made an FPGA-based hardware implementation. The SOFIA core increased the hardware area of the LEON3 core by 12.9%, and reduced the maximum clock frequency by 23.2%. In addition, a software toolchain was developed to transform software written in C to conform to the constraints imposed by our architecture. We compiled several software benchmarks with our toolchain to evaluate the overhead imposed by the SOFIA architecture. Altogether SOFIA imposes an average cycle overhead of 141%, and an average total execution time overhead of 106% when compared to a stock LEON3 core.

Chapter 5

SCM: Secure Code Memory Architecture

Content Sources

de Clercq, R., de Keulenaer, R., Maene, P., Preneel, B., De Sutter, B., and Verbauwhede, I. SCM: Secure Code Memory Architecture. In *Proceedings of the ACM on Asia Conference on Computer and Communications Security* (2017), ASIACCS'17, ACM, pp. 771–776
Contribution: Responsible for hardware designs. Some concepts are the result of brainstorming sessions with co-authors.

In the previous chapter, we introduced SOFIA, while this chapter introduces SCM, a lightweight alternative to SOFIA with reduced functionality. The differences between SOFIA and SCM are as follows. First, SOFIA enforces both CFI and SI, while SCM only enforces SI. This means that both architectures can prevent the execution of injected/tampered code, but only SOFIA can prevent attacks on control flow. Second, SOFIA requires integration with the processor, while SCM is implemented as an IP core. Third, SCM requires only a simple code transformation, while SOFIA imposes a more complicated set of constraints on the software.

5.1 Introduction

Ensuring that attackers cannot perform code injection or code tampering is an important first line of defence against runtime attacks. This is especially important when relying on software to perform security-critical or safety-critical operations. Most modern processor provide page-based protection mechanisms, such as $W \oplus X$, which ensure that code cannot be modified from user space. However, page-based protection can be disabled with a syscall to `VirtualProtect()/mprotect`, which allows usermode applications to change page permissions. In addition, page-based protection cannot prevent fault attacks on the contents of untrusted off-chip memory. Software Integrity (SI) is an alternative to $W \oplus X$, and works by verifying the integrity of software at runtime to ensure that the code has not been tampered with. This can lead to a smaller TCB, since no OS feature needs to be trusted (see Section 3.3.2). SI is also a security feature relied upon by some CFI architectures (see Chapter 3 and Chapter 4) to protect code memory against tampering.

In this chapter, we propose a new Software Integrity (SI) architecture called SCM. It is realized as an IP core that verifies the integrity of code as it is fetched from external memory into the caches of a processor on an SoC. This IP core communicates via standard bus interfaces, which means it can be added to existing SoC designs without requiring any changes to the used IP cores, memories, or interfaces; and without requiring any changes to existing SoC design flows. In other words, none of the already used components needs to feature any security support to provide SI. Moreover, our design requires only minor adaptations to the software build process and offers flexibility in supporting a range of reaction mechanisms. This includes the guarantee that not a single tampered instruction can be executed. Furthermore, our solution fits into many schemes to sign and distribute software. Finally, as we will demonstrate, the performance overhead of SCM is limited.

Many existing works [23,42,44,45,84,99,112,113] provide SI guarantees. However, most of these works require modification to the processor, memory hierarchy, or existing IP cores. Modifying existing IP is difficult, as it requires modification by the IP vendor (which could be expensive), or requires the IP vendor to release it's hardware source files to the client (which could be even more expensive). SCM provides integrity guarantees without making any modification to the existing IP on a SoC. The only requirement is that our IP core needs to be connected to the bus of a SoC, which is a relatively simple task. In this regard, our approach has some similarities with SecBus [19]. However, SCM only uses standard bus interfaces, while SecBus interfaces with a memory controller as well as the bus. Many proposed solutions [23, 44, 45, 84, 99, 112, 113], including the Memory Encryption Engine [55] of Intel SGX, rely on tree structures [100]

to protect write operations to memory. This is a much more heavy-weight approach, since an integrity tree requires additional storage, additional memory accesses, and additional integrity checks for each transaction. All of these require extra area, which is not appropriate for small embedded platforms.

The remainder of this chapter is structured as follows. First, we provide a problem statement, which includes a threat model and the system goals. Next, we present the architecture of SCM, followed, by a discussion of our prototype implementation. Afterwards, we evaluate our prototype implementation in terms of the hardware overhead, the system performance, and the security, followed by a conclusion.

5.2 Problem Statement

5.2.1 Threat Model

The goal of the attacker is to execute tampered code on the system. We focus on static, native code, and exclude just-in-time compiled code or self-modifying code. On many instruction set architectures, instructions alone do not express the semantics of a program efficiently. Instead, instructions are complemented by read-only data. In the remainder of the chapter, we use the term "code" as shorthand for instructions and read-only data.

We consider attackers with three powerful capabilities. First, they can tamper with code after it has been built and before it is installed on a device. Second, they are in control of all addressable off-chip memory. Third, they can perform fault attacks on off-chip memory. This includes physical fault attacks, and software-based fault attacks, such as Rowhammer [69, 89].

We use the Dolev-Yao [41] model, which assumes attackers can not break crypto primitives, but can perform protocol-level attacks. We furthermore assume the attacker controls the SoC's digital inputs, such as the General-Purpose Input/Output, off-chip memory, and networking signals. While we assume the attacker can perform physical attacks, such as fault attacks and side-channel analysis, on off-chip memory, we assume he cannot perform physical attacks on the SoC itself.

5.2.2 System Goal

The goal of SCM is to verify the integrity of code stored in off-chip memory. One might argue that mutable data needs to be protected as well, seeing as the

initial values of global, statically allocated and initialized arrays also part of the program semantics. Protecting mutable memory is out of the scope. However, it is simple to let compilers generate code such that all static initialization values are stored in read-only data sections, not in mutable data sections.

To enable fluent integration into a SoC, SCM is implemented as a standalone IP core that connects to the bus. By doing so, SCM provides the SoC with security guarantees without requiring modification of other IP cores in the SoC. Furthermore, we aim for minimal disruption of the traditional SoC design cycle, by only building on pre-existing interfaces and composition schemes. This is important, as it enables rapid integration of SCM into SoCs, and improves prototyping, development, and production costs.

SCM should provide protection from the following types of attacks (1) *spoofing*: bits are illegitimately modified, (2) *splicing*: bits are illegitimately relocated, and (3) *replay*: fresh bits (e.g., from an updated program version) are partially substituted with stale bits (e.g., from an outdated program version or another user's program version). SCM works on the principle of verifying the integrity of bits that have been fetched from memory, and more specifically from the code sections and the read-only data sections of binaries as they have been allocated in memory.

In the remainder of this chapter we will use the term *memory* to refer to any off-chip memory.

5.3 SCM Design

5.3.1 Conceptual Overview

Modern processors issue memory requests to the memory hierarchy to fetch code to be executed. To achieve the system goal, SCM verifies the integrity of code bytes that have been read from an external memory before they are processed by the processor.

The flow of instructions through the system is shown in Figure 5.1. Using SCM, memory can be requested from either the protected SCM memory region, or from untrusted memory. *Unprotected code* is requested directly from untrusted memory, where it will be stored in caches before being used by the processor. The unprotected parts include the mutable memory regions, but can also include parts of a program that do not require protection. The flexibility to select parts of the software to be protected eases the integration of our solution with existing software.

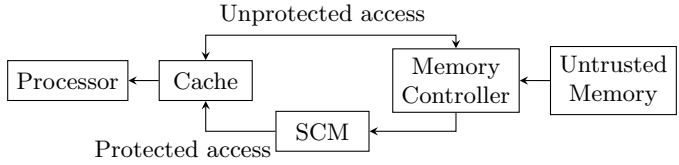

Figure 5.1: Flow of code and data through system.

