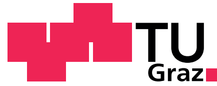


Robert Schilling

Hardware Extensions and Compiler Support for Protection Against Fault Attacks

DOCTORAL THESIS
to achieve the university degree of
Doctor of Technical Sciences; Dr. techn.

submitted to
Graz University of Technology

Assessors

Prof. Stefan Mangard
Graz University of Technology, Austria

Prof. Ingrid Verbauwhede
Katholieke Universiteit Leuven, Belgium

Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology
Graz, September 2023

Don't adventures ever have an end? I suppose not.
Someone else always has to carry on the story.

BILBO BAGGINS, BY J.R.R. TOLKIEN

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used.

Abstract

Software is ubiquitous in all applications of modern lives. The correct execution of software is essential for the functionality and security of the underlying device. Fault attacks undermine the correct software execution and therefore break the security of many devices. Attackers with physical proximity can induce faults and disrupt the correct operation. With recent research results, faults can even be induced via software remotely, bringing those attacks to even larger systems. Unfortunately, the current processors do not offer considerable support in safeguarding software execution from fault attacks.

In this thesis, we tackle the challenge of secure software execution in the presence of fault attacks from three perspectives. First, we look into this problem from an algorithmic and energy efficiency point of view. We analyze a hardware accelerator for fault- and side-channel secure cryptography. Instead of deploying redundancy-based protection against faults, we explore fresh re-keying as an algorithmic way of protection. The accelerator is deeply integrated into a multi-core System-on-Chip (SoC) and provides fault-protected (authenticated) encryption with an energy budget of a few pJ/op. Then, we shift the perspective and analyze how a compiler-assisted approach can be used to protect arbitrary software against faults. We show that existing hardware primitives from the ARM Instruction Set Architecture (ISA) can be used to build a Control-Flow Integrity (CFI) protection scheme against software- and fault-based control-flow attacks. By developing FIPAC, a basic block granular CFI protection scheme with full compiler support, we can protect software with different security guarantees against control-flow attacks. FIPAC is further the pillar to protect the system call interface of an operating system against faults. Eventually, we solve the challenge of unprotected conditional branches in the presence of faults, even with deployed CFI. By designing a protected comparison that is linked to the CFI scheme, we inherently provide protection for security-critical conditional branches. Finally, we investigate the protection of memory accesses in the physical and virtual memory domain against fault attacks. With reasonable hardware and runtime overheads, we protect the memory subsystem of embedded- and application-class processors against faults. These mechanisms use the help of the compiler and can be automatically applied to arbitrary software. These developments show that a compiler-assisted hardware-software code design can lead to efficient and secure countermeasures for different performance profiles.

Acknowledgements

I would like to express my heartfelt gratitude to all the individuals who have helped and supported me throughout my PhD journey.

First and foremost, I would like to thank my advisor Stefan Mangard for his invaluable guidance, support, and encouragement. His insights, expertise, and patience have been instrumental in shaping my research and helping me grow as a scientist. I am truly grateful for all his efforts and for believing in me. I would also like to thank Ingrid Verbauwhede for agreeing to asses my thesis and for her valuable feedback and suggestions.

I would like to thank all my incredible co-authors who either gave me the chance to collaborate with them on their research or contributed actively to mine. In particular, thank you to Luca Benini, Francesco Conti, Michael Gautschi, Germain Haugou, Frank Gürkaynak, Igor Loi, Michael Muehlberghuber, Pascal Nasahl, Stefan Mangard, Antonio Pullini, Davide Rossi, Pasquale Davide Schiavone, Martin Unterguggenberger, Thomas Unterluggauer, Stefan Weiglhofer, and Mario Werner with whom I wrote the papers discussed in this thesis. Moreover, I am thankful to work with Roderick Bloem, Claudio Canella, Daniel Gruss, Jan Hoogerbrugg, Manuel Jelinek, Florian Kargl, Anja Karl, Thomas Korak, Moritz Lipp, Marcel Medwed, Rishub Nagpal, Markus Ortoff, David Schaffenrath, and David Schrammel on exciting papers and discussing new ideas. Thank you all for all the interesting discussions and talks, input, and all your work. Without all of you, my journey through the PhD would not have been the same.

I am deeply grateful to my friends at IAIK for their interesting discussions in the kitchen, on a couch, or elsewhere. Especially, I am very grateful to Mario Werner, Pascal Nasahl, and Martin Unterguggenberger. Mario, thank you for motivating me during my PhD studies and for all the great collaborations we had, for all the Kepabs at Hungerstopper, and for all the beer. Thank you Pascal, for the fruitful discussions, the amazing time in our office, and an and open ear for all problems. Thank you Martin, for giving me a reason to buy strawberry juice again.

I am also grateful to my family and friends for their love and support throughout my PhD studies. Their encouragement and enthusiasm have kept me motivated and inspired. I would also like to extend my special thanks and love to my fiancée, Anna, for her unwavering love, support, and patience. She has been my constant source of strength and inspiration throughout my PhD journey, and I am grateful to have her by my side. Thank you, Anna, for always being there for me. I love you.

Robert

Table of Contents

Affidavit ii

Abstract iii

Acknowledgements iv

Table of Contents v

List of Tables x

List of Figures xi

Glossary xiv

1 Introduction 1
Problem Statement . . . 3
Contribution and Outline . . . 4

2 Fault Attacks and Countermeasures 8
2.1 Physical Fault Induction Techniques . . . 8
2.2 Software-Induced Fault Attacks . . . 9
2.3 Fault Models . . . 10
2.4 Fault Exploitation . . . 11
2.4.1 Fault Attacks on Cryptographic Implementations . . . 12
2.4.2 Exploiting Faults on Non-Cryptographic Software . . . 12
2.5 Countermeasures Against Fault Attacks . . . 13
2.5.1 Duplication-Based Countermeasures . . . 15
2.5.2 Error Detection Codes . . . 16
2.5.3 Fresh Re-keying to Counteract Fault Attacks . . . 20

3 Control-Flow, Control-Flow Attacks, and Control-Flow Integrity 23
3.1 Control-Flow Attacks . . . 24
3.1.1 Software-Based Control-Flow Attacks . . . 24
3.1.2 Fault-Based Control-Flow Attacks . . . 24
3.2 Control-Flow Integrity . . . 25
3.2.1 Software CFI Protection Schemes . . . 25

3.2.2 Fault CFI Protection Schemes 25

4 Energy-Efficient Encryption with Algorithmic Fault Protection of IoT End-Nodes 27
4.1 Background 30
4.1.1 Threat Model 30
4.1.2 Energy and Security Requirements of IoT End-Nodes . . 31
4.1.3 Leakage-Resilient Encryption with Re-Keying and a 2PRG 32
4.1.4 Re-Keying Function 33
4.2 Isap - Lightweight Authenticated Encryption 33
4.3 SoC Architecture 35
4.4 Cluster-Coupled Accelerator Engines 38
4.4.1 Hardware Encryption Engine 40
4.5 Experimental Evaluation 42
4.5.1 System-on-Chip Operating Modes 42
4.5.2 HWCRYPT Performance and Power Evaluation 44
4.5.3 Comparison with State-of-the-Art 45
4.6 Use Cases 47
4.6.1 Secure Autonomous Aerial Surveillance 48
4.6.2 Local Face Detection with Secured Remote Recognition . 50
4.6.3 Seizure Detection and Secure Long-Term Monitoring . . . 50
4.7 State-of-the-Art and Related Work 51
4.7.1 Low-Power Encryption Hardware Intellectual Property (IP) 51
4.7.2 Internet-of-Things (IoT) End-Node Architectures 52
4.8 Conclusion 53

5 FIPAC: Control-Flow Protection with ARM Pointer Authentication 55
5.1 ARM Pointer Authentication 57
5.2 Threat Model and Attack Scenario 58
5.2.1 Threat Model 58
5.2.2 Attack Scenario 59
5.2.3 CFI against Software- and Fault-Based Control-Flow Attacks 61
5.3 Design of FIPAC 62
5.3.1 Signature-Based Control-Flow Integrity 63
5.3.2 State Updates with ARM Pointer Authentication 65
5.3.3 Placement of Checks 66
5.4 Implementation 66
5.4.1 System Implementation 67
5.4.2 CFI Primitives 67
5.4.3 Protection of Control-Flow Instructions 69
5.4.4 Toolchain 70
5.5 Evaluation 72
5.5.1 Security Evaluation 72
5.5.2 Security Comparison 75

5.5.3 Functional Evaluation 77
5.5.4 Performance Evaluation 77
5.6 Example Exploits 81
5.7 Data Protection with FIPAC 83
5.8 Discussion 84
5.8.1 FIPAC Hardware Requirements 84
5.8.2 FIPAC on ARMv8.3-A 85
5.8.3 FIPAC on Other Architectures 85
5.8.4 Dynamic Key Handling 86
5.8.5 Instruction Granular Protection 86
5.8.6 Compatibility 86
5.9 Conclusion 87

6 System Call Flow Protection and Dynamic CFI 88
6.1 Background 90
6.1.1 Linux and the System Call Interface 90
6.2 Threat Model and Attack Scenario 91
6.2.1 Attack Scenario 91
6.2.2 FIPAC Intra-Basic Block Protection 93
6.3 Design of SFP 93
6.3.1 Requirements for System Call Protection 93
6.3.2 System Call Protection 94
6.3.3 Dynamic Instrumentation 95
6.4 Implementation 97
6.4.1 Toolchain 97
6.4.2 Kernel Support 98
6.5 Evaluation 99
6.5.1 Security Evaluation 99
6.5.2 Functional Evaluation 101
6.5.3 Performance Evaluation 101
6.6 Discussion 103
6.6.1 Dynamic System Call Instrumentation 103
6.6.2 CFI Checking Policy Extension 103
6.6.3 Compatibility 103
6.7 Related Work 104
6.8 Conclusion 105

7 Secure Comparisons and Conditional Branches for CFI 106
7.1 Threat Model and Related Work 108
7.1.1 Threat Model 108
7.1.2 Conditional Branch Protection via Re-checking 109
7.1.3 Conditional Branches in the Context of CFI 109
7.2 Protecting Conditional Branches against Fault Attacks 109
7.2.1 Requirements for CFI Protection Scheme 111
7.3 Protected Comparisons with AN-Codes 111
7.3.1 Protected Equal and Not-Equal Condition Computation . 113

7.3.2 Parameter Selection 114
7.4 Implementation and Evaluation 115
7.4.1 Implementation 115
7.4.2 Cost Analysis 115
7.4.3 Performance Evaluation 117
7.5 Security Analysis 118
7.6 Compatibility 118
7.6.1 Compatibility with FIPAC 118
7.6.2 Compatibility with Sponge-Based Control-Flow Protection (SCFP) 118
7.7 Conclusion 119

8 Secure Memory Accesses in the Presence of Fault Attacks 121
8.1 Background of Memory Access 123
8.1.1 ANB-Codes for Memory Access Protection 124
8.1.2 ARM Pointer Authentication 124
8.2 Threat Model and Attack Scenario 124
8.2.1 Threat Model 124
8.2.2 Attack Scenario 125
8.3 Pointer Protection with Residue Codes 126
8.3.1 Overview 126
8.3.2 Pointer Layout and Residue-Code Selection 127
8.3.3 Pointer Operations 128
8.4 Evolved Memory Access Protection 129
8.4.1 Overview 129
8.4.2 The Linking Approach 130
8.4.3 Memory-Mapped I/O 132
8.5 Architecture 133
8.5.1 New Instructions 133
8.5.2 Hardware 134
8.5.3 Software 135
8.6 Evaluation 137
8.6.1 Future Work 138
8.7 Conclusion 138

9 Protected Memory Accesses in the Virtual Memory Domain 140
9.1 Page-based Virtual Memory 142
9.2 Threat Model and Attack Scenario 143
9.2.1 Threat Model 143
9.2.2 Faults on Virtual Memory 143
9.2.3 Requirements for Protected Virtual Memory Accesses . . 144
9.3 Design of SecWalk 145
9.3.1 Protected Pointers and Memory Accesses 145
9.3.2 Secure Page Table Walk 146
9.3.3 Translation Look-Aside Buffer (TLB) Design 150
9.3.4 Page Directory Setup 150

9.3.5 Shared Memory Support 151
9.4 Implementation 151
9.4.1 Hardware Implementation 152
9.4.2 Toolchain Implementation 154
9.5 Evaluation 154
9.5.1 Hardware Evaluation 154
9.5.2 Performance Evaluation 154
9.5.3 Security Evaluation 156
9.6 Related Work 156
9.7 Conclusion 158

10 Conclusion and Outlook **159**
Outlook 160

List of Contributions **163**
10.1 Main Publications 163
10.2 Contributed Publications 164

Bibliography **167**

List of Tables

4.1 Comparison between *Fulmine* and several platforms of the state-of-the-art in encryption, data analytics, and IoT end-nodes. . . 43

5.1 Protection guarantees and vulnerabilities for Software CFI (SCFI) and Fault CFI (FCFI) protection schemes compared to FIPAC. . 76
5.2 Code size overhead for SPECspeed 2017. 79
5.3 Code size overhead for Embench. 80

7.1 Condition values for encoded $<, \leq, >, \geq$ condition values. 114
7.2 Qualitative overhead analysis of the building blocks. 116
7.3 Size and runtime overhead of different branch protections. 117

8.1 Code and runtime overhead for different benchmark programs from an Hardware Description Language (HDL) simulation. . . . 137

9.1 Hardware utilization of SecWalk. 155
9.2 Feature comparison of SecWalk compared to related work. 157

List of Figures

2.1 Re-keying g function combined with an encryption/decryption primitive. 21

3.1 Control-Flow Graph (CFG) showing basic blocks with ARMv8 instructions. 24

4.1 2PRG-based leakage-resilient stream cipher. 32
4.2 Re-keying function based on a polynomial multiplication with a block-cipher-based feed-forward computation. 33
4.3 IsapEnc with initialization. 34
4.4 IsapRk. 34
4.5 IsapMac. 35
4.6 *Fulmine* SoC architecture. The soc domain is shown in shades of blue, the cluster domain in shades of green. 36
4.7 *Fulmine* power management architecture and power modes. . . . 38
4.8 HWCRYPT datapath overview, with details of the *AES Engine* and the *Sponge Engine*. 39
4.9 *Fulmine* chip microphotograph with main components highlighted. 44
4.10 Cluster maximum operating frequency and power in the cry-cnn-sw, kec-cnn-sw, and sw operating modes. Each set of power bars, from left to right, indicates activity in a different subset of the cluster. kec-cnn-sw and sw bars show the additional power overhead from running at the higher frequency allowed by these modes. 45
4.11 Performance and efficiency of the HWCRYPT accelerator in terms of time/energy for elementary output. 46
4.12 A *Fulmine* SoC connected to 16 MB of Flash, 2 MB of Ferroelectric RAM (FRAM), and sensors (the grey area is taken into account for power estimations). 47
4.13 Secure autonomous aerial surveillance use case based on a *ResNet-20* Convolutional Neural Networks (CNN) [He+16] with AES-2PRG encryption for all weights and partial results. kec-cnn-sw and cry-cnn-sw operating modes at V_{DD} =0.8 V. 48