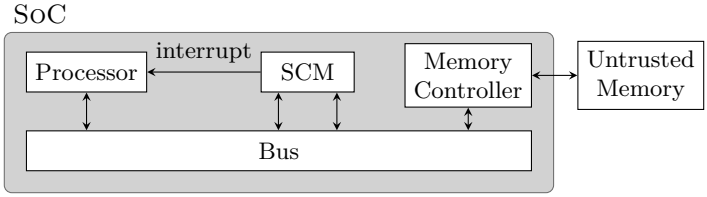

Figure 5.2: Architectural overview of the system

To access and execute a *protected program*, a group of instructions and integrity information are first fetched from the untrusted memory by the memory controller. Next, those bytes are sent to SCM, which verifies their integrity. If the integrity verification succeeds, the group of bytes is passed on to the caches and the processor. If an integrity check fails, SCM will not forward the tampered instructions to the processor. In addition, the processor is notified by means of an interrupt that a security exception has occurred. The processor can then take appropriate action, depending on the specific use-case for the hardware containing SCM.

5.3.2 Architecture

The architecture shown in Figure 5.2 consists of a SoC and untrusted off-chip memory. The SoC consists of a number of different IP cores, including SCM, a memory controller, and a processor that includes a number of caches. Each of these IP cores are able to communicate via the SoC's main bus. The memory controller acts as an interface between the SoC and the untrusted off-chip memory. SCM is responsible for delivering integrity checked code to the processor.

SCM Memory

SCM has a read-only memory region associated with it, which we call *SCM memory*. Each address in the SCM memory region maps to a physical memory address, as described in more detail below. The physical address can be assigned to any untrusted memory, including ROM, DRAM or flash memory. A bus *transaction* is the sequence of bus actions that are needed to perform a read or write. Whenever a bus transaction requests a read from the SCM memory region, SCM needs to respond by delivering integrity-verified bytes. This non-trivial procedure requires the following steps. First, upon receiving the read request from the processor, SCM needs to fetch the bytes from the matching physical memory address by placing a read request on the bus. Second, after the requested bytes have been received from the bus, an integrity verification is performed. Third, if the integrity check succeeds, the requested bytes are delivered to the bus. If the integrity verification fails, dummy values are delivered to the bus instead.

Since integrity computations can introduce a large overhead, the integrity checking algorithm and security parameters need to be chosen carefully to ensure a low overhead.

With the addition of the SCM memory region, we effectively split the address range of a program's main memory into a secured and an unsecured region.

Two port interface

SCM needs to perform two simultaneous bus transactions. One transaction is needed for the processor's request to SCM memory, while the second is needed to fetch the code fragment and integrity information from unprotected physical memory. We propose to solve this by using two ports to interface with the bus. Components connected to a bus follow the master/slave communication model. Therefore, we use a slave port to receive read transactions in the SCM memory region, and a master port to perform read transactions from physical memory.

Integrity verification

We propose to use a Message Authentication Code (MAC) algorithm to verify the integrity and authenticity of code. An m-word MAC is *precomputed* on each group of n code words. The MACs are stored interleaved with code in untrusted memory. We use the term *memory block* to refer to the group of

n code words and m precomputed MAC words. At *run time*, each memory block's integrity is verified before delivering its code words to the bus.

The MAC algorithm uses a secret key that is deeply embedded in the hardware and is only accessible by the MAC algorithm. In addition, the key is only known by the software provider. Since the MAC key is not known to the attacker, he cannot forge a MAC without being detected.

The system needs to protect against spoofing, splicing, and replay attacks (see Section 5.2.2). Computing a MAC over the code words (MAC($inst_1 \parallel \cdots \parallel inst_n$)), leaves the system vulnerable to a splicing attack, as relocated memory blocks would not be detected. This issue can be exploited by an attacker by rearranging existing memory blocks in order to craft malicious code that cannot be detected by SCM. To prevent this attack, we could use MAC($addr \parallel inst_1 \parallel \cdots \parallel inst_n$), where $addr$ is the *physical address* of the memory block. This allows the system to detect any changes to the location of a memory block. However, a replay attack is still possible. Consider the scenario where multiple different programs were transformed under the same key. An attacker can then copy a memory block from one program at $addr_1$ to another program at $addr_1$ without detection. To solve this, we propose to use MAC($addr \parallel \omega \parallel inst_1 \parallel \cdots \parallel inst_n$), where a nonce ω is unique across different programs and different program versions.

Memory map

Each group of n words in the SCM memory region maps to $n + m$ words in physical memory, as shown in Figure 5.3. When a group of instructions is fetched from SCM, the m MAC words are stripped out and only the requested instructions are sent to the processor. This has the advantage that the processor works with a continuous address range that does not contain MAC words.

As shown in Figure 5.1, code typically reaches the processor via caches. To exploit this, parameter n is chosen to match the cache line size. This ensures that each cache line read from SCM memory can be handled by verifying and fetching exactly one memory block.

Software support

The software transformation process is done as follows. First, a customized linker script forces immutable sections in protected segments (in the SCM memory region), while mutable sections are placed in unprotected segments (in

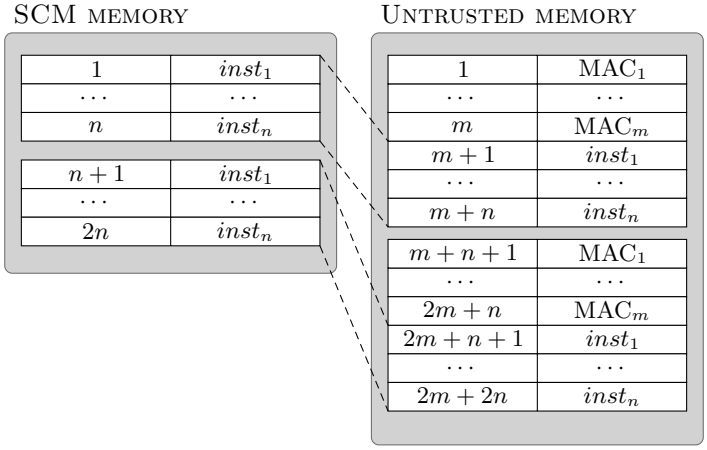

Figure 5.3: Memory mapping between the SCM memory range and untrusted memory.

unprotected memory). This ensures that the program can execute from the SCM memory region.

Afterwards, the MAC precomputation is done. First, the protected segments of the compiled binary is disassembled. Next, a script calculates a MAC on each group of n opcodes. The MAC is stored interleaved with the opcodes. Finally, the transformed segments as well as the unprotected segments are compiled with a linker script that places both segments in unprotected memory.

Finally, the binary is copied to untrusted memory, and is executed from the SCM memory region.

Integrity failures

If an integrity check fails, SCM needs to initiate an appropriate response to recover from the exception. What is appropriate depends on the use case, and hence will differ for each piece of software. While the development of a recovery mechanism for a specific use case is out of scope for this work, several recovery options are supported by SCM.

One approach is to reset the processor upon detection of an integrity exception. However, this cannot be tolerated by some systems, including safety-critical and real-time systems. Another approach is to reload and restart the program. A more complex option is to increment a counter every time a program is restarted

due to an integrity failure, and then rebooting when a threshold value is reached. Some forms of graceful degradation might be useful, or sending notification to online monitoring services.

To provide the necessary flexibility, i.e., to support many forms of reactions in a programmable manner, we designed a generic hardware/software mechanism for SCM to invoke recovery functionality. This mechanism relies on non-maskable interrupts that SCM generates upon integrity failures. When the interrupt occurs, the processor stops executing the current instruction, and transfers control to an interrupt handler. In this handler, any reaction can be programmed, including reloading a program from flash memory to restart a task from a consistent state, or rebooting the processor.

The flexibility of interrupt-based integrity failure handling introduces a potential security problem, as an attacker could also alter the interrupt handler software before an integrity failure occurs. This would prevent the processor from correctly responding to the integrity failure. To address this problem the interrupt handler code can be stored on a small amount of secure memory (e.g., on-chip ROM). Alternatively, a more flexible approach could be to store the interrupt handler in a small SRAM controlled exclusively by SCM. It is then critical that some restrictions be enforced on programming this memory, such as only allowing the memory to be programmed once during boot while the processor is in supervisor mode.

5.4 Prototype Implementation

5.4.1 Target Platform

Zynq SoCs are used to prototype IP cores before fabrication in silicon. Each Zynq SoC contains an FPGA, known as the Programmable Logic (PL), and a non-programmable Processing System (PS), as shown in Figure 5.4. The PS features a dual-core ARM Cortex-A9 processor, a memory controller, and ports to communicate with the PL. The AXI4 [11] bus is used for communicating between IP cores located on either the PS or the PL.

AXI4 supports three interface types. The *AXI-Stream* protocol allows two components to communicate without the bus. The *AXI-Full* protocol is used to transfer large amounts of data via the bus, and supports burst mode transfers. The *AXI-Lite* protocol is used for low speed communication, e.g., memory mapped registers, and implements only a subset of the features of the AXI-Full interface.

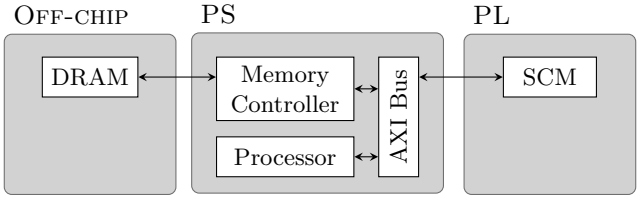

Figure 5.4: System overview.

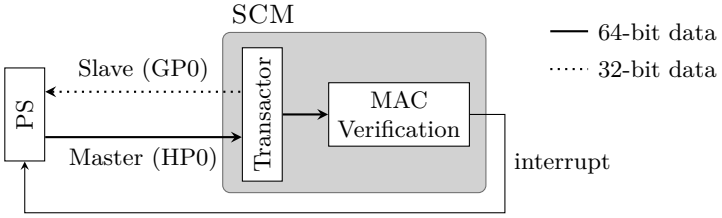

Figure 5.5: The implemented architecture of SCM.

The PS and PL interface with each other via two different types of AXI-based ports. First, the PS can access PL slave devices via general-purpose ports. Second, PL master devices can access the PS, which includes off-chip memory, via AXI high-performance ports. The general-purpose and high-performance ports both support burst transactions, with data widths of 32-bit and 64-bit, respectively.

As shown in Figure 5.5, SCM consists of two subcomponents. The *Transactor* coordinates memory accesses to the PS, while the *MAC verification* component verifies the integrity of memory blocks.

5.4.2 Transactor

The Transactor handles receives requests from the PS, reads memory blocks from the PS, transfers memory blocks to the MAC verification component, and delivers integrity checked code to the PS.

Interfaces

An AXI-Full slave port allows the PS to read protected code from the SCM memory range. It is configured to allow for 128 MB of SCM memory, which

the PS accesses via the 32-bit GP0 port. The 128 MB of SCM memory maps to 160 MB of DRAM, located in the PS.

The Transactor needs a mechanism to fetch a memory block from physical memory after receiving an SCM memory read request. For this mechanism, we evaluated two options. First, using the Xilinx DMA IP core, we measured it takes 60 cycles to receive the first data of a burst read operation from the physical memory. Second, a 64-bit AXI-Full master port connected to the Zynq's high-performance HP0 port allows for performing burst read operations. We measured that at least 20 cycles is required to start receiving data after issuing a burst read request from the PS. Therefore, the AXI-Full approach was used in our prototype.

After receiving the memory block from the PS, the Transactor passes it to the MAC verification component, which performs the verification before delivering the code to the PS via the slave port. In order to avoid causing a data-abort exception or freezing the PS, it is essential that the slave port responds to read requests with the requested number of memory elements. So upon a verification failure, instead of sending the potentially tampered code bytes, the Transactor simply sends the required number of zero values.

Fetching memory blocks

Each group of n words in SCM memory space maps to a memory block of $n+m$ elements in physical memory (see Section 5.3.2). Since the cache line size of the ARM processor is eight 32-bit processor words, we select $n = 8$. To provide 64-bit security, we use a 64-bit MAC, for which we select $m = 2$. So to serve a read request of 8 words from SCM memory, we need to fetch 10 physical memory words.

With the AXI protocol, the number of words fetched in a burst operation must be a power of 2. We opted to use one 16-word burst of which 6 words are dropped over using an 8-word burst followed by a 2 word burst because every burst involves a 20 cycle delay before the first word arrives, regardless of the burst size. After that initial delay, one word is received every cycle. Furthermore, the extra power consumption of unnecessarily reading six more words is partially compensated by initiating one fewer transaction.

Overlapping read transactions

To allow instructions and read-only data to co-exist inside the SCM memory range, the Transactor's AXI-Full slave port needs to support overlapping read

transactions. Such transactions occur when a new read transaction is issued while the slave is busy processing another transaction. This can happen when a program executing from the SCM memory range executes a load instruction that fetches data from the SCM memory range. This presumably happens when the instruction prefetcher is busy with a speculative fetch from SCM memory, while at the same time a load occurs, causing the memory controller to issue another read request from the SCM memory range.

To support overlapping reads, the Transactor waits for the current read transaction to finish before processing the next. Therefore, registers should be used to store address read channel information, as this could be overwritten when a new transaction arrives.

5.4.3 MAC Verification

The MAC verification component performs integrity verification of memory blocks. A 64-bit AXI-Stream slave interface is used to receive memory blocks, addresses, and nonces. While data is received from the Transactor, the runtime MAC is calculated. For each memory block, tampering is detected by comparing the runtime MAC to the precomputed MAC. Only untampered code is forwarded to the PS, and detection of tampering fires an interrupt.

For the MAC cryptographic primitive we selected COPA's PMAC1 construction [9]. COPA is an Authenticated Encryption mode of operation for block ciphers, which means it can be used with any symmetric encryption algorithm. However, since we only require authentication, we only use PMAC1, and not the full implementation of COPA.

Although AES is used in COPA's original design, our implementation uses PRINCE [18], which is highly efficient [79]. We placed two pipeline registers inside our PRINCE implementation to allow the MAC verification component to meet the timing constraint of 100 MHz (see Section 5.5.2). A three cycle implementation of AES will likely have a huge overhead in terms of area and delay, as observed by [79] in a comparison of single cycle implementations. PRINCE's 64-bit block size allows for more effective use of the Zynq's 64-bit HP0 port, since memory blocks received from HP0 can immediately be processed by our MAC primitive.

The nonce is updated by writing to a special SCM register, thereby facilitating context-switches between programs with different nonces.

5.4.4 Integrity Violations

To handle integrity failures, we configured the PS to allow for fabric interrupts via an interrupt request line, and installed a software interrupt handler on the interrupt line. For our prototype, we implemented an interrupt handler that displays a message when such an interrupt occurs.

5.5 Evaluation

5.5.1 Security Evaluation

In SCM, memory tampering and MAC forgery are infeasible, since forged blocks can only be verified online. For an n-bit MAC, an adversary has to perform an average of 2^{n-1} random online MAC verifications before this strategy will succeed [59]. Therefore, a successful forgery of a memory block will require 70,193 years (on average) to succeed on a 100 MHz SCM core.

5.5.2 Hardware evaluation

We evaluated our design on a ZedBoard, which consists of a Xilinx XC7Z020-CLG484-1 FPGA SoC package. It features a dual-core 667 MHz ARM Cortex A9, 512 MB DDR3, 32 KB L1 cache for each core, and a 512 KB L2 cache. The processor supports prefetching of code and data before they are needed by the processor. The FPGA is comprised of 53,200 LUTs and 106,400 flip flops.