4.14 Local face detection, secured remote recognition use case based on the 12-net and 24-net CNNs from Li et al. [Li+15] on a 224×224 input image, with full ISAP encryption of the image if a potential face is detected. CRY-CNN-SW operating mode at V_{DD} =0.8 V. We consider that the first stage 12-net classifies 10 % of the input image as containing faces, and that the second stage 24-net is applied only to that fraction. 49
4.15 Electroencephalogram (EEG)-based seizure detection and secure data collection. CRY-CNN-SW operating mode at V_{DD} =0.8 V. . . 51

5.1 Valid control-flow. 60
5.2 Detectable control-flow attack. 61
5.3 Successful control-flow attack. 61
5.4 Justifying signature for control-flow merges. 63
5.5 CFI state patch for direct calls. 64
5.6 CFI state patch for indirect calls. 65
5.7 Custom toolchain to build protected binaries. 71
5.8 Control-flow hijack from `B` to `C`. Due to a state collision, the control-flow hijack is not detected. 74
5.9 A coarse-grained check policy. After n updates, a collision rectifies the faulty state. 74
5.10 Collision probability after N state updates. 75
5.11 Number of functions with N basic blocks. 76
5.12 Runtime overhead for SPECspeed 2017. 78
5.13 Runtime overhead for Embench. 79

6.1 Linux system call invocation. 91
6.2 Redirecting a system call using fault attacks. 92
6.3 System Call protection in SFP. Before a syscall, we cryptographically bind the syscall to the CFI state for later verification and second-stage linking in the kernel. 96
6.4 The microbenchmark shows the system call latency of the `getpid` system call for different kernel configurations. SFP increases the system call latency by 1.9 %. 102
6.5 Macrobenchmark shows the performance impact of SFP on the SPEC 2017 benchmark. We evaluate the impact of CFI only and SFP, including the system call protection. 103

7.1 Conditional branch with CFI state. 110
7.2 Protected conditional branch with state update and n-bit redundantly encoded comparison. 111
7.3 Modified LLVM compiler pipeline. Grey boxes indicate modifications or additions of/to the regular compilation flow. 116

8.1 Attack vector: Modified pointers and manipulated memory accesses. 125

8.2 Encoded pointer representation. The actual 40-bit pointer value p, the MMIO tag bit, and 23 bits of redundancy r_p comprise an encoded 64-bit pointer. 128
8.3 Byte-wise data linking of a 64-bit word. Each byte gets XORed with its respective XOR-reduced encoded address. 132
8.4 Modified processor pipeline. The instruction decode stage is extended with a 12-bit residue encoder, the execution stage with a residue Arithmetic Logic Unit (ALU), and the write-back stage with a pointer-reduction data-linking unit. 135
8.5 Residue ALU with a 41-bit adder and a shared residue encoder. The addition result is automatically checked after the operation by re-encoding the result and comparing it with the computed residues and generating a redundant error signal. 136

9.1 Attack vector: A faulted page table translation leads to a wrong memory access. 144
9.2 Encoded virtual address in Sv39. The upper 25-bits denote the redundancy information of the multi-residue code. 146
9.3 Sv39 page table entry with the extended encoded Physical Page number (PPN) to store the redundancy information. 147
9.4 Secure page table walk with linked page table entries. 148
9.5 CVA6 hardware architecture with SecWalk. The yellow parts indicate changes in the design. 151
9.6 Hardware architecture the load-store-unit of CVA6. The yellow parts indicate changes in the design. 152
9.7 Residue page table walker exploiting the redundancy properties of residue codes. 153
9.8 Performance evaluation using microbenchmarks. 155

Glossary

AES-GCM	AES in the Galois Counter Mode
ALU	Arithmetic Logic Unit
AMD-SP	AMD Secure Processor
ARM PA	ARM Pointer Authentication
ASIC	Application Specific Integrated Circuit
ASIL	Automotive Safety Integrity Level
ASLR	Address Space Layout Randomization
BMC	Baseboard Management Controller
CFG	Control-Flow Graph
CFI	Control-Flow Integrity
CNN	Convolutional Neural Networks
cpb	cycles per byte
CPU	Central Processing Unit
CSR	Control and Status Register
DAG	Directed Acyclic Graph
DEMUX	Demultiplexers
DFA	Differential Fault Analysis
DFIA	Differential Fault Intensity Analysis
DMA	Direct Memory Access
DOP	Data-Oriented Programming
DPA	Differential Power Analysis
DVFS	Dynamic Voltage and Frequency Scaling
DWT	Digital Wavelet Transform
ECB	Electronic-Code-Book
ECG	Electrocardiogram
ECU	Electronic Control Unit
EDC	Error Detection Code
EEG	Electroencephalogram
ELF	Executable and Linkable Format

EMG	Electromyogram
FCFI	Fault CFI
FLL	Frequency-Locked Loop
FPGA	Field Programmable Gate Array
FRAM	Ferroelectric RAM
FSA	Fault Sensitivity Analysis
GPSA	Generalized Path Signature Analysis
HDL	Hardware Description Language
HWCRYPT	Hardware Encryption Engine
IoT	Internet-of-Things
IP	Intellectual Property
IR	Intermediate Representation
ISA	Instruction Set Architecture
JIT	Just-in-Time
JOP	Jump-Oriented Programming
LSU	Load-and-Store Unit
LUT	Lookup Table
MAC	Message Authentication Code
MMIO	Memory-Mapped I/O
MMU	Memory-Management Unit
MPSoC	Multi-Processor SoC
OS	Operating System
PAC	Pointer Authentication Code
PC	Program Counter
PCA	Principal Components Analysis
PLC	Programmable Logic Controller
PMP	Physical Memory Protection
PO	Page Offset
PPN	Physical Page number
PTE	Page Table Entry
PTW	Page Table Walker
RCE	Remote Code Execution

ResPTW	Residue Page Table Walker
ROP	Return-Oriented Programming
SCFI	Software CFI
SCFP	Sponge-Based Control-Flow Protection
SCM	Standard Cell Memory
SECDED	Single-Error Correction and Double-Error Detection
SEV	Secure Encrypted Virtualization
SFA	Statistical Fault attacks
SFCFI	Software-Fault CFI
SIFA	Statistical Ineffective Fault Attacks
SIMD	Single Instruction Multiple Data
SoC	System-on-Chip
SPA	Simple Power Analysis
SVM	Support Vector Machine
TCDM	Tightly-Coupled Data Memory
TEE	Trusted Execution Environment
TLB	Translation Look-Aside Buffer
TMR	Triple Modular Redundancy
UAV	Unmanned Aerial Vehicle
ULP	Ultra-Low Power
VPN	Virtual Page Number
XEX	XOR–Encrypt–XOR
XTS	XEX-based Tweaked-Codebook mode with Ciphertext Stealing

1 Introduction

In the last few decades, there has been a massive transformation in the landscape and scale of computing devices. The number of applications as well as the number of processing devices massively increased to hundreds of billions of processing cores [ARM21]. In the past, computers or other processing systems were used only for specific applications in specific domains. However, today's landscape of computers is vastly different, as they have become ubiquitous and present in almost every aspect of our lives. The appearance and emergence of the Internet-of-Things (IoT), which connects a large number of different devices and applications, increases this transformation. Traditionally, devices of the IoT simply run bare-metal applications, but as embedded systems became more sophisticated, small real-time operating systems were deployed on them. Nowadays, with all the advances in technology, these devices often are based on powerful application-class processors that run off-the-shelf operating systems. These evolutions also changed the landscape of commodity devices. In addition to personal computers such as desktops and laptops, we now rely heavily on powerful smartphones, tablets, and even cloud-based systems on a daily basis. Consequently, our modern lives are entangled with computers in various forms and form factors. Independent of the device's form factor, performance, or concrete application, they all contain or process a wide array of valuable assets that require varying degrees of protection.

The proliferation of computing systems, coupled with their abundance of valuable assets, has a notable consequence – the significant expansion of the attack surface. This expanded attack surface provides malicious actors with numerous opportunities to launch various attacks, whether extracting sensitive information, gaining unauthorized access to systems, or achieving other malicious objectives. One common approach employed by attackers to perform an attack is to exploit different software vulnerabilities, which can have different root causes.

For example, a bug in the application logic or a lack of proper access control in the software can pave the way for unauthorized access to highly sensitive data. However, software attacks can also occur at a lower level of abstraction. One example is a memory corruption vulnerability on the heap memory, which can serve as the entry point of a full exploit. Memory corruption can be used to craft a control-flow hijack, such as Return-Oriented Programming (ROP) [Sha07] or Jump-Oriented Programming (JOP) [Che+10]. This attack methodology is extremely powerful and can even lead to arbitrary code execution of the attacked system.

Although software-based attacks pose a significant threat to the system's security, it is crucial to recognize that not all hijacks are solely based on exploiting software bugs. There exists another class of attacks known as physical attacks, which exploit the physical aspects of the system to compromise its security. Physical attacks can further be classified into passive and active attacks. Passive attacks, such as side-channel analysis, involve monitoring the physical characteristics of a system during its operation to extract sensitive information. These attacks capitalize on the unintended side effects of the system's physical implementation, such as power consumption, electromagnetic emissions, or timing variations. By carefully analyzing this side-channel information, an attacker can deduce cryptographic keys, passwords, or confidential data without directly compromising the hardware, software, or underlying algorithms. On the other hand, active attacks exploit physical effects to manipulate or disrupt the behavior of the system directly. One such example is fault attacks, where an attacker deliberately induces faults or errors in the system to subvert its regular operation.

During a fault attack, the attacker actively influences the device's operating conditions with the goal of manipulating an inner state of the system. Typically, the fault attacks manipulate the operating conditions out of the device's specifications, leading to unintended behavior. At this operating point, the correct execution of all operations is not guaranteed anymore, which possibly breaks the security measures of the system. Classic fault injection attacks exploit the power supply [Bar+09; BFP19; BS03], the clock signal [Bar+06; PQ03; RSG21], the temperature [HS13], or by shooting with a laser or electromagnetic impulse onto the chip surface [BS97; Mor+13; SA02; Sel+15; WWM11; ZAV04]. The effort here depends on the methodology used, but dedicated setups to perform fault injections can be fairly cheap [New23]. Especially in the context of processor architectures, fault attacks have shown to be an effective method to consistently skip or repeat instructions [KH14; KSV13; SH08]. While these fault methodologies require physical access to the device, more recent attacks have shown that this constraint can be relaxed. Software-induced fault attacks make faults a severe threat to a wide range of devices. For example, the Rowhammer effect [Kim+14] can be used to manipulate bits in memory. This behavior can even be exploited remotely via Javascript [GMM16] or over the network interface [Lip+20; Tat+18]. Recent fault attacks, such as Plundervolt [Mur+20], VoltJockey [Qiu+19a; Qiu+19b], or CLKscrew [TSS17], show the impact of fault attacks on commodity devices.

Both passive and active physical attacks showcase the importance of considering the physical aspects of a system's security. While software attacks are well-known and widely addressed with different protection mechanisms, physical attacks demonstrate that they require dedicated countermeasures. Implementing countermeasures against side-channel analysis or ensuring fault tolerance is essential to mitigate the risk of physical attacks and safeguard the integrity and confidentiality of computing systems.

In order to defend against fault attacks, it is necessary for critical applications to implement targeted countermeasures. Historically, much of the research on protecting software against fault attacks focused on securing cryptographic algorithms. As a result, specialized protection schemes have been developed and deployed to secure specific cryptographic schemes. While these countermeasures effectively protect cryptographic functions, other parts of the software are still vulnerable to fault attacks. This is especially true for non-cryptographic elements of the software, which may be just as critical as cryptographic functions. For example, fault attacks have been used to compromise and attack Trusted Execution Environments (TEEs) from all major vendors, showing the severance of faults on commodity systems. To address this issue, it is necessary to develop and deploy generic countermeasures that can provide protection against fault attacks across a wide range of applications. Furthermore, it is important that countermeasures can be applied transparently to existing software, *i.e.*, automatically during compilation or with transparent hardware changes, to be able to protect existing codebases against fault attacks.

Problem Statement

Fault attacks have become a ubiquitous threat to a wide range of devices. Traditionally, this class of attacks was only applicable to small and embedded devices where the attacker has physical access, *i.e.*, performing a physical fault attack. However, new research methodologies have been developed such that fault attacks are now also applicable to commodity desktop and server processors. In this setting, faults can either be induced via a physical attack but also remotely via a software interface, *i.e.*, a software-induced fault attack. Consequently, a large set of different devices with different application and performance profiles are now susceptible to fault attacks.

Widely available architectures only consider a threat model from the software perspective. The hardware provides simple isolation primitives such as memory management/protection units or privilege modes, which are sufficient in their threat model. These protection schemes succeed in isolating code and data, providing integrity, and only granting access to their designated users. However, the software relies on a key assumption that the hardware is operating as expected, *i.e.*, there is a contract between the hardware and the software. In this contract, the software assumes that all executed instructions behave like it is specified in the instruction set specification. This assumption is not necessarily correct when faults are considered in the threat model since faults can manipulate the

hardware.

An attacker that has the capability to induce a fault can compromise a system. Even a single bitflip, e.g., during a conditional branch, can already be sufficient to completely undermine the full security of a system. Such a fault can bypass permission checks, the verification of a cryptographic signature, or elevate privileges. In fact, the correct control-flow of a program is essential to maintain the security guarantees of a system. Similarly, a memory access creates different attack points, where a fault attack can redirect the access and further compromise the system. Faults can also be combined with classical software attacks, forming a new class of combined attacks. These issues can have serious consequences, including loss of sensitive information, financial damage, and reputation harm.

In summary, current computing architectures are not equipped to handle environments in which an attacker can launch physical or combined physical and software attacks. The processing systems do not have the necessary safeguards to maintain the correct control-flow and to ensure the integrity of memory requests. As a result, these architectures are not yet suitable for use in such a setting. Hence, there is a need to develop new and effective methods for protecting software against fault or combined attacks that are transparently applicable to many devices.

Contribution and Outline

In this thesis, we advance the research of fault security by developing hardware extensions and compiler support. We focus our research in three directions: ① protecting data against fault attacks, ② providing Control-Flow Integrity (CFI) protection for software- and fault-based control-flow attacks, and ③ protecting the memory subsystem against faults. These three pillars offer foundational protection for secure software execution for different components of a system.

By integrating fault countermeasures at the algorithmic level, we show that we can build energy-efficient encryption schemes that meet the requirements of today's IoT platforms. In terms of CFI, we explore the usage of existing architectural features of ARM-based platforms and analyze if they can be repurposed to develop CFI protection for software- and fault-based control-flow attacks. By reusing existing hardware features, we show that fault resilience on the control-flow can be improved at a larger scale, as no hardware changes are needed. We show that state-based CFI protection schemes can act as the foundation for other countermeasures, as we use this method to protect the system call interface and conditional branches against faults. In the third part, we explore the efficient protection of memory accesses against faults. Integrating small hardware changes to the system combined with a toolchain allows us to automatically protect the memory subsystem from small embedded platforms up to application-class systems. The exploitation of compiler-based approaches allows us to automatically protect software against fault attacks, which are less error-prone and also applicable to existing code bases.

The threat models throughout this thesis comprise powerful attackers that

are capable of inducing fault in the related components of the system, e.g., the memory subsystem. We include faults independently of the induction method, *i.e.*, we cover physical and software-induced injection methods. The details of different induction methods and threat models are further detailed in Chapter 2.