The Xilinx Vivado 2015.2 design suite was used for synthesising our hardware implementation. It uses an area of 6295 LUTs, and 5880 flip flops. Both the PS and the FPGA-based PL use a clock frequency of 100 MHz.

5.5.3 Performance Evaluation

The tool support described in Section 5.3.2 was used to transform the benchmarks. To avoid integrity violations due to the processor's prefetcher issuing reads outside the protected segments, the SCM memory region is expanded by inserting padding data plus correct MACs.

We used the following baremetal benchmarks. First, `qsort v1` [58] performs a Quicksort on 10,000 strings stored in read-only data. Second, `qsort v2` [58]

Table 5.1: Software benchmarks for SCM

Benchmark	DRAM (cycles)	SCM (cycles)	Reads	Overhead
`qsort v1`	40.35M	40.93M	40.03k	1.43%
`qsort v2`	36.21M	36.20M	18	-0.02%
`sjeng`	22.13G	22.13G	899.59k	-0.02%
`jpeg`	97.08M	97.11M	1462	-0.02%

performs a Quicksort on 10,000 strings stored in mutable data. Third, `sjeng` [61] plays a game of chess, and `jpeg` [58] performs jpeg encoding.

Each benchmark was executed from DRAM as well as from SCM memory. For the latter, both code (.text) and read-only data (.rodata) were mapped to SCM memory, and measurements were performed in unprotected code. The average overhead is shown in Table 5.1. The *"Reads"* column indicate the number of eight word burst reads that was performed by each benchmark. For `qsort v2`, `jpeg`, and `sjeng` we measured an overhead of -0.02%, which is below the noise margin of the performance counters that we used to perform the measurements, as determined by the standard deviations. For `qsort v1` we measured a significantly larger overhead of 1.43%, which is larger than the noise margin of the performance counters. These results show that SCM introduces only a small runtime overhead.

5.6 Conclusion

In this chapter we introduced a new hardware-based SI architecture, called SCM, that protects the integrity of code and read-only data stored in memory. SCM is realized as a standalone IP core that connects to the bus of a System-on-Chip (SoC). The protection mechanism performs a MAC-based integrity verification of code as it is fetched from external memory into the caches of a processor on an SoC. SCM is capable of protecting the code integrity against an attacker that is in control of both data and code memory. We demonstrated the feasibility of using such an architecture by evaluating the design on an FPGA, and our evaluation showed a minimal performance overhead when executing benchmark programs from the protected SCM memory region.

Chapter 6

Conclusions

In this chapter we provide a summary of the contributions of this thesis, together with possible future work.

6.1 Conclusions

In this thesis, we investigated security mechanisms to prevent software from misbehaving. To this end, we focused on hardware support to detect runtime attacks, prevent the illegal modification of software, and provide support for isolating software. This encompassed three fields of research, namely, Control Flow Integrity (CFI), which aims to detect illegal control flow modifications, Software Integrity (SI), which aims to prevent the execution of tampered software, and Protected Module Architectures (PMAs), which facilitate the secure execution of security-sensitive code in an area that is isolated from the rest of the system. The main contributions of this thesis are two-fold. We designed three new security mechanisms to detect and prevent runtime attacks, and further analysed existing hardware-based security mechanisms to detect attacks on control flow. In general, we typically consider the attacker capable of controlling data memory, while sometimes also controlling program memory. However, in some cases we even consider attackers with additional capabilities.

In Chapter 2 we presented a mechanism for handling interrupts for a program counter-based PMA that maintains the confidentiality of the protected module data. Our mechanism requires both software and hardware techniques. We explored the trade-off between performance and hardware area by making three

designs, each with a different hardware cost. Our FPGA prototype showed that interrupt support can be provided at a small cycle overhead, while incurring a minimal hardware area overhead.

In Chapter 3 we described and analysed 21 hardware-based CFI architectures in terms of their policies, security, and practical limitations. We found that all current CFI architectures are vulnerable to non-control data attacks, since current CFI architectures only ensure that the executed edges are valid, and do not ensure that the sequence of executed edges are correct. Most architectures relied upon a Shadow Call Stack (SCS) to protect backward edges, while using different mechanisms to protect forward edges. We found that SCSs provide practical and excellent protection against attacks on backward edges. However, we found that it is difficult to provide practical and strong protection for forward edges. A major problem is that high-security CFI, aka fine-grained CFI, relies on a CFG which is generated through static analysis. In practice it is difficult to generate a fully precise CFG on programs which contain calculated branches, and current software tools resort to over-approximation to solve this. Alternatively, low-security CFI, aka coarse-grained CFI, can provide practical protection which does not require static analysis. However, it enforces a much less strict policy that can not prevent all illegal branches. We therefore believe that CFI still has unsolved problems.

In Chapter 4 we presented SOFIA, which enforces both CFI and SI through processor modifications. SOFIA is the only known architecture to enforce CFI using instruction-set randomization. This is achieved by decrypting instructions with information derived from the control flow of a program, and verifying the integrity of instructions using a MAC. This enables SOFIA to protect software against runtime attacks, and is implemented as an extension to a processor's instruction pipeline. SOFIA is designed to be security-critical, since it prevents the execution of tampered instructions as well as instructions resulting from illegal control flow. It also protects against fault attacks on control flow, since only valid control flow can allow for the correct decryption of instructions. While SOFIA does rely on a CFG, precise static analysis is ensured by disallowing the use of indirect forward branches. In addition, launching a CRA is near impossible, since gadgets cannot be identified from the encrypted binary. In order to ensure that the software complies to the strict constraints imposed by the architecture, a software toolchain was developed. The toolchain was used to compile several benchmarks, and SOFIA was evaluated on an FPGA, which showed an average performance overhead of 106%, while incurring a hardware area increase of 12.9% LUTs, and a clock speed reduction of 23.2% when compared to an unmodified processor.

In Chapter 5 we presented a design called SCM, which is a light-weight alternative to SOFIA, with a reduced functionality. While SOFIA enforces

both CFI and SI through processor modifications, SCM enforces only SI and is implemented as an IP core. In fact, it is the first hardware-based SI architecture that is realised as a standalone IP core that connects to the bus via standard interfaces. This allows for the integration into an SoC without requiring changes to the used IP, such as the processor or memory hierarchy. SCM protects against code tampering by verifying the integrity of code as it is fetched from external memory into the caches of the SoC's processor. A limitation of only protecting the integrity of software at runtime (as is done in the current SCM), is that it cannot protect against CRAs. Our FPGA prototype showed a minimal performance overhead, while incurring a hardware area cost of 6295 LUTs and 5880 registers.

6.2 Future work

Control Flow Integrity

One of the most important limitations of CFI is that it cannot protect against non-control data attacks. A promising new line of work, called Control Flow Attestation [7, 40], aims to address this problem by recording the control flow paths that were taken inside an executed program, and reporting it to an external party for verification. However, the biggest limitation of this approach is that the security checks are not performed online, since an external party is required to verify the correctness of the measured control flow after the program has executed. Therefore, there is a need to develop security architectures that can solve this problem through online checks.

Providing strong and practical protection against attacks on forward edges cannot be done with the current available solutions. Fine-grained CFI aims to solve this, but it suffers from practical problems. It is well-known that a precise CFG cannot be computed for some types of programs. This is especially problematic for certain programming constructs that rely on indirect forward branches, such as function pointers passed between functions and C++ virtual methods. To solve this, over-approximation is used, which leads to a CFG that contains many more edges than what is strictly necessary. This is a fundamental limitation for fine-grained CFI policies, since the security of the enforced policy relies on the precision of the CFG.

As such, there is a strong need to develop security policies which do not require CFG. It is not yet clear how to achieve this. However, Code Pointer Integrity is a promising solution that could provide fine-grained protection, while at the same time not relying on a CFG.