This thesis is based on seven peer-reviewed papers, of which I am the primary author, as listed in Section 10.1. The remaining chapters of this thesis contain large verbatim blocks of these publications. At the beginning of each chapter, we state the contributions of each individual to those publications. Summarized, the contributions of this thesis are as follows:

Chapter 2 gives an overview of related fault attack methodologies. We discuss traditional fault attacks requiring physical access and also advanced software-based fault attacks. For both types of attacks, we show how they can be used to exploit different systems. Eventually, we summarize different state-of-the-art countermeasures for protecting general-purpose software against fault attacks.

Chapter 3 introduces the basic definitions of the control-flow of a program. We introduce different attacks on the control-flow, depending on the attacker's capabilities. We further discuss CFI, a generic approach to protect the control-flow against different attacks. In this context, we cover different protection schemes targeting different threat models.

Chapter 4 presents a hardware accelerator for energy-efficient fault- and side-channel secure encryption. Instead of deploying traditional redundancy-based countermeasures against fault attacks, we exploit the protection against fault at the algorithmic level by using fresh-rekeying. The encryption accelerator incorporates two encryption modes based on an AES-based stream cipher together with a leak-free re-keying function and Isap, a sponge based authenticated encryption scheme. The hardware accelerator is integrated into a multi-core System-on-Chip (SoC) and is taped out to an Application Specific Integrated Circuit (ASIC). In this work, we focus on evaluating the performance and energy efficiency of the cryptographic accelerator and based on different uses cases for this SoC. This chapter is based on two papers that were published TCAS-I'17 [Con+17b] and DATE'18 [Sch+18b].

Chapter 5 presents FIPAC, a new CFI protection scheme for ARM-based systems to counteract fault and software attacks. We build upon ARM Pointer Authentication (ARM PA), a new feature of recent ARM architectures that provides new instructions to cryptographically sign and authenticate pointers. We use this feature to create to cryptography-based fine-granular CFI protection scheme for ARM-based commodity systems. With the help of ARM PA, FIPAC computes a cryptographically secure Control-Flow Graph (CFG) at basic block granularity. An LLVM-based compiler instruments arbitrary C programs at the level of basic blocks of the CFG and supports the instrumentation with different checking policies. We provide an extensive evaluation to showcase the functionality and the performance overheads and discuss FIPAC's security

guarantees. The evaluation on the SPEC 2017 benchmark with different security policies shows a geometric mean code overhead between 51–91 % and a runtime overhead between 19–63 %. While these overheads are higher than for CFI protection schemes against pure software attacks, FIPAC outperforms related work protecting the control-flow against faults. This work was published at COSADE'22 [SNM22b].

Chapter 6 extends the scope of protection of FIPAC and protects the interface of user programs with the kernel. We show that state-based CFI protection schemes, such as FIPAC, can be the foundation to provide system call flow integrity. By statically linking the system call at the call site and dynamically verifying it in the Linux kernel, we provide system call flow integrity. Due to the dynamic instrumentation, we further provide security against side-channel and replay attacks. Our prototype is based on the original toolchain of FIPAC but includes a modified Linux kernel with support for dynamic CFI instrumentation and system call verification. The evaluation of micro- and macrobenchmarks based on SPEC 2017 shows an average runtime overhead of 1.9 % and 20.6 %, which is only an increase of 1.8 % over plain control-flow protection with FIPAC. This small impact on the performance shows the efficiency of SFP in protecting all system calls and providing integrity for the user-kernel transitions. This work was published at HASP'22 [Sch+23].

Chapter 7 aims to extend the control-flow protection of existing CFI protection schemes towards the data domain by protecting the control-flow of conditional branches. Conditional branches use application data to decide on the next program location. We protect this decision in a three-fold way: First, we encode application data to the redundant AN-code domain. Second, we develop new comparison algorithms that preserve the redundancy throughout the computation. Finally, we link the comparison result with the CFI protection scheme, thus, providing end-to-end protection for conditional branches. We extend the LLVM compiler so that standard code and conditional branches can be protected automatically and analyze its security. Our design shows that the overhead in terms of size and runtime is lower than state-of-the-art duplication schemes. This work was published at DATE'18 [SWM18].

Chapter 8 provides a new protection scheme to comprehensively protect pointers, pointer arithmetic, and memory accesses against fault attacks. We encode pointers to a redundant multi-residue representation, which provides fault protection without additional storage costs. Pointer arithmetic is protected within the domain of the multi-residue code. Finally, we link the encoded address to the data to protect the memory access against unintended redirects. To demonstrate the applicability of this concept, we extend a RISC-V processor with a residue-code Arithmetic Logic Unit (ALU) and linked load and store instructions and provide a prototype implementation based on an Field Programmable Gate Array (FPGA). We further develop a custom LLVM-based toolchain that transforms all pointer arithmetic and load and store instructions to the protected domain.

Our evaluations on embedded benchmarks show that the countermeasure induces an average overhead of 10 % in terms of code size and 7 % regarding runtime, which makes it suitable for practical adoption. This work was published at ACSAC'18 [Sch+18c].

Chapter 9 brings the previous work of Chapter 8 to larger applications by providing support for virtual memory. Protected virtual addresses are translated via a secure page table walk to yield a protected physical address for the actual memory access. With the same linking approach from Chapter 8, still on the protected physical address, SecWalk supports shared memory, which is needed for larger applications. To show the feasibility of this protection scheme, we extend an application-class RISC-V processor with a residue code ALU and a protected hardware-based page table walk. We present an FPGA prototype implementation and show that the area of the processor increases by 10 % . To show the applicability on real-life applications, we port the microkernel seL4 to SecWalk, which yields a code overhead of 13.1 % and a runtime overhead of 11.6 %. This work was published at HOST'21 [Sch+21].

Chapter 10 concludes this thesis by summarizing our achievements. We provide an outlook for future work and list all publications that have been included in this thesis and all additional co-authored publications.

2 Fault Attacks and Countermeasures

In this section, we first elaborate on different methods of fault attacks and related fault models. In the second part, we provide an overview of countermeasures and discuss encoding schemes that are used in this thesis for different designs.

2.1 Physical Fault Induction Techniques

In a physically induced fault attack, the attacker has physical access to the device and is in control of the operating conditions. During a fault attack, the attacker actively influences the device's operating conditions with the goal of manipulating an inner state of the system, *i.e.*, it is an active physical attack. At this operating point, the correct execution of all operations of the system is not guaranteed anymore.

Fault injection can occur from different induction methods. For example, by shortly manipulating the power supply of a chip, e.g., the power supply is reduced for a short period of time, a faulty operation can manifest within the chip [Bar+09; BFP19; BS03]. Similar to this approach, also the glitches on the clocks signal of a chip are used to induce a fault [Bar+06; PQ03; RSG21]. By overclocking the chip during a single clock period, the chip can fail to update a register, which for example, leads to a skipped instruction. To have better precision on the induced fault, more advanced but also more expensive induction methods come into play. By shooting with a laser or electromagnetic impulse onto the chip surface [BS97; Mor+13; SA02; Sel+15; WWM11; ZAV04], the fault has a new variable, *i.e.*, the location on the chip surface. There exist many other induction techniques, e.g., even the temperature [HS13] can used to disturb the operation of an electronic system. Especially in the context of processor

architectures, fault attacks are an effective method to skip or repeat instructions consistently [KH14; KSV13].

Meanwhile, commercial setups are available comprising the required hardware and the necessary software tools. For example, ChipShouter [New23] is a reasonably cheap fault injection setup for electromagnetic faults. There are even advanced fault injection setups available, comprising a computer-controlled X-Y coordinate table combined with a laser or EM-based induction technique [Ris23a; SGS23]. While these setups are more expensive, they provide better precision in injecting a fault.

Traditionally, these fault attacks target smaller systems such as smartcards or low-power embedded microcontrollers for Internet-of-Things (IoT) devices. However, recent research shows that fault attacks, such as undervolting, can also be applied to desktop processors. VoltPillager [Che+21] performs message injection on a system bus between the Central Processing Unit (CPU) and its voltage regulators using an external device. This approach allows the attacker to control the voltage to eventually undervolt the processor precisely. VoltPillager manages to bypass Intel SGX and performs key recovery attacks from cryptographic algorithms running inside the enclave. Interestingly, they also manage to create buffer under- and overflows, typically in classic software attacks.

A similar attack has been performed on recent AMD CPUs that support AMD Secure Encrypted Virtualization (SEV) [Buh+21]. SEV uses an internal hardware root-of-trust, the AMD Secure Processor (AMD-SP), a dedicated ARM-based microcontroller within the AMD System-on-Chip (SoC). Interestingly, the processor-controlled external voltage regulator also controls the voltage of the AMD-SP. By injecting messages on the communication bus between the CPU and the voltage regulator, Buhren et al. are able to glitch the voltage of the AMD-SP. They are able to exploit this to deploy custom firmware to the AMD-SP, which allows an adversary to decrypt a virtual machine's memory or extract the endorsement keys of a SEV-enabled CPU.

2.2 Software-Induced Fault Attacks

Traditionally, physical fault attacks require physical access to a device. However, with the advent of software-induced fault attacks, the constraint of requiring physical access can be relaxed. With this new induction technique, it is possible to induce faults in a system from software, which can even be performed remotely over the network interface.

It started with finding the Rowhammer effect [Kim+14], where an attacker frequently accessed the memory and thereby caused bitflips in the neighboring memory cells. Rowhammer was exploited to escalate privileges on Linux [Goo15], escape from software sandboxes [Goo15], or gain root privileges on Android devices [Vee+16]. The access pattern of the memory accesses depends on the relative location between the aggressor and victim cells. Still, the research community found patterns such as single-sided or double-sided [SD15], one-location [Gru+18], or the half-double pattern [Kog+22]. While this method

required local code execution, the attack methodology has been carried over to a remote setting. With Javascript.js [GMM16], it is possible to trigger the Rowhammer effect on client machines by loading malicious Javascript code from a website. ThrowHammer [Tat+18] or NetHammer [Lip+20] induce bitflips over the network interface. With SpyHammer [Oro+22], the Rowhammer effect can even be used to leak the temperature of the victim's DRAM module.

While Rowhammer is a new methodology to induce faults in a system, more classical fault injection techniques can be triggered via software. For example, Plundervolt [Mur+20], VOLTpwn [Ken+20], or VoltJockey [Qiu+19a; Qiu+19b; Qiu+20] exploit Dynamic Voltage and Frequency Scaling (DVFS) to manipulate the voltage of a processor. These attack methodologies modify the DVFS interface from software with the goal of inducing a fault in the processor. With this methodology, they are able to break the integrity of Intel SGX enclaves, leak AES encryption keys of AES-NI or ARM TrustZone, or break RSA-based authentication from ARM TrustZone. CLKscrew [TSS17], on the other hand, uses DVFS to manipulate the clock signal and induce faults in ARM-based Android devices. They use this methodology to extract secret encryption keys or to escalate privileges and load self-signed code into ARM TrustZone. In PMFault [CO23], researchers show that it is possible to mount a remotely controlled fault attack by controlling the PMBus. By exploiting software weaknesses within the Baseboard Management Controller (BMC), they are able to control the voltage of the CPU remotely and perform under- and overvolting attacks. Using this method, they are able to induce faults during an RSA computation, break the security guarantees of Intel SGX, or even brick the CPU.

When looking into all these new methodologies, one can observe that they all target larger application-class processors, even commodity desktop systems. Typically, these systems haven't had fault attacks in their threat model, but this new research shows there is a practical threat.

2.3 Fault Models

A fault model characterizes the effect and impact of a fault injection independently of the used fault induction method. It, therefore, provides a clear abstraction between the induction methodologies described in Section 2.1 and Section 2.2 and the actual effect to the system. Such a fault model makes it easier to design countermeasures, as it specifically describes the attacker's capabilities. Typically, a fault model M is described by a set of parameters, including the fault type t, the locality and precision l, and the timing τ, *i.e.*, $M = \{t, l, \tau\}$.

Fault Type t. There are different types, which can be categorized in the following types:

- *Stuck-At-Fault*: An intermediate signal, a flip flop, or a memory cell is forced to a specific value. When the signal is forced to the value zero, the

fault is denoted as *Stuck-at-Zero* or *SA0*. When the signal is forced to the value one, the fault is denoted as *Stuck-at-One* or *SA1*.

- *Bitflip*: The value of a signal, flip flop, or memory cell gets inverted.
- *Random Fault*: A signal, flip flop, or memory cell gets assigned a random value.

Fault Location and Granularity l. The precision determines at what granularity a fault can be injected, e.g., the attacker can only inject a fault at a single bit or perform a multi-bit fault injection. It further defines the locality of the fault, e.g., is the attacker capable of inducing a fault precisely at a specific gate or register, at a particular subsystem of the chip, or has the attacker no control over the location.

Fault Timing τ. The timing determines how precise a fault can be injected in the temporal domain. Depending on the synchronization mechanisms of the device under attack, the timing may be more precise or not.

Fault Duration $t\tau$. Faults can be categorized as transient, permanent, or destructive faults. A transient fault only occurs during a single operation, e.g., one addition operation of the CPU is disturbed, but the next addition computes correctly. A permanent fault can happen on registers that a subsequently read, and its value is used for further computations. If there is a fault there, all computations are permanently disturbed until a new value is written to the register or the device is put into reset. A destructive fault permanently modifies the calculation and does not vanish after resetting.

These fault types described before can apply to specific components of a system, *i.e.*, with a particular locality. For example, the fault model can specify a random multi-bit fault on the program counter of a processor, that allows the attacker to redirect the control-flow.

2.4 Fault Exploitation

Injecting faults into a digital circuit is a powerful threat allowing adversaries to break the security of a system entirely. Typically, the effect of a fault is modeled at the bit level with transient bit-flips and permanent stuck-at effects [VKS11].

Injecting a fault, irrespective of the method used, does not necessarily break or exploit the system. Nevertheless, the research community studied different applications where faults are dangerous for the overall security. It started with the well-famous fault attack on the RSA cryptosystem [BDL97] and evolved ever since. A single fault during the computation is enough to recover the private key and break the security.

2.4.1 Fault Attacks on Cryptographic Implementations

In the past, cryptographic algorithms were a traditional target of fault attacks. Boneh et al. identified a vulnerability in the RSA cryptosystem [BDL97], where a pair of valid and faulty signatures are enough to reveal all the private information. Attack methods also evolved for symmetric cryptographic primitives. Biham and Shamir [BS97] developed a method technique Differential Fault Analysis (DFA) to reveal the private encryption key for the DES cipher, which emerged in its own field of research. This approach was applied to many other symmetric ciphers [AMT13; BGN05; CY03; SBM15; TMA11]. These attacks solely analyze the correct and faulty computations and then are able to recompute the secret encryption key.

Later on, research developed different strategies to exploit faults in cryptographic implementations. This research has led to the development of attack methodologies such as Fault Sensitivity Analysis (FSA) [Li+10], Differential Fault Intensity Analysis (DFIA) [Gha+14], Statistical Fault attacks (SFA) [Fuh+13], or Statistical Ineffective Fault Attacks (SIFA) [Dob+18]. The broad research on this topic shows that fault attacks pose a serious threat to various different cryptographic implementations in practice.