Another direction is to follow an incremental approach, where a number of different CFI policies are designed to each defend against a specific attack. The idea is that the designs should be used simultaneously in order to create a strong defence against many different attacks. This will likely be an iterative process, which requires re-evaluating the system after each added policy in order to find the remaining exploitable attack vectors. For instance, we know that SCS provides an excellent defence against ROP. Furthermore, using SCS together with Branch Regulation (Section 3.6.8), which ensures that the semantics of basic blocks are followed, could make sense since it reduces the number of exploitable JOP gadgets. However, even when these two policies are used simultaneously, some attacks will not be detected, such as manipulating a pointer to execute an unwanted shared library function. To solve this, another security policy can be introduced to address the specific issue. However, it is likely that after fixing that problem, another attack vector will be discovered, which may require the design of yet another security policy to prevent it.

A common problem when evaluating a given CFI architecture is to determine the precise level of protection provided. To solve this, there is a need for a standardised testing methodology to evaluate the security attained by a given solution. This is non-trivial, since new attack vectors are frequently discovered. One solution could be to create a repository of code re-use attacks, and attacks that can circumvent the security of a specific CFI architecture. Ideally the project should be open-source to allow researchers to add newly discovered attack vectors to expand the knowledge of possible attacks, and further allow the evolution of defence mechanisms.

Low-latency block ciphers

Hardware security architectures often make use of cryptographic primitives, such as block ciphers, that are integrated into the pipeline stages of the processor. This allows the architectures to make security guarantees that would otherwise be impossible, such as preventing the execution of tampered instructions or preventing tampered control flow. A tight relationship is created between the crypto and the processor's instruction pipeline stages, which causes the efficiency of the crypto to severely impact the performance of the processor. For instance, the critical path of the crypto can limit the maximum clock frequency of the processor, or the processor might stall the processor's pipeline while waiting for the crypto to finish a computation. In the past, there have been only a few proposals for low-latency ciphers, such as PRINCE [18] and QARMA [14]. As such, there is a need for the further development of low-latency cryptographic primitives to enable the next generation of hardware supported security architectures.

SOFIA

SOFIA suffers from a large performance overhead due to increased code size, cycle overhead, and clock speed degradation. The increase in code size and cycle overhead is largely due to the strict set of constraints that are imposed on the software, leading to the insertion of a large number of padding instructions and multiplexer trees. In the future, the toolchain can be optimised to improve the cycle overhead by reducing the number of inserted and executed padding instructions. The current toolchain cannot handle optimisation passes, and can be improved by providing support for optimisation flags, such as `"-O2"`. Further improvements in cycle overhead could be obtained by optimising the length of the paths taken through the multiplexer trees.

SCM

The current implementation of SCM can only protect immutable data. In the future, the architecture can be extended by developing a light-weight mechanism to provide support for protecting mutable data. Furthermore, it could be valuable to experiment with an implementation of SCM that protects different memory regions in the system, e.g., ROM, flash, and on-chip memory. Yet another direction is to explore the necessary steps to allow SCM to operate alongside an OS with virtual memory support.

Secure Interrupts

The secure interrupts architecture is currently limited to protecting the confidentiality of the data stored inside the secure world. The architecture can be improved by developing protection against interrupt spoofing by ensuring that secure world interrupts can only be invoked by a legitimate interrupt. The availability of the system can also be improved by ensuring that a malicious interrupt service routine cannot hijack the control flow by executing code which never returns. Furthermore, the architecture can benefit from having support for multiple protection domains.

Bibliography

[1] Microsoft. Data Execution Prevention (DEP). `http://support.microsoft.com/kb/875352/EN-US/`, 2006.

[2] OpenCores project web site. `http://www.opencores.org/`, 2014. Accessed: 2014-02-20.

[3] BCC - Bare-C Cross-Compiler User's Manual. *Cobham* (2016).

[4] Pointer Authentication on ARMv8.3. *Qualcomm Technologies, Inc.* (2017).

[5] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-Flow Integrity. In *Proceedigns of the ACM Conference on Computer & Communications Security* (2005), ACM, pp. 340–353.

[6] Abadi, M., Budiu, M., Erlingsson, Ú., and Ligatti, J. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security 13*, 1 (2009).

[7] Abera, T., Asokan, N., Davi, L., Ekberg, J.-E., Nyman, T., Paverd, A., Sadeghi, A.-R., and Tsudik, G. C-flat: Control-flow attestation for embedded systems software. In *Proceedings of the Conference on Computer and Communications Security* (2016), CCS '16, ACM, pp. 743–754.

[8] Alexander, S. Defeating compiler-level buffer overflow protection. *;login issue: June 2005, Volume 30, Number 3*.

[9] Andreeva, E., Bogdanov, A., Luykx, A., Mennink, B., Tischhauser, E., and Yasuda, K. AES-COPA v.2. *CAESAR submission* (2015).

[10] ARM. *ARM Security Technology - Building a Secure System using TrustZone Technology*, 2009.

[11] ARM. ARM AMBA AXI and ACE Protocol Specification - AXI3, AXI4, and AXI4-Lite, ACE and ACE-Lite. White paper, 2011.

[12] Arora, D., Ravi, S., Raghunathan, A., and Jha, N. K. Hardware-assisted run-time monitoring for secure program execution on embedded processors. *IEEE Transactions on Very Large Scale Integrated Systems 14*, 12 (Dec. 2006), 1295–1308.

[13] Aumüller, C., Bier, P., Fischer, W., Hofreiter, P., and Seifert, J.-P. Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures. In *International Workshop on Cryptographic Hardware and Embedded Systems* (2003), CHES '02, Springer-Verlag, pp. 260–275.

[14] Avanzi, R. The QARMA block cipher family. almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes. *IACR Transactions on Symmetric Cryptology 2017*, 1 (2017), 4–44.

[15] Balasch, J., Gierlichs, B., and Verbauwhede, I. An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs. In *Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography* (2011), FDTC '11, IEEE Computer Society, pp. 105–114.

[16] Baratloo, A., Singh, N., and Tsai, T. Transparent run-time defense against stack smashing attacks. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (2000), ATEC '00, USENIX Association, pp. 21–21.

[17] Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. Jump-oriented programming: a new class of code-reuse attack. In *Symposium on Information, Computer & Communications Security* (2011), ACM, pp. 30–40.

[18] Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E. B., Knežević, M., Knudsen, L. R., Leander, G., Nikov, V., Paar, C., Rechberger, C., Rombouts, P., Thomsen, S. S., and Yalçın, T. PRINCE: A Low latency Block Cipher for Pervasive Computing Applications. In *Proceedings of the International Conference on The Theory and Application of Cryptology and Information Security* (2012), ASIACRYPT'12, Springer-Verlag, pp. 208–225.

[19] Brunel, J., Pacalet, R., Ouaarab, S., and Duc, G. SecBus, a Software/Hardware Architecture for Securing External Memories. In *Proceedings of the IEEE International Conference on Mobile Cloud*

Computing, Services, and Engineering (2014), MOBILECLOUD '14, IEEE Computer Society, pp. 277–282.

[20] Burow, N., Carr, S. A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., and Payer, M. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys 50*, 1 (Apr. 2017), 16:1–16:33.

[21] Carlini, N., Barresi, A., Payer, M., Wagner, D., and Gross, T. R. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the USENIX Security Symposium* (Aug. 2015), USENIX Association, pp. 161–176.

[22] Carlini, N., and Wagner, D. ROP is Still Dangerous: Breaking Modern Defenses. In *Proceedings of the USENIX Security Symposium* (2014), USENIX Association, pp. 385–399.

[23] Champagne, D., and Lee, R. Scalable architectural support for trusted software. In *International Symposium on High-Performance Computer Architecture* (2010), IEEE, pp. 1–12.

[24] Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., and Xie, L. Drop: Detecting return-oriented programming malicious code. In *Proceedings of the International Conference on Information Systems Security* (2009), ICISS '09, Springer-Verlag, pp. 163–177.