2.4.2 Exploiting Faults on Non-Cryptographic Software

While previous works focused on the exploitation of faults in cryptographic implementations, faults are also a threat to general-purpose software. As discussed previously, faults can effectively be used to skip instructions, redirect the control-flow, or manipulate the data. Consequently, these effects can seriously affect the system's security. By inducing targeted faults into the program counter of a processor, faults enable an adversary to arbitrarily hijack the control-flow of a program [NT; TM17; TSW16].

One prominent target of fault attacks in the context of non-cryptographic software is secure boot. Different fault injection methodologies are used to compromise the boot process of various architectures. For example, voltage glitching on ARM-based architectures [Her+21; TS16] bypasses this first layer of security. This has been extended to also use electromagnetic fault injection [CH17] or even laser fault injection [Vas+20]. Vasselle et al. [Vas+20] perform laser fault injection on a large application-class SoC of an Android phone to bypass secure boot.

Typically, faults are used to break the security of a device, for example, to break dedicated security defenses. For example, [SMS23] uses voltage fault injection to break the security of ARM TrustZone-M, a Trusted Execution Environment (TEE) for embedded ARM processors.

With the rise of computing power, the automotive area also becomes a target of fault attacks, especially Electronic Control Units (ECUs). For example, in [WP17], fault attacks are used to attack Automotive Safety Integrity Level (ASIL) certified processors. While the safety mechanisms in those processors increase the complexity of fault attacks, they do not prevent them. Further-

more, electromagnetic fault injection is used to attack recent processor cores of ECUs [OFl20]. Faults are also used to attack the diagnostic protocols of these systems [PC18]. In [NT], voltage glitching is used to attack the AUTOSAR operating system, which is widely used on many ECUs. Within their exploit, they are able to reach arbitrary code execution by causing an attacker-controlled value to be loaded in the program counter of the processor.

Fault attacks also pose a threat to devices used in the financial sector. While smartcards have dedicated countermeasures against faults, voltage glitching is, for example, used to attack the Trezor hardware Bitcoin wallet [RND19] to retrieve the private key of the device. In other attacks, the operating system of a smart card was faulted [AK] in order to get access to a Pay-TV service. Furthermore, fault attacks were used to exploit the bootloader of the Xbox360 gaming console [Fre11] to be able to run counterfeit software on the gaming device. More recently, the Apple AirTag – a Bluetooth beacon that connects to the Apple networks – was jailbroken via a fault attack on the included microprocessor [sta21]. By exploiting a fault, researchers could jailbreak the device, dump the firmware, and upload custom software.

As discussed in Section 2.1 and Section 2.2, new attack methodologies enable fault attacks also for larger systems, e.g., desktop and server-class processors. In their proof-of-concept exploits, they attack the TEEs of the host processor, which often processes sensitive data or performs critical operations. Exploits show they are able to extract sensitive data, decrypt protected memory, or leak encryption keys from a trusted and protected environment. Summing up, these exploits show that fault attacks also pose a serious threat to non-cryptographic applications that are even running on larger systems.

2.5 Countermeasures Against Fault Attacks

In the physical domain, there exist sensor-based approaches to detect fault injection at the chip level [HBB16; Mut+22]. Such sensors aim to detect unintended manipulations of certain signals or properties, e.g., the power supply or clock signal. In case of a fault, the system can react adequately depending on its escalation policy. However, these mechanisms are outside the focus of this thesis.

In the digital domain, redundancy is required to defend systems against fault attacks effectively. Thus, it is necessary to implement dedicated countermeasures, which include a variety of techniques, such as the use of redundant components or systems, error detection, and correction algorithms. Countermeasures can either be implemented in software or hardware and have different trade-offs. One fundamental property that all of these countermeasures have in common is redundancy. By introducing redundant elements into a system, it becomes possible to detect errors or faults that may be introduced by an attacker or even correct them. Redundancy can be added using different mechanisms, but there are generally two forms.

Spatial Redundancy

For countermeasures that are based on spatial redundancy, the systems exploit redundancy in the spatial domain. A classic example of protecting data with this approach are Error Detection Codes (EDCs), where data is either encoded or concatenated with certain bits of redundancy. These additional bits allow the system to detect up to a certain amount of bitflips in the original data. More advanced encoding schemes even support the correction of bitflips, which is prominently used for server-class memory in ECC RAM. Different schemes support binary or arithmetic operations directly in the encoded domain extending their scope of protection also for the computation of data. We will discuss different encoding schemes in Section 2.5.2 that are later in this thesis exploited in concrete designs in Chapter 7, Chapter 8, and Chapter 9.

Temporal Redundancy

The second form is temporal redundancy, where the redundant behavior is performed in the time dimension, *i.e.*, the same operation is performed twice or multiple times. Only if all operations yield the same result the computation is considered genuine without manipulation. If the operation is executed more than two times, Triple Modular Redundancy (TMR) can also be applied to temporal redundancy to provide error correction. Both approaches can be implemented both in hardware and software with different trade-offs.

Protection of Cryptographic Algorithms

To counteract fault attacks on cryptographic implementations, research has developed dedicated countermeasures that are specifically tailored for certain cryptographic algorithms [RG14]. To protect the AES block cipher, they compute a parity of the data at the beginning of one encryption round. In the following, they also compute the parity then over the linear and non-linear parts of the cipher. In the end, they perform a parity check [MSY06] to match the parity of the output data with the computed parity over the cipher round. While this approach increases the security, it is susceptible to multi-bit faults, as there is only one parity bit, and its code is a linear function. To improve the security, more advanced countermeasures use multi-bit parity codes with a non-linear function.

There also exist several proposals for countermeasures for different cryptographic schemes [Bou+12; Kis+16; KQ07; MSY06]. These countermeasures are mostly specifically tailored to a specific cryptographic algorithm, thus not generic. New fault attacks on different algorithms may require new methods for protection. In the following, we discuss generic error detection codes that apply to many applications.

2.5.1 Duplication-Based Countermeasures

Instruction duplication is the simplest form of redundancy as it is generic and applicable to many different areas. In practice, duplication-based approaches exist in software and hardware and form temporal- or spatial-based countermeasures.

Instruction Duplication

Duplicating instructions or modular redundancy has been shown to be a generic countermeasure to protect arbitrary software against fault attacks [Bar+10b]. After duplicating an instruction, a check sequence is inserted into the code that errors if both computations differ, thus forming a temporal protection scheme. Conditional branches are protected by replicating the branch multiple times, resulting in a comparison tree. Only if all comparisons yield the same result the conditional branch is considered to be genuine. While such a scheme improves the security, inducing the same fault multiple times can bypass the protection. Barry et al. [BCR16] automate the process of duplication with the help of a compiler, such that duplicated instructions always write the same register and do not perform any check operations. However, they only assume a limited instruction skip fault model and do not consider faults in data or the computation.

Duplication can also be performed on different granularity at the software level. For example, a certain function can be called twice or even three times, and then the result is compared. In the case of three executions, one error can even be corrected, yielding a classical TMR protection scheme. To bring in diversity, the implementations of the called function may differ but implement the same specification.

Duplicating instructions is not only suitable for generic software; it is also applicable to cryptographic implementations. For example, Barenghi et al. [Bar+10a] perform instruction duplication with subsequent checks, *i.e.*, one operation expands to four (including the comparison and branch for the error case), protecting against single errors. They extend their scheme to perform instruction triplication to even perform single-error correction or double-error detection. By selectively applying their countermeasure to cryptographic implementations, *i.e.*, they only protect the last three rounds, they yield overheads between 83 – 330 %.

Spatial protection, in the form of parallel instruction execution, can also be implemented in software. Single Instruction Multiple Data (SIMD) instructions perform the same operation on multiple parts of one data word, e.g., four 8-bit additions performed on a 32-bit data word. While these instructions are traditionally used for data processing, they can also be used to perform redundant computing in the context of fault attacks. By executing the same data in parallel data streams followed by comparisons, arbitrary software can be protected against faults [Che+17; Che18; Lac+18]. However, these approaches are limited by the available SIMD instruction and their precision, e.g., they only support 8-bit or 16-bit computation.

Circuit Duplication

Duplication or replication is also applicable to hardware implementations. Here the logic and/or flip-flop elements are replicated with comparison and/or voting elements at the end, thus providing spatial protection Synthesis tools such as Synopsys Synplify [Syn23] can automate this process and automatically generate TMR for specified hardware. These tools can automatically insert TMR on different granularities, *i.e.*, they replicate the hardware on the logic- or block-levels for annotated modules.

Spatial redundancy can also be implemented at the algorithmic level, e.g., in hardware, where a circuit is replicated twice or multiple times. The operation is then performed independently on all circuits in lock-step, and their results are compared to see if they match, thus being able to detect a fault. This concept is, for example, used in the OpenTitan [Ope23b] project, where the internal processor is duplicated. The second core operates lock-step with a random delay of a few cycles to provide the necessary error detection capabilities. If the circuit is replicated more than two times, it even supports error correction, *i.e.*, TMR. For example, in the commercial TriCore architecture from Infineon [Inf23], two cores can operate in lock-step, and checks are performed in between. They use this configuration mainly for safety-critical applications in the automotive area.

Inverse Computation

Concurrent error detection, such as used to enhance the reliability of devices, is used to implement spatial redundancy. One approach of a countermeasure aims to generically protect encryption/decryption schemes [Kar+02]. First, the input of a computation is first buffered in a dedicated register. After encrypting the data, the result gets decrypted again, which must yield the original input data again. By using two different computations, *i.e.*, encryption and decryption, this countermeasure also adds diversity on top of redundancy.

A similar approach can be used for involution ciphers [JWK04], which have the property $x = F(F(x))$ on the encryption or on inside components. Then after performing a cryptographic operation, e.g., one round of computation, the result is taken, and by exploiting the involution property, the input data is recomputed. Finally, the result is compared with the buffered input data, and an error is raised if the data mismatches.

2.5.2 Error Detection Codes

To counteract fault attacks and to protect data against unwanted manipulations, Error Detection Codes (EDCs) [Bro60; Pet; WKK09] are a long-established and well-studied research field. Their principle is to encode the data and add additional redundancy bits such that unwanted manipulations can be detected. While initially being developed to protect data during storage or transmission in harsh environments, more advanced EDCs even support computation directly on encoded data. Such an encoding scheme has two advantages: First, it omits

the necessity of decoding the data, which significantly improves the runtime performance when operating with encoded data. More importantly, it provides end-to-end protection of data throughout computation since the system never uses plain unencoded data. In the following sections, we discuss different encoding schemes and show their advantages and disadvantages.

A binary block encoding scheme uses code words of n-bit length to represent datawords of length k, with $k < n$. The append $r = n - k$ bits provide the redundancy properties for the encoding scheme. Typically, binary block codes are denoted as $[n,k]$- or $[n,k,D]$-codes, where D denotes the minimum pairwise Hamming distance between all code words. If all bits of the dataword are embedded in the code word, we denote it as a *systematic* code. One advantage of such a code is that the dataword is always visible without needing to decode the code word. Consequently, in *non-systematic* codes, the data and redundancy parts are mixed up and require a dedicated decoding operation to retrieve the dataword again. A systematic code may separate the operations on the data- and redundancy part, thus, forming a *separable* code. If there is only a single computation performed on the combined data and redundancy, the code is called *non-separable*.

Binary Linear Codes

One example of an encoding scheme that supports computation in the encoded domain are binary linear codes [Ham50], which are defined by its $k \times n$ generator matrix $\mathbf{G}$. For systematic linear codes using the Galois Field GF(2), the generator matrix has the form $[\mathbf{I}|\mathbf{P}]$, where $\mathbf{I}$ is a $k \times k$ identity matrix, and $\mathbf{P}$ is the $k \times r$ parity matrix. To encode data to the binary linear code, a left-side multiplication with the generator matrix $\mathbf{G}$ is performed. Decoding is done by a right-multiplication of the code word with the code-specific parity matrix that can be computed out of the generator matrix. For binary linear codes, the parity matrix is defined by $\mathbf{H} = [\mathbf{P}^T|\mathbf{I}]$. The result of that multiplication is the syndrome s, which is always zero if the code word was error free. However, if the code word includes an error (up to the Hamming distance of the code), the syndrome s becomes a different value than zero.

From the linearity property of this encoding scheme follows that the sum of code words again yields a valid code word. In the Galois Field GF(2), the sum is equal to the XOR operation; thus, binary linear codes are closed under XOR. Binary linear codes also support the AND operation, but it requires specific correction terms to compute the result. Furthermore, these codes only preserve the error detection capabilities if only one operand is affected by a fault.

ECC Memory. One prominent use case of binary linear codes is ECC memory, specially designed for server-class or industrial control applications. Here, the RAM module contains dedicated memory chips that store the error-correcting codes for the data. Typically, these memory modules use a Single-Error Correction and Double-Error Detection (SECDED) Hamming code, which gets computed and checked by the memory controller of the system. This encoding scheme

supports the detection of up to two bitflips and the correction of even a single-bit error.

AN(BD)-Codes

One example of arithmetic codes are so-called AN-codes [Bro60; For], which encoding step is defined in Equation (2.1). A multiplication between the functional value n and the encoding constant A forms the code word n_c. Note the subscript c denotes the encoded value.

$$n_c = A \cdot n \tag{2.1}$$

Using this multiplication, only multiples of the encoding constant A are valid code words; everything in between correlates to an invalid value. By multiplying the data with the encoding constant, the redundancy information is bound to the code word and cannot be separated. Thus it forms a *non-systematic* and *non-separable* encoding scheme.

The redundancy properties of the AN-code are defined by the encoding constant A. The minimum Hamming distance between all code words gives a quantitative measure of how strong the chosen encoding constant is. Note that AN-codes limit the functional value to be less than the encoding constant to preserve the error detection capabilities.

To verify if the code word is valid, a modulo operation with the encoding constant is performed as defined in Equation (2.2). For a valid AN-code, this remainder must be zero. To decode the code word and retrieve the original data, an integer division with the encoding constant A is performed.

$$0 \equiv n_c \bmod A \tag{2.2}$$

AN-codes limit the functional value to be less than A to preserve the error detection capabilities. The encoding constant is chosen by the designer and defines the redundancy properties of the code. The minimum Hamming distance between the code words gives a quantitative measure of how strong the selected A is. Finding a good A so far is limited by exhaustive search [MS09], but good encoding constants have already been found. Hoffmann et al. [Hof+14] call these constants so-called *Super A*s because their minimum Hamming distance is maximal for a given word width. Furthermore, also other parameters, such as the maximum data size of the system, influence the selection.

AN-codes are arithmetic codes and, therefore, also support encoded processing [For] for certain arithmetic operations. Since AN-codes are closed under addition and subtraction, which is shown in Equation (2.3), these operations do not require any modification. Other operations, such as multiplication and division, require special correction terms but are supported.

$$\begin{aligned} z_c &= x_c + y_c \\ &= A \cdot x + A \cdot y \\ &= A \cdot (x + y) \\ &= A \cdot z \end{aligned} \tag{2.3}$$

Fetzer et al. [FSS09] use this encoding scheme to build an AN-code LLVM compiler, which transforms all operations to the domain of AN-codes, to protect the data processing. Unfortunately, even plain AN-codes already add slowdowns of a factor of 30, thus limiting its practical application.