[25] Chen, S., Xu, J., Sezer, E. C., Gauriar, P., and Iyer, R. K. Non-control-data attacks are realistic threats. In *Proceedings of the USENIX Security Symposium* (2005), USENIX Association, pp. 177–191.

[26] Cheng, Y., Zhou, Z., Miao, Y., Ding, X., and Deng, H. R. ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014* (2014).

[27] Christoulakis, N., Christou, G., Athanasopoulos, E., and Ioannidis, S. HCFI: Hardware-enforced Control-Flow Integrity. In *Proceedings of the ACM Conference on Data and Application Security and Privacy* (2016), CODASPY '16, ACM, pp. 38–49.

[28] Conti, M., Crane, S., Davi, L., Franz, M., Larsen, P., Negro, M., Liebchen, C., Qunaibit, M., and Sadeghi, A.-R. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the Conference on Computer and Communications Security* (2015), CCS '15, ACM, pp. 952–963.

[29] Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., and Zhang, Q. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the USENIX Security Symposium* (1998), USENIX Association, pp. 5–5.

[30] Das, S., Zhang, W., and Liu, Y. A fine-grained control flow integrity approach against runtime memory attacks for embedded systems. *IEEE Transactions on Very Large Scale Integrated Systems 24*, 11 (Nov. 2016), 3193–3207.

[31] Davi, L., Hanreich, M., Paul, D., Sadeghi, A.-R., Koeberl, P., Sullivan, D., Arias, O., and Jin, Y. HAFIX: Hardware-assisted Flow Integrity Extension. In *Proceedings of the Design Automation Conference* (2015), DAC '15, ACM, pp. 74:1–74:6.

[32] Davi, L., Sadeghi, A.-R., Lehmann, D., and Monrose, F. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the USENIX Security Symposium* (Aug. 2014), USENIX Association, pp. 401–416.

[33] de Clercq, R., De Keulenaer, R., Coppens, B., Yang, B., Maene, P., de Bosschere, K., Preneel, B., de Sutter, B., and Verbauwhede, I. SOFIA: Software and Control Flow Integrity Architecture. In *Proceedings of the Conference on Design, Automation & Test in Europe* (2016), DATE '16, IEEE, pp. 1172–1177.

[34] de Clercq, R., de Keulenaer, R., Maene, P., Preneel, B., De Sutter, B., and Verbauwhede, I. SCM: Secure Code Memory Architecture. In *Proceedings of the ACM on Asia Conference on Computer and Communications Security* (2017), ASIACCS'17, ACM, pp. 771–776.

[35] de Clercq, R., Götzfried, J., David, U., Maene, P., and Verbauwhede, I. SOFIA: Software and Control Flow Integrity Architecture. In *Computers & Security* (2017), vol. 68, pp. 16–35.

[36] de Clercq, R., Roy, S. S., Vercauteren, F., and Verbauwhede, I. Efficient Software Implementation of ring-LWE Encryption. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition* (2015), DATE '15, ACM, pp. 339–344.

[37] De Clercq, R., Schellekens, D., Piessens, F., and Verbauwhede, I. Secure Interrupts on Low-End Microcontrollers. In *International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (2014), IEEE, pp. 147–152.

[38] de Clercq, R., Uhsadel, L., Van Herrewege, A., and Verbauwhede, I. Ultra Low-Power Implementation of ECC on the ARM Cortex-M0+. In *Proceedings of the Design Automation Conference* (2014), DAC '14, ACM, pp. 112:1–112:6.

[39] de Clercq, R., and Verbauwhede, I. A Survey of Hardware-based Control Flow Integrity. In *ArXiv CoRR* (2017), abs/1706.07257.

[40] Dessouky, G., Zeitouni, S., Nyman, T., Paverd, A., Davi, L., Koeberl, P., Asokan, N., and Sadeghi, A.-R. LO-FAT: Low-Overhead Control Flow ATtestation in Hardware. In *Proceedings of the Design Automation Conference* (2017), DAC '17, ACM, pp. 24:1–24:6.

[41] Dolev, D., and Yao, A. C. On the security of public key protocols. *IEEE Transactions on Information Theory 29*, 2 (1983), 198–208.

[42] Domingo-Ferrer, J. Software run-time protection: A cryptographic issue. In *Workshop on the Theory and Application of Cryptographic Techniques* (1990), Springer, pp. 474–480.

[43] Durumeric, Z., Kasten, J., Adrian, D., Halderman, J. A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M., and Paxson, V. The matter of heartbleed. In *Proceedings of the Conference on Internet Measurement Conference* (2014), IMC '14, ACM, pp. 475–488.

[44] Elbaz, R., Champagne, D., Gebotys, C., Lee, R., Potlapally, N., and Torres, L. Hardware mechanisms for memory authentication: A survey of existing techniques and engines. In *Transactions on Computational Science IV*. Springer, 2009, pp. 1–22.

[45] Elbaz, R., Torres, L., Sassatelli, G., Guillemin, P., Bardouillet, M., and Martinez, A. A parallelized way to provide data encryption and integrity checking on a processor-memory bus. In *Proceedings of the Design Automation Conference* (2006), ACM, pp. 506–509.

[46] Eldefrawy, K., Tsudik, G., Francillon, A., and Perito, D. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *Proceedings of the Network and Distributed System Security Symposium* (2012).

[47] Embedded Microprocessor Benchmark Consortium (EEMBC). Coremark. `http://www.eembc.org/coremark`, 2016. [Online; accessed 19-Sep-2016].

[48] Evans, I., Fingeret, S., Gonzalez, J., Otgonbaatar, U., Tang, T., Shrobe, H., Sidiroglou-Douskos, S., Rinard, M., and Okhravi, H. Missing the point(er): On the effectiveness of code pointer integrity. In *Proceedings of the IEEE Symposium on Security and Privacy* (2015), SP '15, IEEE Computer Society, pp. 781–796.

[49] Francillon, A., Nguyen, Q., Rasmussen, K. B., and Tsudik, G. A Minimalist Approach to Remote Attestation. In *Proceedings of the Conference on Design, Automation & Test in Europe* (2014), DATE '14, European Design and Automation Association, pp. 244:1–244:6.

[50] Francillon, A., Perito, D., and Castelluccia, C. Defending embedded systems against control flow attacks. In *Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code* (2009), SecuCode '09, ACM, pp. 19–26.

[51] Gaisler. Cobham Gaisler AB. LEON3 synthesizable processor. `http://www.gaisler.com`, 2015. [Online; accessed 26-Nov-2015].

[52] Göktaş, E., Athanasopoulos, E., Polychronakis, M., Bos, H., and Portokalidis, G. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the USENIX Security Symposium* (2014), USENIX Association, pp. 417–432.

[53] Goktas, E., Athanasopoulos, E., Bos, H., and Portokalidis, G. Out of control: Overcoming control-flow integrity. In *IEEE Security & Privacy* (2014), IEEE, pp. 575–589.

[54] Götzfried, J., Müller, T., de Clercq, R., Maene, P., Freiling, F., and Verbauwhede, I. Soteria: Offline Software Protection Within Low-cost Embedded Devices. In *Proceedings of the Annual Computer Security Applications Conference* (2015), ACSAC 2015, ACM, pp. 241–250.

[55] Gueron, S. A memory encryption engine suitable for general purpose processors. Cryptology ePrint Archive, Report 2016/204, 2016.

[56] Guo, Z., Bhakta, R., and Harris, I. G. Control-flow checking for intrusion detection via a real-time debug interface. In *2014 International Conference on Smart Computing Workshops* (Nov 2014), pp. 87–92.

[57] Gupta, A., Kerr, S., Kirkpatrick, M. S., and Bertino, E. Marlin: Making it harder to fish for gadgets. In *Proceedings of the ACM Conference on Computer and Communications Security* (2012), CCS '12, ACM, pp. 1016–1018.