Unfortunately, AN-codes can only protect the arithmetic operations, but they do not protect the memory access of the data. Forin and Schiffel et al. [For; Sch+10] extend simple AN-codes by assigning a variable-dependent signature B_x to each encoded variable x_c, forming a so-called ANB-code. This yields the encoding formula as shown in Equation (2.4). Note that the variable-dependent signature B_x must be less than the encoding constant A.

$$x_c = A \cdot x + B_x \tag{2.4}$$

By adding the variable-dependent signature B_x to the AN-code, the AN-code property that all encoded values are a multiple of A is intentionally destroyed. Since B_x is less than A, decoding works the same as for normal AN-codes using an integer division. A check is also performed using a modulo operation with the encoding constant, which now must yield the signature B_x. Schiffel et al. automated this process and developed a compiler toolchain to keep track of all assigned signatures and to insert the correct check operations. By assigning a variable-dependent signature to the code words, a wrong memory access can be detected, as long as signatures do not cancel out due to arithmetic. However, using this encoding scheme in practice is challenging. First, ANB-codes have a significant overhead of around 90 % on average on top of ordinary AN-codes. Second, the signature B_x must be less than the encoding constant A, limiting the number of variables that can be protected. Furthermore, every location in memory requires a different signature, which is not practical.

Forin et al. even extended their encoding scheme further to detect outdated uses of a variable. They introduce a timestamp D to the code word that counts the variable updates, yielding the so-called ANBD-code. This yields the encoding formula as shown in Equation (2.5).

$$x_c = A \cdot x + B_x + D \tag{2.5}$$

To verify the correctness of a variable, the variable-dependent signature B_x and the timestamp D need to be known. ANBD-codes are expensive in both code and runtime overhead. On average, they slow down the program by a factor of 150 [Sch+10]. Therefore, this encoding scheme can only be used selectively to protect certain regions of a program.

Residue Codes

Residue codes [Mas64] form a different class of arithmetic codes. In this encoding scheme, a code word x_c is defined by concatenating the data with its residue $x_c = (x, r_x = M|x)$. Here, x denotes the payload data, and r_x is the redundancy part, the residue. The residue is computed as the remainder with respect to a modulus M, which defines the redundancy properties. Due to this concatenation, the tuple of data and residue are separable, on which independent operations are performed. Therefore, such an encoding scheme forms a *systematic* and *separable* encoding scheme.

This property allows the system to easily access the payload data without expensive decoding operations. Although the modulus M defines the robustness of this code, a simple bitflip on the data and on the modulus can create a new valid code word. Thus, the Hamming distance between two simple residue encoded code words is only 2.

To scale the robustness of residue codes, and to yield a higher Hamming distance, the number of residues can be increased, forming a multi-residue code [Rao70; RG71]. The modulus M is now defined by $M = \{m_0, \ldots, m_n\}$, where m_i is the actual modulus for one residue and n is the number of residues. Similar to AN-codes, finding a good set of moduli is a challenging task and is currently only possible via exhaustive search but in a more efficient way [MS09]. Since (multi)-residue codes are arithmetic codes, they also natively support certain arithmetic operations within the encoded domain. Here, the arithmetic operations are performed both on the functional data part and on the residue value. Equation (2.6) shows the arithmetic addition operation performed on multi-residue encoded data. First, the addition is performed on the plain payload data. Second, the addition is performed on every residue independently, followed by a modular reduction with the corresponding moduli m_i.

$$\begin{aligned} z_c &= x_c + y_c \\ &= (x + y, \forall i : m_i|\,(r_{i,x} + r_{i,y})) \end{aligned} \tag{2.6}$$

Similar to the addition, residue codes also support subtractions as shown in Equation (2.7).

$$\begin{aligned} z_c &= x_c - y_c \\ &= (x - y, \forall i : m_i|\,(r_{i,x} - r_{i,y})) \end{aligned} \tag{2.7}$$

Residue codes also support protecting the multiplication operation. However, for the work in Chapter 8 and Chapter 9, we only require additions and subtractions.

2.5.3 Fresh Re-keying to Counteract Fault Attacks

The probability for a key recovery via Differential Power Analysis (DPA) to be successful rises with the number of side-channel observations for different

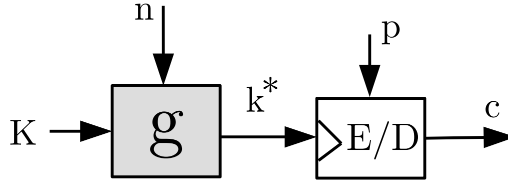


Figure 2.1: Re-keying g function combined with an encryption/decryption primitive.

inputs under the same key K. Therefore, one approach to counteract DPA is frequent re-keying [Koc03; Med+10], where the goal is to design a cryptographic scheme such that for a certain key K, the number of different inputs to the underlying cryptographic primitive is upper-bounded by some small number q (q-limiting [Sta+10]). As soon as the limit of q different inputs is reached, another key K' is selected, which limits the data complexity per single key K. Thus, for a certain key K, the cryptographic implementation can only generate the side-channel leakage for q different inputs, which effectively limits the feasibility of DPA to recover K. As a result, the implementation of the cryptographic primitive is only required to resist Simple Power Analysis (SPA) attacks. By always using a new encryption key, no encryption is performed twice with the same key, thus also providing DFA security by design at the algorithmic level.

Always using a fresh key is expensive in terms of secure storage, key exchange, and other factors. Therefore, re-keying schemes have been developed that take a secret master key and a fresh value that is typically not secret to compute a fresh intermediate session key. In Figure 2.1, we show such a scheme, where a re-keying function g is combined with an encryption or decryption primitive. The re-keying function g uses a secret master key K and a fresh public nonce n as the input to compute the single-used session k^*. For every encryption, the re-keying function then computes a fresh session key k^*, which is only used *once* and is different for every encryption or decryption operation. However, this approach shifts the burden of DPA protection from the cipher to the re-keying function. Since the re-keying function is much easier in complexity, traditional protection schemes such as masking [AG03; GP99] or shuffling [Vey+12] are applicable with less overhead. This approach also mitigates DFA attacks on the encryption primitive by design. By always using a fresh nonce, and therefore using a fresh session key k^* for the encryption or decryption operation, an attacker cannot observer a faulty and non-faulty encryption output, which is necessary to mount a DFA attack. This mitigation strategy is vastly different compared to redundancy-based approaches, as it protects the cipher on the *algorithmic* level by extending the mode of operation. While shuffling the operations is primarily used to increase the complexity of side-channel attacks, it also increases the complexity of fault attacks. Due to the random sequence of operations, precise fault injection is not possible anymore, thus reducing the probability of successful attacks. However, it depends on the concrete attack and their required fault granularity if shuffling increases the protection. Masking generally does not increase the protection level

in terms of faults [BH08].

There are several suitable designs for the re-keying function $g : (K, n) \mapsto k^*$ available. One prominent way is to build g such that it is easy to protect with classical countermeasures such as masking or shuffling. Medwed et al. [Med+10] propose a polynomial multiplication in a finite field of the key master key K and a nonce n that fulfills those properties and is easy to protect. However, as pointed out in [Bel+15; BFG14; Dob+14; GJ16; Med+11; PM16], the algebraic structure of a multiplication opens the door to combined attacks on g and the encryption. To mitigate this type of attack, Dobraunig et al. proposed to add a block-cipher-based feed-forward computation [Dob+15] after the multiplication, which we further detail in Section 4.1.4.

Another approach for designing a secure re-keying function is exploiting a GGM construction [GGM86] as proposed in [FPS12; Sta+10]. The GGM construction is a tree-like approach to mix a secret K with a public n, where on each tree level, exactly one bit of the public n is evaluated. Starting with $s_0 = K$, the key s_{i+1} is computed by encrypting one of two predefined plaintexts p_0 or p_1 ,with the key s_i and block cipher E, depending on the i-th bit of n. The output of the last level is then, after post-processing, used as the session key k^*. GGM-based re-keying is 2-limiting and hence considered to be secure against DPA. As a variant, [Dob+17] presents the sponge version of GGM-based re-keying, IsapRk, which we further detail in Section 4.2.

3

Control-Flow, Control-Flow Attacks, and Control-Flow Integrity

The control-flow of a program refers to the order in which all instructions of a program are executed. It determines the path the program takes from the beginning to the end of its execution. We generally differentiate between strictly linear or sequential instructions, e.g., arithmetic operations that do not actively manipulate the control-flow, and dedicated control-flow instructions like branches or jumps. These instructions actively change the program counter of the processor to an arbitrary location, thus affecting the control-flow.

Conditional branches fall into a special category of instructions since, at this point, data merges with the control-flow, or in other words, the application data modifies the control-flow. Depending on the outcome of a previous comparison operation, the branch is executed and jumps to a different location. If the comparison is false, the subsequent instruction gets executed. Conditional branches can either be implemented with a separate comparison and branch instruction, like in the ARMv8 [ARM20] architecture. In such architectures, the comparison instruction modifies a CPU flag that the actual branch instruction uses to decide if the branch should be taken. On the other hand, RISC-V [Wat+14] defines a single combined compare and branch instruction in their Instruction Set Architecture (ISA).

The code of a program is laid out in segments that only contain zero to many strictly sequential instructions followed by at least one control-flow instruction. These segments are denoted as *basic blocks*. The control-flow instructions at the end of a basic block connect the different basic blocks of the program, forming the Control-Flow Graph (CFG). The nodes or basic blocks of this graph are direct edges and denote the control-flow transfers. Figure 3.1 shows a CFG with

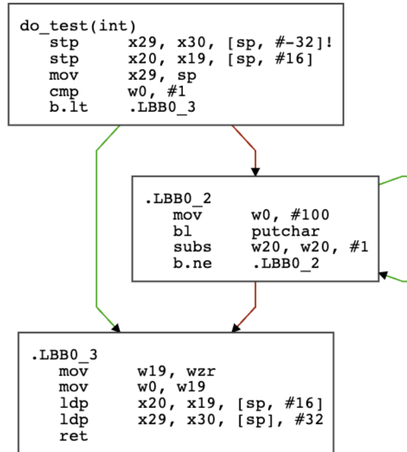


Figure 3.1: CFG showing basic blocks with ARMv8 instructions.

different basic blocks, direct, and conditional branches with instructions from the ARMv8 ISA [ARM20].

3.1 Control-Flow Attacks

In a control-flow attack, the adversary hijacks the program's control-flow to redirect it to a different location. Depending on the origin of the hijack, we differentiate between software- and fault-based control-flow attacks.

3.1.1 Software-Based Control-Flow Attacks

In a software-based control-flow attack, the adversary exploits a memory bug and overwrites code- or data pointers. Successful control-flow attacks use the return address [Sha07], jumps addresses [Che+10], or data pointers [Hu+16]. This attack allows the attacker to arbitrarily switch locations within the program when executing a hijacked code pointer. However, in the software setting, the attacker's possibility to hijack the control-flow is by overwriting a code pointer that is later jumped to.

3.1.2 Fault-Based Control-Flow Attacks

Although faults can be used to attack the same control-flow data, *i.e.*, return addresses, code- or data pointers, faults increase the attack surface. While in a software-based control-flow attack, the adversary is limited by the exploitability of the underlying memory vulnerability, faults allow to hijack the control-flow arbitrarily and at a much finer granularity. Fault-based control-flow attacks can corrupt [LG19] or skip instructions [Blö+14], change the program counter [NT;

TM17; TSW16], or modify addresses used by indirect or direct calls in registers, memory, or the code segment [Mor+13; TBC19]. These attacks target the control-flow within (*intra*) or over (*inter*) a basic block, *i.e.*, consecutive instructions without control-flow. While intra-basic block attacks allow the attacker to skip/manipulate individual instructions in a basic block, inter-basic block attacks enable the attacker to redirect the control-flow to an arbitrary code position by corrupting the addresses of branches/calls.

3.2 Control-Flow Integrity

To protect a program from intra/inter-basic block control-flow attacks, enforcing Control-Flow Integrity (CFI) has shown to be an effective and generic defense [Aba+05]. Existing software-based CFI protection schemes provide different enforcement granularities and either addresses a software *or* fault attacker but not both. Although there are schemes addressing both threats, they require intrusive hardware changes, which are not feasible for commodity systems.

3.2.1 Software CFI Protection Schemes

Software CFI (SCFI) [Aba+05] protects the program against software-based control-flow attacks while not denoting how it is implemented, *i.e.*, in software or hardware. Such a countermeasure enforces the CFG extracted at compile-time by dynamically protecting a subset of inter-basic block control-flow transfers at runtime. This coarse-grained CFI policy only protects indirect calls or returns, as they are the only targets of a memory vulnerability. The CFI instrumentation is either inserted during the compilation [Tic+14] or on the complete binary [ZS13].

To improve the performance of such countermeasures and to make the deployment more practical, the CFI policy was gradually relaxed. CPI [Eva+15] and CCFI [Mas+15] protect a broad range of forward- and backward-edges of the program by maintaining the integrity of code-pointers. PARTS [Lil+19] protects code-pointers by signing and verifying them using ARM Pointer Authentication before using them. When using a signed code-pointer on an indirect call, for example, the `blraa` instruction first verifies the integrity of the signed target address before taking the branch. If the verification fails, *i.e.*, the Pointer Authentication Code (PAC) does not match the expected PAC, the application traps and aborts. PACStack [Lil+21] protects return addresses on the stack by utilizing ARM Pointer Authentication (ARM PA) to cryptographically link and verify them.

3.2.2 Fault CFI Protection Schemes

Fault CFI (FCFI) protects against fault-based control-flow attacks, thus providing protection on a much finer granularity. Such protection schemes can either be implemented in software or with hardware support. FCFI protection schemes capable of *detecting* intra-basic block control-flow hijacks, e.g., instruction skips,

employ a global CFI state. This state gets updated with the execution of each instruction, *i.e.*, the CFI granularity. Maintaining and checking a state at this granularity is expensive, so these schemes require hardware changes [Cle+17; Sug11; Sul+17; Wer+18; WWM15]. As this is not possible for commodity devices, software-based FCFI protection schemes provide a trade-off between security and performance by protecting all control-flow transitions between basic blocks, hence, providing inter-basic block CFI. In CFCSS [OSM02] and SWIFT [Rei+05], each basic block is assigned a signature to update a global CFI state. When entering the basic block, the global control-flow state is updated with this signature and is compared to match the expected state at this location.

Algorithm 1 CFI state update function.

1: **function** UPDATE(S, Sig_{BB})
2: $r_1 \leftarrow \text{Sig}_{BB}$
3: $S \leftarrow S \oplus r_1$
4: **end function**

Algorithm 1 shows an XOR-based CFI state update function, as used in CFCSS [OSM02], where a global CFI state S is XORed with the basic block signature Sig_{BB}. At specific program locations, checks are included, comparing the CFI state to the expected value to detect control-flow deviations. This approach of CFCSS yields a runtime overhead between 107–426 % [Gol+03]. ACFC [VHM03] reduces the performance penalty down to 47 % by decreasing the checking precision and thereby reducing the security guarantees. Other approaches [HLB19; LHB14] annotate the source code with counter increment and verification macros to detect control-flow deviations. However, a protection scheme requiring manual source code modifications is not practical. It cannot easily protect legacy code, making a large-scale deployment hard.