[58] Guthaus R., M., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. MiBench: A free, commercially representative embedded benchmark suite, 2001.

[59] Handschuh, H., and Preneel, B. Minding your MAC algorithms. *Information Security Bulletin 9*, 6 (2004), 213–221.

[60] He, W., Das, S., Zhang, W., and Liu, Y. No-jump-into-basic-block: Enforce basic block cfi on the fly for real-world binaries. In *Proceedings of the Design Automation Conference 2017* (2017), DAC '17, ACM, pp. 23:1–23:6.

[61] Henning, J. L. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News 34*, 4 (2006), 1–17.

[62] Hu, H., Shinde, S., Adrian, S., Chua, Z. L., Saxena, P., and Liang, Z. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proceedings of the IEEE Symposium on Security and Privacy* (2016), IEEE, pp. 969–986.

[63] Intel. Intel Control-flow Enforcement Technology Preview. `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`, 2017. [Online; accessed 19-Jun-2017].

[64] International Organization for Standardization and International Electrotechnical Commission. Infomation technology - Secruity techniques - Message Authentication Codes (MACs). ISO/IEC 9797-1:1999(E), 1999.

[65] Kayaalp, M., Ozsoy, M., Abu-Ghazaleh, N., and Ponomarev, D. Branch regulation: Low-overhead protection from code reuse attacks. In *International Symposium on Computer Architecture* (2012), IEEE, pp. 94–105.

[66] Kayaalp, M., Ozsoy, M., Ghazaleh, N. A., and Ponomarev, D. Efficiently securing systems from code reuse attacks. *IEEE Transactions on Computers 63*, 5 (2014), 1144–1156.

[67] Kayaalp, M., Schmitt, T., Nomani, J., Ponomarev, D., and Abu-Ghazaleh, N. Scrap: Architecture for signature-based protection from code reuse attacks. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2013), HPCA '13, IEEE Computer Society, pp. 258–269.

[68] Kc, G. S., Keromytis, A. D., and Prevelakis, V. Countering Code-injection Attacks with Instruction-set Randomization. In *Proceedings of the Conference on Computer and Communications Security* (2003), CCS '03, ACM, pp. 272–280.

[69] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., Wilkerson, C., Lai, K., and Mutlu, O. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM SIGARCH Computer Architecture News* (2014), vol. 42, IEEE Press, pp. 361–372.

[70] Kinder, J., and Kravchenko, D. Alternating control flow reconstruction. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation* (2012), VMCAI'12, Springer-Verlag, pp. 267–282.

[71] Koeberl, P., Schulz, S., Sadeghi, A.-R., and Varadharajan, V. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the European Conference on Computer Systems* (2014), EuroSys '14, ACM, pp. 10:1–10:14.

[72] Kumar, R., Singhania, A., Castner, A., Kohler, E., and Srivastava, M. B. A System For Coarse Grained Memory Protection In Tiny Embedded Processors. In *Design Automation Conference* (2007), pp. 218–223.

[73] Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., and Song, D. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Oct. 2014), USENIX Association, pp. 147–163.

[74] Lee, J., Heo, I., Lee, Y., and Paek, Y. Efficient Security Monitoring with the Core Debug Interface in an Embedded Processor. *ACM Transactions on Design Automation of Electronic Systems 22*, 1 (May 2016), 8:1–8:29.

[75] Lee, Y., Heo, I., Hwang, D., Kim, K., and Paek, Y. Towards a Practical Solution to Detect Code Reuse Attacks on ARM Mobile Devices. In *Workshop on Hardware and Architectural Support for Security and Privacy* (2015), ACM, pp. 3:1–3:8.

[76] Lee, Y., Lee, J., Heo, I., Hwang, D., and Paek, Y. Integration of ROP/JOP Monitoring IPs in an ARM-based SoC. In *Proceedings of the Conference on Design, Automation & Test in Europe* (2016), DATE '16, IEEE, pp. 331–336.

[77] Lee, Y., Lee, J., Heo, I., Hwang, D., and Paek, Y. Using CoreSight PTM to Integrate CRA Monitoring IPs in an ARM-Based SoC. *ACM Transactions on Design Automation of Electronic Systems 22*, 3 (Apr. 2017), 52:1–52:25.

[78] Maene, P., Götzfried, J., de Clercq, R., Muller, T., Freiling, F., and Verbauwhede, I. Hardware-Based Trusted Computing Architectures for Isolation and Attestation. *IEEE Transactions on Computers PP(99)* (2017).

[79] Maene, P., and Verbauwhede, I. Single-cycle implementations of block ciphers. In *International Workshop on Lightweight Cryptography for Security and Privacy - Volume 9542* (2016), LightSec 2015, Springer-Verlag, pp. 131–147.

[80] Mahmood, A., and McCluskey, E. J. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers 37*, 2 (Feb. 1988), 160–174.

[81] Mao, S., and Wolf, T. Hardware support for secure processing in embedded systems. *IEEE Transactions on Computers 59*, 6 (2010), 847–854.

[82] Mashtizadeh, A. J., Bittau, A., Boneh, D., and Mazières, D. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the Conference on Computer and Communications Security* (2015), CCS '15, ACM, pp. 941–951.

[83] Masti, R. J., Marforio, C., Ranganathan, A., Francillon, A., and Capkun, S. Enabling trusted scheduling in embedded systems. In *Proceedings of the Annual Computer Security Applications Conference* (2012), ACM, pp. 61–70.

[84] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V., and Savagaonkar, U. R. Innovative instructions and software model for isolated execution. In *Proceedings of the International Workshop on Hardware and Architectural Support for Security and Privacy* (2013), HASP '13, ACM, pp. 10:1–10:1.

[85] Namprempre, C., Rogaway, P., and Shrimpton, T. Reconsidering generic composition. In *Advances in Cryptology–EUROCRYPT 2014*. Springer, 2014, pp. 257–274.

[86] Newsham, T. Format string attacks, 2000.

[87] Noorman, J., Agten, P., Daniels, W., Strackx, R., Herrewege, A. V., Huygens, C., Preneel, B., Verbauwhede, I., and Piessens, F. Sancus: Low-cost trustworthy extensible networked devices with a zero-software Trusted Computing Base. In *Proceedings of the USENIX Security Symposium* (2013), pp. 479–494.

[88] Ozdoganoglu, H., Vijaykumar, T. N., Brodley, C. E., Kuperman, B. A., and Jalote, A. Smashguard: A hardware solution to prevent security attacks on the function return address. *IEEE Transactions on Computers 55*, 10 (Oct. 2006), 1271–1285.

[89] Qiao, R., and Seaborn, M. A New Approach for Rowhammer Attacks. In *IEEE International Workshop on Hardware-Oriented Security and Trust, HOST'16* (2016), pp. 161—166.

[90] Rahmatian, M., Kooti, H., Harris, I. G., and Bozorgzadeh, E. Hardware-assisted detection of malicious software in embedded systems. *IEEE Embedded Systems Letters 4*, 4 (Dec. 2012), 94–97.

[91] Reparaz, O., de Clercq, R., Roy, S. S., Vercauteren, F., and Verbauwhede, I. Additively homomorphic ring-LWE masking. In *International Workshop on Post-Quantum Cryptography* (2016), Springer, pp. 233–244.

[92] Reparaz, O., Roy, S. S., de Clercq, R., Vercauteren, F., and Verbauwhede, I. Masking ring-LWE. *Journal of Cryptographic Engineering 6*, 2 (2016), 139–153.

[93] Salwan, J. ROPgadget. `http://shell-storm.org/project/ROPgadget`, 2017. [Online; accessed 16-Mar-2017].

[94] Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.-R., and Holz, T. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *Proceedings of the IEEE Symposium on Security and Privacy* (2015), SP '15, IEEE Computer Society, pp. 745–762.

[95] Shacham, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the ACM Conference on Computer & Communications Security* (2007), ACM, pp. 552–561.

[96] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security* (2004), CCS '04, ACM, pp. 298–307.

[97] SPARC International, Inc. The SPARC Architecture Manual, Version 8. `http://www.gaisler.com/doc/sparcv8.pdf`, 1991.

[98] Strackx, R., Piessens, F., and Preneel, B. Efficient Isolation of Trusted Subsystems in Embedded Systems. In *Security and Privacy in Communication Networks* (2010), pp. 344–361.

[99] Suh, G., Clarke, D., Gassend, B., Van Dijk, M., and Devadas, S. Aegis: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the International Conference on Supercomputing* (2003), ACM, pp. 160–171.

[100] Suh, G., Clarke, D., Gassend, B., van Dijk, M., and Devadas, S. Efficient memory integrity verification and encryption for secure processors. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture* (2003), MICRO 36, IEEE Computer Society, pp. 339–350.

[101] Sullivan, D., Arias, O., Davi, L., Larsen, P., Sadeghi, A.-R., and Jin, Y. Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity. In *Proceedings of the Design Automation Conference* (2016), DAC '16, ACM, pp. 163:1–163:6.

[102] Szekeres, L., Payer, M., Wei, T., and Song, D. Sok: Eternal war in memory. In *Proceedings of the IEEE Symposium on Security and Privacy* (2013), SP '13, IEEE Computer Society, pp. 48–62.

[103] Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., and Ning, P. On the expressiveness of return-into-libc attacks. In *Recent Advances in Intrusion Detection* (2011), Springer, pp. 121–141.

[104] Turan, F., De Clercq, R., Maene, P., Reparaz, O., and Verbauwhede, I. Hardware Acceleration of a Software-based VPN. In *International Conference on Field Programmable Logic and Applications (FPL)* (2016), IEEE, pp. 1–9.

[105] Turing, A. M. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society 2*, 42 (1936), 230–265.

[106] Van Bulck, J., Noorman, J., Mühlberg, J. T., and Piessens, F. Towards availability and real-time guarantees for protected module architectures. In *Companion Proceedings of the International Conference on Modularity* (2016), ACM, pp. 146–151.

[107] Verbauwhede, I., Karaklajic, D., and Schmidt, J.-M. The fault attack jungle - a classification model to guide you. In *Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography* (2011), FDTC '11, IEEE Computer Society, pp. 3–8.

[108] Wartell, R., Mohan, V., Hamlen, K. W., and Lin, Z. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the ACM Conference on Computer and Communications Security* (2012), CCS '12, ACM, pp. 157–168.

[109] Werner, M., Wenger, E., and Mangard, S. Protecting the control flow of embedded processors against fault attacks. In *International Conference on Smart Card Research and Advanced Applications - Volume 9514* (2016), CARDIS 2015, Springer-Verlag New York, Inc., pp. 161–176.

[110] Wilander, J., Nikiforakis, N., Younan, Y., Kamkar, M., and Joosen, W. RIPE: Runtime Intrusion Prevention Evaluator. In *Proceedings of the Annual Computer Security Applications Conference* (2011), ACSAC '11, ACM, pp. 41–50.

[111] Wilken, K., and Shen, J. P. Continuous signature monitoring: efficient concurrent-detection of processor control errors. In *International Test Conference 1988 Proceeding* (Sep 1988), pp. 914–925.

[112] Williams, P., and Boivie, R. CPU support for secure executables. In *International Conference on Trust and Trustworthy Computing* (2011), Springer, pp. 172–187.

[113] Yan, C., Englender, D., Prvulovic, M., Rogers, B., and Solihin, Y. Improving cost, performance, and security of memory encryption and authentication. In *ACM SIGARCH Computer Architecture News* (2006), vol. 34, IEEE Computer Society, pp. 179–190.

[114] Zhang, W., Bao, Z., Lin, D., Rijmen, V., Yang, B., and Verbauwhede, I. RECTANGLE: a bit-slice lightweight block cipher suitable for multiple platforms. *Science China Information Sciences 58*, 12 (2015), 1–15.

Curriculum Vitae

Ruan de Clercq was born on September, 11 1981 in Pretoria, South Africa. He obtained a Bachelor's degree in Computer Engineering from the University of Pretoria, South Africa in 2010, and a Master's degree in Electrical Engineering from the KU Leuven, Belgium in 2013. He was generously funded by an Erasmus Mundus Action 2 scholarship during his Master's degree studies.

In September 2013, he joined the COSIC (Computer Security and Industrial Cryptography) research group as a research assistant. Prior to this, he spent some time working as a software engineer at Saab Systems Grintek as well as LQS International Ltd. in South Africa.

List of publications

Journals

1. de Clercq, R., Götzfried, J., David, U., Maene, P., and Verbauwhede, I. SOFIA: Software and Control Flow Integrity Architecture. In *Computers & Security* (2017), vol. 68, pp. 16–35

2. Maene, P., Götzfried, J., de Clercq, R., Muller, T., Freiling, F., and Verbauwhede, I. Hardware-Based Trusted Computing Architectures for Isolation and Attestation. *IEEE Transactions on Computers PP(99)* (2017)

3. Reparaz, O., Roy, S. S., de Clercq, R., Vercauteren, F., and Verbauwhede, I. Masking ring-LWE. *Journal of Cryptographic Engineering 6*, 2 (2016), 139–153

International Conferences

1. de Clercq, R., de Keulenaer, R., Maene, P., Preneel, B., De Sutter, B., and Verbauwhede, I. SCM: Secure Code Memory Architecture. In *Proceedings of the ACM on Asia Conference on Computer and Communications Security* (2017), ASIACCS'17, ACM, pp. 771–776

2. Turan, F., De Clercq, R., Maene, P., Reparaz, O., and Verbauwhede, I. Hardware Acceleration of a Software-based VPN. In *International Conference on Field Programmable Logic and Applications (FPL)* (2016), IEEE, pp. 1–9

3. de Clercq, R., De Keulenaer, R., Coppens, B., Yang, B., Maene, P., de Bosschere, K., Preneel, B., de Sutter, B., and Verbauwhede, I. SOFIA: Software and Control Flow Integrity

Architecture. In *Proceedings of the Conference on Design, Automation & Test in Europe* (2016), DATE '16, IEEE, pp. 1172–1177

4. Reparaz, O., de Clercq, R., Roy, S. S., Vercauteren, F., and Verbauwhede, I. Additively homomorphic ring-LWE masking. In *International Workshop on Post-Quantum Cryptography* (2016), Springer, pp. 233–244

5. Götzfried, J., Müller, T., de Clercq, R., Maene, P., Freiling, F., and Verbauwhede, I. Soteria: Offline Software Protection Within Low-cost Embedded Devices. In *Proceedings of the Annual Computer Security Applications Conference* (2015), ACSAC 2015, ACM, pp. 241–250

6. de Clercq, R., Roy, S. S., Vercauteren, F., and Verbauwhede, I. Efficient Software Implementation of ring-LWE Encryption. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition* (2015), DATE '15, ACM, pp. 339–344

7. De Clercq, R., Schellekens, D., Piessens, F., and Verbauwhede, I. Secure Interrupts on Low-End Microcontrollers. In *International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (2014), IEEE, pp. 147–152

8. de Clercq, R., Uhsadel, L., Van Herrewege, A., and Verbauwhede, I. Ultra Low-Power Implementation of ECC on the ARM Cortex-M0+. In *Proceedings of the Design Automation Conference* (2014), DAC '14, ACM, pp. 112:1–112:6

Unpublished Manuscripts

1. de Clercq, R., and Verbauwhede, I. A Survey of Hardware-based Control Flow Integrity. In *ArXiv CoRR* (2017), abs/1706.07257

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF ELECTRICAL ENGINEERING
COSIC
Kasteelpark Arenberg 10, bus 2452
B-3001 Leuven
ruan.declercq@esat.kuleuven.be
http://www.esat.kuleuven.be/cosic/