4 Energy-Efficient Encryption with Algorithmic Fault Protection of IoT End-Nodes

Cryptography is a fundamental aspect of modern information security and is crucial in protecting various assets during computation, storage, and communication. Especially with the rise of the Internet-of-Things (IoT), where many devices are connected over the network, and vast amounts of data need to be stored and transmitted, efficient protection is necessary. Cryptography provides means to achieve these goals to provide security for data at any stage of the device. By utilizing cryptographic techniques, sensitive information can be shielded from unauthorized access, ensuring confidentiality, integrity, or authentication.

As discussed in Chapter 2, cryptographic algorithms can be attacked using faults. Especially if devices are deployed in hostile environments, as it is the case for the IoT, this threat becomes much more prominent. Consequently, the cryptographic algorithms on such exposed devices require dedicated protection mechanisms against fault attacks.

Devices in the IoT do not only face requirements from the security perspective. The device stack of the IoT ranges from cloud solutions to IoT gateways down to IoT end-nodes, which form the smallest class of devices. Since IoT end-nodes must work within a tiny power envelope, this fact introduces a significant limit on the volume of data that can be processed and transmitted. To address this limiting factor, near-sensor smart data analytics, directly on the IoT end-node, is a promising direction.

Addressing the requirements regarding of power and energy, as well as protection against physical attacks, is a challenging topic nowadays, more important

than ever. Adding dedicated countermeasures against physical attacks increases power consumption and limits the applications of such IoT end-nodes. One promising direction is the combination of re-keying, as discussed in Section 2.5.3, combined with a leakage-resilient encryption primitive that protects against fault attacks at the algorithmic level. However, currently, there is no public research that practically shows the suitability of such schemes in the setting of IoT end-nodes with a limited power and energy budget. Consequently, it requires concrete hardware implementations and thorough evaluations in suitable use cases to prove its relevance for IoT end-nodes.

Contribution

In this chapter, we propose HWCRYPT, a highly efficient hardware cryptographic engine for fault- and side-channel secure encryption modes. The cryptographic hardware accelerator provides fault- and side-channel secure encryption primitives that are protected at the algorithmic level rather than employing traditional countermeasures such as redundancy. HWCRYPT is integrated into the 65 nm *Fulmine secure data analytics* System-on-Chip (SoC), which tackles the two main limiting factors of IoT end-nodes while providing full programmability, low-effort data exchange between processing engines, (sufficiently) high speed, and low-energy. The SoC is based on the architectural paradigm of tightly-coupled heterogeneous shared-memory clusters [Con+14], where several engines, which can be either programmable cores or specialized hardware accelerators, share the same first-level scratchpad via a low-latency interconnect. In *Fulmine*, the engines are four enhanced 32-bit OpenRISC cores, HWCRYPT, the highly efficient cryptographic engine for fault- and side-channel secure encryption, and one multi-precision convolution engine specialized for Convolutional Neural Networks (CNN) computations. Due to their memory-sharing mechanism, cores and accelerators can exchange data in a flexible and efficient way, removing the need for continuous copies between cores and accelerators. The proposed SoC performs computationally intensive data analytics workloads with no compromise in terms of security and privacy, thanks to the embedded encryption engine. At the same time, *Fulmine* executes cryptographic operations, full complex pipelines including CNN-based analytics, and other arbitrary tasks executed on the processors. Summarized, our contributions are:

- We present an energy-efficient cryptographic hardware accelerator with fault side-channel protection at the algorithmic level.
- We tightly integrate the hardware accelerator into the memory hierarchy of a multi-core SoC. We present *Fulmine*, a 65 nm *secure data analytics* SoC, which is designed to be an energy-efficient IoT end-node.
- We extensively evaluate the accelerator on isolated benchmarks and develop and evaluate different use cases of the implementation.

Scientific Contribution

Chapter 4 is primarily based on two publications and builds upon my master's thesis, where I developed and implemented a cryptographic accelerator of *Fulmine*. While the chip design was done during the master's thesis, we received the sample chips at the start of the PhD. As a follow-up of the master thesis and at the beginning of my PhD studies, we characterized the chip, evaluated the efficiency and performance in isolated benchmarks, and developed fully integrated end-to-end use case applications. These evaluations led to two publications that are included in this thesis.

The first paper was published IEEE Transactions on Circuits and Systems I, and received the IEEE Transactions on Circuits and Systems Darlington Best Paper Award:

Francesco Conti, **Robert Schilling**, Pasquale Davide Schiavone, Antonio Pullini, Davide Rossi, Frank Kagan Gürkaynak, Michael Muehlberghuber, Michael Gautschi, Igor Loi, Germain Haugou, Stefan Mangard, and Luca Benini. "An IoT Endpoint System-on-Chip for Secure and Energy-Efficient Near-Sensor Analytics." In: *IEEE Trans. Circuits Syst. I Regul. Pap.* 64-I (2017), pp. 2481–2494. DOI: 10.1109/TCSI.2017.2698019

I am one of the main authors of this paper, besides Francesco Conti who developed the convolutional accelerator of *Fulmine*. Pasquale Davide Schiavone contributed to the Single Instruction Multiple Data (SIMD) instructions of the core. Antonio Pullini contributed to the uDMA engine of the chip. Michael Gautschi contributed to the uncore part of the chip. Davide Rossi and Igor Loi contributed to the backend design of the chip. Germain Haugou provided the software-development kit to program the chip. Frank K. Gürkaynak and Michael Muehlberghuber were my supervisors during my master's thesis at ETH Zurich. Stefan Mangard and Luca Benini supported this work in many discussions and gave feedback to the paper.

Scientific Contribution

The second paper was presented at DATE 2018 in Dresden (Germany).

Robert Schilling, Thomas Unterluggauer, Stefan Mangard, Frank K. Gürkaynak, Michael Muehlberghuber, and Luca Benini. "High speed ASIC implementations of leakage-resilient cryptography." In: *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*. IEEE, 2018, pp. 1259–1264. DOI: 10.23919/DATE.2018.8342208

I am the primary author of this paper that conceptionally introduces the cryptographic hardware architecture of HWCRYPT. All evaluations and the development of the use case applications happened during the PhD studies. Thomas Unterluggauer contributed to the text of this paper, and Stefan Mangard supported this work in many discussions and contributed text to the introduction of this paper. Frank K. Gürkaynak, Michael Muehlberghuber, and Luca Benini were my supervisors during my master's thesis at ETH Zurich.

Outline

The remainder of this chapter is organized as follows: Section 4.1 discusses the threat model of this work, then the requirements for modern IoT end-nodes, and finally presents the two encryption modes with fault security at the algorithmic level. Section 4.3 describes the architecture of the SoC, and the cluster-coupled hardware coprocessors are detailed in Section 4.4. Section 4.5 evaluates the implementation results and overall performance of the design, while Section 4.6 focuses on real-world use cases. In Section 4.7, we compare *Fulmine* with the state-of-the-art in low-power IoT end-nodes. Finally, Section 4.8 concludes the chapter.

4.1 Background

In this section, we first introduce the threat model of this work and then present the requirements of IoT end-nodes with respect to energy efficiency and security. We then present two encryption modes that we integrate into the hardware accelerator. Both encryption schemes protect against Differential Fault Analysis (DFA) and Differential Power Analysis (DPA) at the algorithmic level.

4.1.1 Threat Model

In this work, we assume a powerful attacker that is capable of inducing faults in the cryptographic subsystem of an IoT end-node. This assumption results from the use case of IoT end-nodes, which operate in hostile environments under

the attacker's control. While IoT end-nodes evolved to complex SoCs, we limit faults to the cryptographic accelerator. The attacker's objective is to inject faults during the cryptographic computation, with the goal of performing DFA to recover the encryption keys. We assume other parts of the system to be protected with orthogonal protection mechanisms like data redundancy or Control-Flow Integrity (CFI).

4.1.2 Energy and Security Requirements of IoT End-Nodes

One key driver for the development of the IoT is collecting rich and diverse information streams from sensors, which can then be fed to state-of-the-art learning-based data analytics algorithms. Since IoT end-nodes must work within a tiny power envelope, this fact introduces a significant limit on the volume of data that can be processed and transferred securely, e.g., the size of captured images. To address the first limiting factor, near-sensor smart data analytics is a promising direction. IoT end-nodes must evolve from simple data collectors and brokers into analytics devices that are able to perform a pre-selection of potentially interesting data and/or transform it into a more abstract, higher *information density* form, such as a classification tag. With the burden of *sensemaking* partially shifted from centralized servers to distributed end-nodes, the energy spent on communication and the network load can be minimized effectively, and more information can be extracted, making the IoT truly scalable. However, performing analytics such as feature extraction or classification directly on end-nodes does not address the security concerns. It worsens them: *distilled* data that is stored or sent over the network at several stages of the analytics pipeline is even more privacy-sensitive than the raw data stream [Kha+12; Zha+14].

Protecting sensitive data at the boundary of the on-chip analytics engine is a way to address these security issues. However, cryptographic algorithms come with a significant workload, which can easily be of 100-1000s of processor instructions per encrypted byte [Din+19]. Even further, when countermeasures for fault attacks are needed, the computational workload increases even more, often not practical for devices with a limited power budget. This security workload is added to the computational effort imposed by leading feature extraction and classification algorithms, such as deep CNNs. CNNs are extremely powerful in terms of data analytics and state-of-the-art results in fields such as computer vision, e.g., object detection [KSH12], scene parsing [CMB15], and semantic segmentation tasks [Gir+14], and audio signal analytics [Dah+12] have been demonstrated. While effective, deep CNNs usually necessitate many billions of multiply-accumulate operations, as well as storage of millions of bytes of pre-trained *weights* [He+16].

The combined workload necessary to tackle these two limitations to the development of smarter IoT end-nodes - namely, the necessity for *near-sensor analytics* and that for *security* - is formidable, especially under the limited available power envelope and the tight memory and computational constraints of deeply embedded devices. One possible solution is to augment IoT end-nodes with specialized blocks for compute-intensive data processing and encryption functions

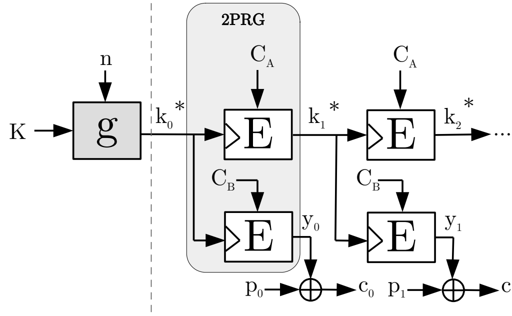


Figure 4.1: 2PRG-based leakage-resilient stream cipher.

while retaining full software programmability to cope with lower computational-intensity tasks. Specialized processing engines should be tightly integrated both with the software-programmable cores and with one another, streamlining the process of data exchange between the different actors as much as possible to minimize the time and energy spent in data exchange. At the same time, to simplify their usage from the developer's perspective, it should be possible to abstract them, integrating them into standard programming models used in software development for IoT-aware platforms.

4.1.3 Leakage-Resilient Encryption with Re-Keying and a 2PRG

One example of leakage-resilient encryption that is secure against DPA and DFA by design is the 2PRG construction combined with a secure re-keying function [Pie09; Sta+10], as shown in Figure 4.1. This scheme consists of two basic parts: (1) a secure re-keying function g, and (2) a leakage-resilient encryption scheme. The re-keying function $g : (K, n) \mapsto k^*$ securely derives a fresh session key k^* from a pre-shared master secret K and a fresh nonce n and hence must be implemented such as to resist both Simple Power Analysis (SPA) and DPA attacks. However, the choice of a fresh nonce n results in a fresh session key k^* to be used for each invocation of the leakage-resilient encryption scheme, thus providing protection against DFA attacks at the algorithmic level. The leakage-resilient encryption scheme, on the other hand, is designed such as to guarantee a bounded data complexity $q = 2$ per key when performing the actual en-/decryption. For this purpose, the leakage-resilient encryption mode in Figure 4.1 utilizes a key update step $k_i^* \mapsto k_{i+1}^*$ to provide a different key for the encryption of each plaintext block p_i. More concretely, the 2PRG primitive used in Figure 4.1 encrypts two constant values C_A and C_B, using a block cipher E with the input key k_i^* to give the next block's key k_{i+1}^* and a pad y_i. The pad y_i is used for en-/decrypting the respective p_i/c_i.

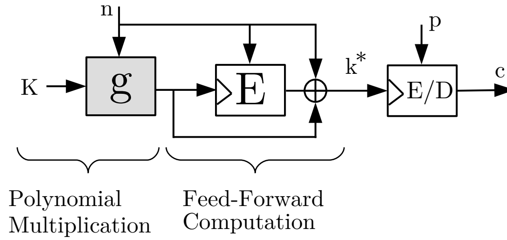


Figure 4.2: Re-keying function based on a polynomial multiplication with a block-cipher-based feed-forward computation.

4.1.4 Re-Keying Function

The re-keying function depicted in Figure 4.1 is used to provide a fresh session key and a secure initialization of the 2PRG. In Section 2.5.3, we presented the basic principles of frequent re-keying. As discussed in this chapter, one prominent example of a re-keying function is a polynomial multiplication in a finite field between the master key and a fresh nonce since it can easily be protected against DPA with classical countermeasures such as masking or shuffling. However, as pointed out in [Bel+15; BFG14; Dob+14; GJ16; Med+11; PM16], the algebraic structure of a multiplication opens the door to combined attacks on g and the encryption. To mitigate this type of attack, Dobraunig et al. proposed adding a block-cipher-based feed-forward computation [Dob+15] after the multiplication, as shown in Figure 4.2. We select this design since still easy to provide DPA protection against the polynomial multiplication.

4.2 Isap - Lightweight Authenticated Encryption

ISAP [Dob+17] is an Encrypt-then-MAC scheme consisting of the two algorithms ISAPENC and ISAPMAC depicted in Figure 4.3 and Figure 4.5 and using the keys K_E and K_A, respectively. ISAPENC uses a nonce n for re-keying during initialization and follows the same principle of continuous key updates during encryption as previous modes. However, in addition, ISAPMAC ensures DPA security for ISAPENC in case of malicious modifications of ciphertexts. In terms of ISAPMAC, the core idea to prevent DPA is to bind the Message Authentication Code (MAC) key k_A^* to the hash y of the ciphertexts. This automatically results in different keys for different ciphertexts. For the security level $\kappa = 128$ bits, ISAPENC and ISAPMAC use KECCAK-f[400] as the permutation p with $a = 20$, $b = 12$, and $c = 12$ rounds, and the rates $r_1 = 144$, $r_2 = 1$, and $r_3 = 144$. With these parameters, injection of the nonce in ISAPENC is 2-limiting, and DPA is prevented. However, a side-channel secure re-keying function g is required for ISAPMAC to derive the session key k_A^* from the pre-shared authentication master key K_A. Here, ISAP uses the GGM-like counterpart named ISAPRK, which is shown in Figure 4.4. The construction of ISAPRK is also embedded in ISAPENC and essentially comprises the nonce-absorbing initialization part in Figure 4.3

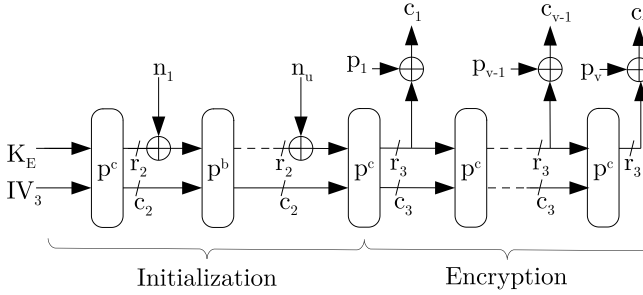


Figure 4.3: ISAPENC with initialization.

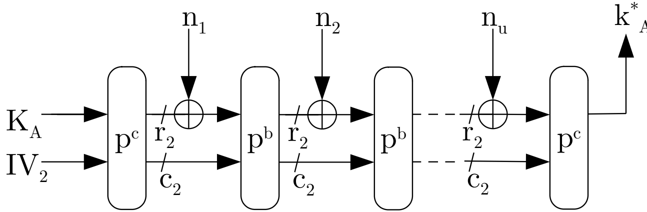


Figure 4.4: ISAPRK.

but truncates the output to give a κ-bit session key.

By design, ISAP prevents DFA at the algorithmic level. Since the nonce changes for every encryption, the initialization phase of ISAPENC always yields a different starting state for encryption, thus protecting the master key K_E against DFA and DPA. Similarly to that, the re-keying function ISAPRK serves the same purpose to protect the master key K_A against DFA attacks by design. Note that fault attacks such as Statistical Fault attacks (SFA) or Statistical Ineffective Fault Attacks (SIFA) may still be possible on ISAP. To address this threat, the re-keying function of ISAP was updated in [Dob+20] to overwrite parts of the computed session key with parts of the nonce. This modification makes the re-keying function hard to invert such that the master key cannot be recovered without additional implementation attacks. However, during the implementation of this hardware accelerator, our implementation does not yet contain this change.

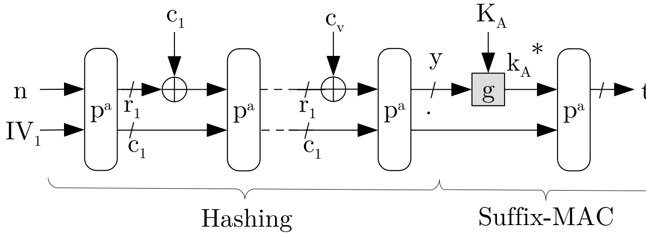


Figure 4.5: IsapMac.

4.3 SoC Architecture

The *Fulmine* multi-core SoC, as shown in Figure 4.6, implements a secure near-sensor data analytics architecture, which leverages highly efficient processors for software programmable signal processing and control, flexible hardware acceleration for cryptographic functions, convolutional neural networks, and a highly optimized subsystem implementing power management and efficient communication and synchronization among cluster resources. The architecture, based on the PULP platform [Ros+15], is organized in two distinct voltage and frequency domains, CLUSTER and SOC, communicating through an AXI4 interconnect and separated by dual-clock FIFOs and level shifters. Two Frequency-Locked Loops (FLLs) are used to generate clocks for the two domains, which rely on separate external voltage regulators for their supply and can be independently power-gated. The FLLs work with a 100 kHz external reference clock and support fast switching between different operating modes within less than 10 reference cycles in the worst case.

The CLUSTER domain is built around six *processing elements* (four general-purpose processors and two flexible accelerators) that share 64 kB of level 1 Tightly-Coupled Data Memory (TCDM), organized in eight word-interleaved SRAM banks. A low-latency logarithmic interconnect [Rah+11] connects all processing elements to the TCDM, enabling fast and efficient communication among the resources of the cluster. The TCDM-interconnect supports single-cycle access from multiple processing elements to the TCDM banks. If two masters attempt to access the same bank in the same clock cycle, one of them is stalled using a starvation-free round-robin arbitration policy. The two hardware accelerators, *Hardware Cryptography Engine* (HWCRYPT) and *Hardware Convolution Engine* (HWCE), can directly access the same TCDM used by the cores. This architecture allows data to be seamlessly exchanged between cores and accelerators without requiring explicit copies and/or point-to-point connections. To avoid a dramatic increase in the area of the TCDM-interconnect, as well as to keep the maximum power envelope in check, the two accelerators share the same set of four physical ports on the interconnect. The two accelerators are used in a time-interleaved fashion, allowing one accelerator full access to the TCDM at a time, which is suitable for data analytics applications where computation can be divided into several separate stages.

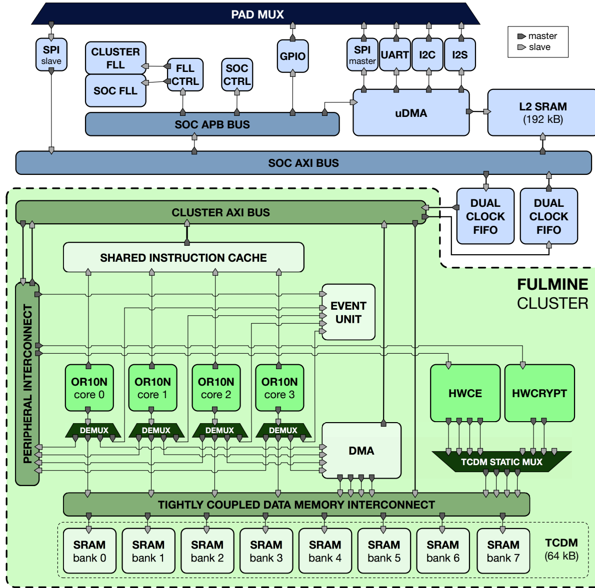


Figure 4.6: *Fulmine* SoC architecture. The SOC domain is shown in shades of blue, the CLUSTER domain in shades of green.

The four OR10N cores are based on an in-order, single-issue, 4-stage pipeline, implementing the OpenRISC [OPE12] Instruction Set Architecture (ISA), improved with extensions for higher throughput and energy efficiency in parallel signal processing workloads [Gau+17]. GCC 4.9 and LLVM 3.7 toolchains are available for the cores, while OpenMP 3.0 is supported on top of the bare-metal parallel runtime. The cores share a single instruction cache of 4 kB of Standard Cell Memory (SCM) [Tem+16] that can increase energy efficiency by up to 30 % compared to an SRAM-based private instruction cache on parallel workloads [Ros+16]. The ISA extensions of the core include general-purpose enhancements automatically inferred by the compiler, such as zero-overhead hardware loops and load and store operations embedding pointer arithmetic and other DSP extensions that can be explicitly included by means of *intrinsic* calls. For example, to increase the number of effective operations per cycle, the core includes SIMD instructions working on 8 bit and 16 bit data, which exploit 32 bit registers as vectors. Furthermore, the core is enhanced with a native dot-product instruction to accelerate computation-intensive classification and signal-processing algorithms. This single-cycle operation supports both 8 bit and 16 bit vectors using two separate datapaths to reduce the timing pressure on the critical path. Fixed point numbers are often used for embedded analytics and signal-processing applications. For this reason, the core has also been extended with single-cycle fixed point instructions including rounded additions, subtractions, multiplications with normalization, and clipping instructions.

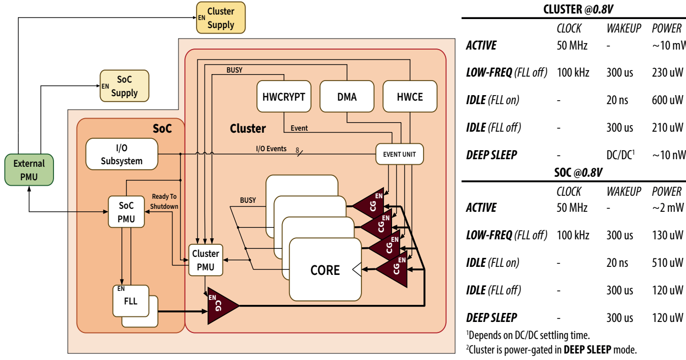
The cluster features a set of peripherals, including a Direct Memory Ac-

cess (DMA) engine, an event unit, and a timer. The processors can access the control registers of the hardware accelerators and of the other peripherals through a memory-mapped interface implemented as a set of private, per-core Demultiplexerss (DEMUXs), and a peripheral interconnect shared among all cores. The peripheral interconnect implements the same architecture of the TCDM-interconnect, featuring a different addressing scheme to provide 4 kB of address map for each peripheral.

The DMA controller available in the cluster is an evolution of the one presented in [Ros+14], and enables fast and flexible communication between the TCDM and the L2 memory through four dedicated ports on the TCDM-interconnect and an AXI4 plug on the cluster bus. In contrast to traditional memory-mapped interfaces, access to the internal DMA programming registers is implemented through a sequence of control words sent to the same address, significantly reducing DMA programming overheads, *i.e.*, less than 10 cycles to initiate a transfer, on average. The DMA supports up to 16 outstanding 1D or 2D transfers to hide L2 memory latency and allows 256 byte bursts on the 64-bit AXI4 interface to guarantee high bandwidth. Once a transfer is completed, the DMA generates an event to the cores that can independently synchronize on any of the enqueued transfers by checking the related transfer ID on the DMA control registers. Synchronization of DMA transfers and hardware-accelerated tasks is hardware-assisted by the event unit. The event unit can also be used to accelerate the typical parallelization patterns of the OpenMP programming model, requiring, for example, only 2 cycles to implement a *barrier*, 8 cycles to open a *critical section*, and 70 cycles to open a *parallel section*. These features are all essential to guarantee high computational efficiency during the execution of complex tasks such as CNNs in *Fulmine*, as detailed in Section 4.4.

The SOC domain contains 192 kB of L2 memory for data and instructions, a 4 kB ROM, a set of peripherals, and a power management unit. Furthermore, the SOC includes a (quad) SPI master, I2C, I2S, UART, GPIOs, a JTAG port for debugging, and a (quad) SPI slave that can be used to access all the SoC internal resources. An I/O DMA subsystem (uDMA) allows to autonomously copy data between the L2 memory and the external interfaces, even when the cluster is in sleep mode. This mechanism allows us to relieve cores from the frequent control of peripherals necessary in many microcontrollers and to implement a double buffering mechanism both between IOs and L2 memory and between L2 memory and TCDM. Therefore, I/O transfers, L2 memory to TCDM transfers, and computation phases can be fully overlapped.

A sophisticated power management architecture distributed between the SOC and CLUSTER domains can completely clock-gate all the resources when idle, as shown in Figure 4.7 (*idle* mode with FLL on). The power manager can also be programmed to put the system in a low-power retentive state by switching down the FLLs and relying on the low-frequency reference clock (*low freq* and *idle* mode). Finally, it can be used to program the external DC/DC converter to fully power-gate the CLUSTER domain. The event unit is responsible for automatically managing the transitions of the cores between the active and idle state. To



CLUSTER @0.8V	CLOCK	WAKEUP	POWER
ACTIVE	50 MHz	-	~10 mW
LOW-FREQ *(FLL off)*	100 kHz	300 us	230 uW
IDLE *(FLL on)*	-	20 ns	600 uW
IDLE *(FLL off)*	-	300 us	210 uW
DEEP SLEEP	-	DC/DC[1]	~10 nW[2]

SOC @0.8V	CLOCK	WAKEUP	POWER
ACTIVE	50 MHz	-	~2 mW
LOW-FREQ *(FLL off)*	100 kHz	300 us	130 uW
IDLE *(FLL on)*	-	20 ns	510 uW
IDLE *(FLL off)*	-	300 us	120 uW
DEEP SLEEP	-	300 us	120 uW

[1]Depends on DC/DC settling time.
[2]Cluster is power-gated in **DEEP SLEEP** mode.

Figure 4.7: *Fulmine* power management architecture and power modes.

execute a *wait-for-event* instruction, the cores try to read a special register in the event unit; this load is kept stalled until the event comes so that the core pipeline is stalled in a known state. After pending transactions and cache refills are complete, the event unit gates the core clock. The clock gating manager gates the cluster clock if the *idle* mode is selected and no engine is busy, or it activates the handshaking mechanism with the external regulator to power gate the cluster if the *deep-sleep* mode is selected. Once the wake-up event reaches the power management unit, the latter reactivates the cluster, then it forwards the event notification to the event unit, waking up the destination core on turn. SOC and CLUSTER events can be generated by all SoC peripherals including GPIOs, by hardware accelerators, by the DMA, by the cluster timer, or by processors through the configuration interface of the event unit mapped in the peripheral interconnect. Figure 4.7 reports all the power modes along with their average wakeup time and power consumption, divided between the CLUSTER and SOC domains. As the CLUSTER and SOC power domains are managed independently, it is possible to transparently put the CLUSTER in *idle*, where it consumes less than 1 mW, when waiting for an event such as the end of an I/O transfer to L2 or an external interrupt that is expected to arrive often. It is possible to partially trade-off wakeup time versus power by deciding whether to keep the FLLs active in *idle* mode: by paying a ∼400 µW cost, wakeup time is reduced to essentially a single clock cycle (20 ns) versus a maximum of 10 reference cycles (∼320 µs) if the FLL is off. The *deep sleep* mode instead enables efficient duty cycling in the case computing bursts are relatively rare by completing power-gating the CLUSTER domain and keeping the SOC domain in a clock-gated, retentive state.

4.4 Cluster-Coupled Accelerator Engines

In this section, we describe in detail the architecture of the two cluster-coupled accelerator engines, HWCRYPT and HWCE. The main purpose of these engines is to provide a performance and efficiency boost on computations, and they were

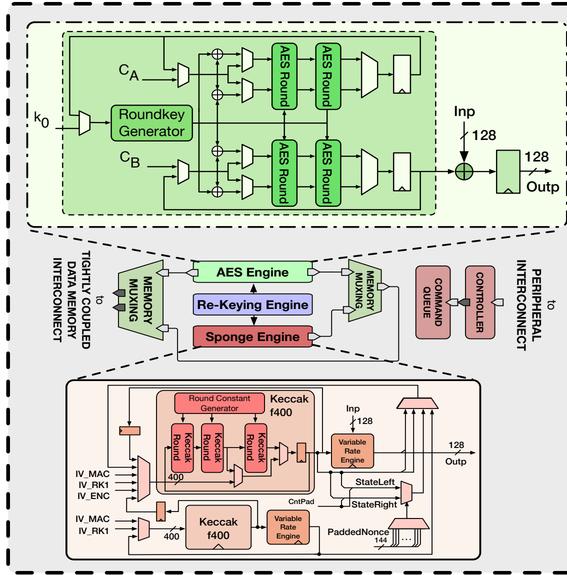


Figure 4.8: HWCRYPT datapath overview, with details of the *AES Engine* and the *Sponge Engine*.

designed to minimize active power, e.g., by using aggressive clock gating on time-multiplexed sub-modules and by making use of latches in place of regular flip-flops to implement most of the internal buffering stages.

The shared-memory nature of the HWCRYPT and HWCE accelerators enables efficient zero-copy data exchange with the cores and the DMA engine, orchestrated by the cluster event unit. This architecture enables complex computation patterns with frequency transfers of data set tiles from/to memory. A typical application running on the *Fulmine* SoC operates conceptually in the following way. First, the input set, e.g., a camera frame, is loaded into the L2 memory from an external I/O interface using the uDMA. The cluster can be left in sleep mode during this phase and woken up only at its conclusion. The input set is then divided into tiles of appropriate dimension so that they can fit in the L1 shared TCDM; one tile is loaded into the cluster, where a set of operations are applied to it either by the software cores or the hardware accelerators. These operations can include en-/decryption and convolutions in hardware, plus any software-implementable filter. The output tiles are then stored back to L2 memory using DMA transfers, and computation continues with the next tile. Operations such as DMA transfers can typically be overlapped with computation by using double buffering to reduce the overall execution time.

4.4.1 Hardware Encryption Engine

The Hardware Encryption Engine (HWCRYPT), as shown in Figure 4.8, implements a dedicated acceleration unit for a variety of cryptographic primitive operations, exploiting the advantages of the shared memory architecture of the SoC. HWCRYPT is based on two parallel cryptographic engines, one implementing based on two high-performant instances of the AES-128 [NIS01] block cipher and the other one based on Isap [Dob+16; Dob+20] with the Keccak-f[400] [Ber+09] permutation as its underlying permutation primitive. Furthermore, it includes a dedicated re-keying engine to initialize the *AES Engine*.

The HWCRYPT utilizes two 32 bit memory ports of the TCDM-interconnect, while an internal interface performs the conversion from 32 bit to the 128 bit format used by the encryption engines. The system is designed so that memory interface bandwidth matches the requirements of all cipher engines. HWCRYPT is programmed and started through a dedicated set of configuration registers, which allows the reconfiguration of a new encryption operation while the HWCRYPT is busy by using a command queue that supports up to four pending operations. The current state of the HWCRYPT can be monitored via status registers. The accelerator supports a flexible event and interrupt system to indicate when one or all operations have finished allowing the processing core to reconfigure the DMA of HWCRYPT. All HWCRYPT blocks are aggressively clock gated, so each component consumes power only when in active use.

AES Engine

The *AES Engine* implements the 2PRG-based stream cipher, as discussed in Figure 4.1, using the two instances of AES-128. Computing the encryption pad for one message block and updating the key requires two invocations of AES-128. Since these computations do not have a data dependency, they can be parallelized. For this reason, the hardware design contains two instances of AES-128, which share the same AES key scheduling unit. Both AES instances are implemented fully in parallel with two AES rounds in the combinational path. Since this architecture can execute two AES rounds within one clock cycle, executing all ten rounds takes five clock cycles plus one cycle of preloading the input register.

Apart from the leakage-resilient cryptographic mode of operation, the *AES Engine* also implements the Electronic-Code-Book (ECB) mode as well as the XEX-based Tweaked-Codebook mode with Ciphertext Stealing (XTS) [Dwo+10]. XTS uses two different encryption keys, one to derive the initial tweak and the other one to encrypt the data. When using the same key for deriving the initial tweak and encrypting the data, the encryption scheme is changed to XOR–Encrypt–XOR (XEX) [Rog04] without implications to the overall security. Furthermore, the accelerator supports the individual execution of a cipher round similar to the Intel AES-NI instructions [Gue10] to boost the software performance of other new AES round-based algorithms [HKR15; WP13].

Re-Keying Engine

The re-keying engine is used to initialize the 2PRG of the *AES Engine* to provide fault security at the algorithmic level, as described in Figure 4.1. This engine that is part of HWCRYPT takes a master key K and a fresh nonce n from the configuration registers and computes a session key $k^* = K \cdot n$, where $\cdot$ denotes a polynomial multiplication in $GF(2^8)[y](y^{16}+1)$ as proposed by Medwed et al. [Med+10]. The polynomial multiplication algorithm is transformed to the operand scan form, which is better suited for a parallel implementation. To achieve a parallel implementation, the re-keying engine contains 16 instances of a $GF(2^8)$ multiplier. Given this partial parallel architecture, computing one session key takes 18 clock cycles, including the time for configuring all registers. The re-keying engine offers side-channel protection by means of additive masking, with a configurable masking order and shuffling of the partial products of the polynomial multiplication. In addition to shuffling the start index as proposed in [Med+10], this architecture supports the shuffling of all partial products resulting in 16! different shuffling sequences. Note that shuffling does not decrease the multiplication throughput of our implementation. However, masking decreases the performance since our architecture only contains one polynomial multiplier. Concretely, computing a masked session key takes $18 \cdot (d+1)$ clock cycles, where d denotes the masking order. Both the masking and shuffling unit get their required randomness from a shared pseudo-random number generator, which security is not scope of this work. Furthermore, the re-keying engine performing polynomial multiplication also supports a post-processing step comprising a feed-forward computation with a block cipher as proposed by Dobraunig et al. [Dob+15], which mitigates key recovery through time-memory trade-off attacks such as in [Dob+14]. To achieve this task, we reuse AES-128 from the *AES Engine*, which adds a static overhead of 20 clock cycles to the re-keying operation.

Sponge Engine

The *Sponge Engine* implements two instances of the KECCAK-f[400] permutation, each based on three permutation rounds. KECCAK-f[400]'s architecture is optimized to match the length of the critical path of the AES-128 engine. Permutations support a flexible configuration of the rate and round parameters. The rate defines how many bits are processed within one permutation operation, and it can be configured from 1 bit to 128 bits in powers of two. This parameter supports a trade-off between security and throughput. The more bits are processed in one permutation call, the higher the throughput - but with a cost regarding the security margin of the permutation. The round parameter configures the number of KECCAK-f[400] rounds applied to the internal state. It can be set up as a multiple of three or for 20 rounds as defined by the specification of KECCAK-f[400]. The two instances of permutations are combined to implement an authenticated encryption scheme ISAP based on a sponge construction with a prefix message authentication code that additionally provides integrity and authenticity on top of confidentiality. In the sponge construction for encryption,

the initial state of the sponge is filled with the key K and the initial vector IV. The engine then absorbs all bits of the nonce n combined with the application of the Keccak-f[400] permutation p to compute a fresh session state. After executing the Keccak-f[400] permutation p, we sequentially squeeze an encryption pad and apply the permutation function to encrypt all plaintext blocks P_i via an XOR operation. Since the nonce is different for every operation, also the session key differs. Consequently, the encryption pads are different for every encryption, thus, protecting against DFA by design. Apart from this favorable mode of operation, the *Sponge Engine* also provides encryption without authentication and direct access to the permutations to allow the software to accelerate any Keccak-f[400]-based algorithm.

4.5 Experimental Evaluation

In this section, we analyze the measured performance and efficiency of our platform on the manufactured *Fulmine* prototype chips fabricated in UMC 65 nm LL 1P8M technology. Figure 4.9 shows a microphotograph of a manufactured *Fulmine* chip, which occupies an area of 2.62 mm×2.62 mm.

4.5.1 System-on-Chip Operating Modes

An important constraint for the design of small, deeply embedded systems such as the *Fulmine* SoC is the maximum supported power envelope. This parameter is important in selecting the system battery and the external DC/DC converter. To maximize energy efficiency, the worst case for the DC/DC converter, *i.e.*, the peak power, should not be too far from the average working power to be delivered. However, a SoC like *Fulmine* can operate in many different conditions: in pure software, with part of the accelerator functionality available, or with both accelerators available. These modes are characterized by very different average switching activities and active power consumption. In pure software mode, it is often desirable to push frequency as much as possible, while when using accelerators, it can be convenient to relax it to improve power consumption. Moreover, some of the internal accelerator datapaths are not easily pipelined, as adding pipeline stages severely hits throughput - this is the case of the HWCRYPT sponge engine (Section 4.4.1), which relies on tight loops of Keccak-f[400] rounds as visible in the datapath in Figure 4.8. Relaxing these paths can improve the overall synthesis results for the rest of the circuit.

Multi-corner multi-mode synthesis and place & route were used to define three operating modes that the developer can statically select for the target application: in the cry-cnn-sw mode, all accelerators and cores can be used. In the kec-cnn-sw mode, the processing cores and part of the accelerators can be used: the HWCE fully, the HWCRYPT limited to Keccak-f[400] primitives. In this mode, the frequency can be pushed significantly further than in the cry-cnn-sw mode. Finally, in the sw mode, only the cores are active, and the

		Technology	Area [mm²]	Power[a] [mW]	Conv. Perf.[b] [GMAC/s]	Conv. Eff.[b] [GMAC/s/W]	Enc. Perf.[c] [Gbit/s]	Enc. Eff.[c] [Gbit/s/W]	SW Perf. [MIPS]	SW Eff. [MIPS/mW]	Eq. Eff.[d] [pJ/op]
AES	Mathew et al. [Mat+14] @ 0.43V, 324MHz	Intel 22nm	2.74e-3	0.43	-	-	0.124	289	-	-	0.19[g]
	Zhang et al. [Zha+16] @ 0.9V, 1.3GHz	TSMC 40nm	4.29e-3	4.39	-	-	0.446	113	-	-	0.49[g]
	Zhao et al. [ZHA15] @ 0.5V, 34MHz	65nm LL	0.013	0.05	-	-	0.027	574	-	-	0.10[g]
	Hocquet et al. [Hoc+11] @ 0.36V, 0.32MHz	65nm LP	0.018	2.5e-4	-	-	3.6e-7	144	-	-	0.39[g]
CNN	Origami [CB17] @ 0.8V, 190MHz	UMC 65nm	3.09	93	37	402	-	-	-	-	0.69[g]
	ShiDianNao [Du+15]	65nm	4.86	320	64	200	-	-	-	-	1.39[g]
	Eyeriss [Che+16] @ 1V, 200MHz	TSMC 65nm LP	12.25	278	23	83	-	-	-	-	3.35[g]
	Jaehy. et al. [Sim+16] @ 1.2V, 125MHz	65nm	16.00	45[e]	32	710[e]	-	-	-	-	0.39[g]
	Park et al. [Par+15] @ 1.2V, 200MHz	65nm	10.00	37[f]	41	1108[f]	-	-	-	-	0.25[g]
IoT	SleepWalker [Bol+13] @ 0.4V, 25MHz	65nm	0.42	0.175	-	-	-	-	25	143	6.99
	Myers et al. [Mye+15] @ 0.4V, 0.7MHz	65nm	3.76	0.008	-	-	-	-	0.7	88	11.4
	Konij. et al. [Kon+16] @ 1.2V, 10MHz	180nm	37.7	0.52	-	-	-	-	10.4	20	50.0
	Mia Wallace [Pul+16] @ 0.65V, 68MHz	UMC 65nm	7.4	9.2	2.41	261	-	-	270	29	22.5
Fulmine	**cry-cnn-sw @ 0.8V, 85MHz**	**UMC 65nm LL**	**6.86**	**24**	**4.64**	**309**	**1.78**	**67**	**333**	**14**	**5.74**
	kec-cnn-sw @ 0.8V, 104MHz			**13**	**6.35**	**465**	**1.6**	**100**	**408**	**31**	
	sw @ 0.8V, 120MHz			**12**	-	-	-	-	**470**	**39**	

[a] Power and efficiency numbers refer to core power, excluding I/Os.
[b] Considering 1 MAC = 2 ops where Gop/s are reported. *Fulmine* numbers refer to the 4bit weights mode.
[c] Refers to AES-128-2PRG for *Fulmine* in CRY-CNN-SW or ISAP for *Fulmine* in KEC-CNN-SW.
[d] Considering the local face detection workload of Section 4.6.2. 1op = 1 OpenRISC equivalent instruction from the set defined in [OPE12].
[e] Weights produced on-chip from a small set of Principal Components Analysis (PCA) bases to save area/power.
No evaluation on the general validity of this approach is presented in [Sim+16].
[f] Performance & power of inference engines only, estimating they are responsible for 20% of total power.
[g] Application Specific Integrated Circuit (ASIC) equivalent efficiency refers to an AES-only or CNN-only equivalent workload.

Table 4.1: Comparison between *Fulmine* and several platforms of the state-of-the-art in encryption, data analytics, and IoT end-nodes.

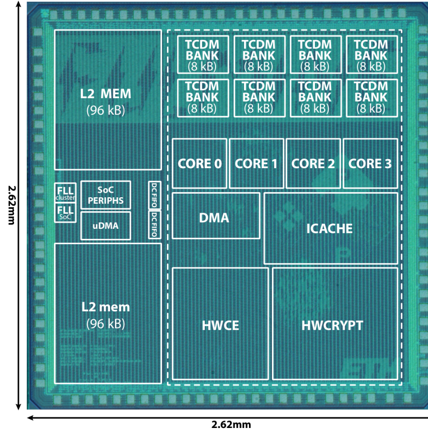


Figure 4.9: *Fulmine* chip microphotograph with main components highlighted.

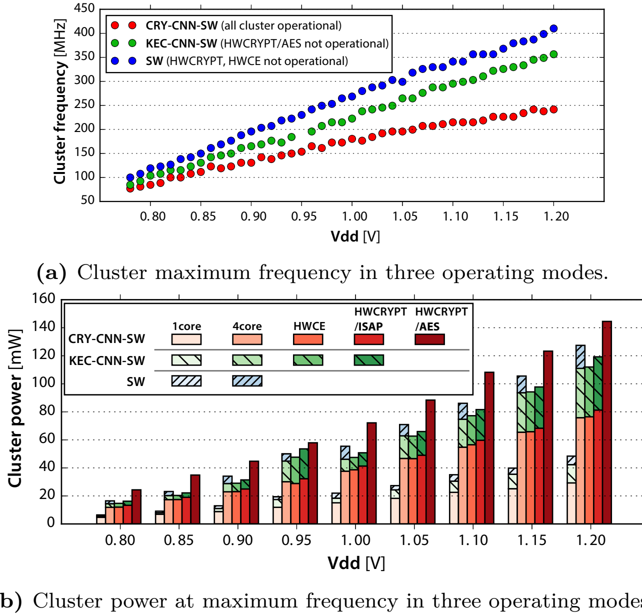
operating frequency can be maximized. Figure 4.10 shows frequency scaling in the three operating modes while varying the cluster operating voltage V_{DD}. The three modes were designed so that at V_{DD} =1.2 V, current consumption under full load is close to 100 mA (*i.e.*, 120 mW of power consumption), as can be seen in Figure 4.10b.

4.5.2 HWCRYPT Performance and Power Evaluation

Due to a throughput-oriented hardware implementation, HWCRYPT achieves a significant acceleration compared to an optimized software implementation running on the OpenRISC cores. To encrypt one 8 kB block of data using the AES-128-ECB mode, HWCRYPT requires ~3100 clock cycles, including the initial configuration of the accelerator. This is a 450× speedup compared to a software implementation on a single core. When parallelizing the software implementation to all four cores, the hardware accelerator still reaches a speedup of 120×. The throughput of HWCRYPT in AES-128-ECB mode is 0.38 cycles per byte (cpb).

Using the AES-128-2PRG configuration, HWCRYPT reaches a throughput of 0.42 cpb. When comparing that to an optimized software implementation on a single core, this speeds up the throughput by a factor of 510× and by a factor 297× when running on four cores. It is important to note that, contrary to the ECB mode, 2PRG encryption cannot be efficiently parallelized in software due to data dependencies.

The authenticated encryption scheme based on Isap achieves a throughput of 0.51 cpb by utilizing both permutation instances in parallel. The first permutation encrypts the data, and the second one is used to compute the message authentication code to provide integrity and authenticity. This performance is achieved in a maximum-rate configuration of 128 bit per permutation call and 20



(a) Cluster maximum frequency in three operating modes.

(b) Cluster power at maximum frequency in three operating modes.

Figure 4.10: Cluster maximum operating frequency and power in the CRY-CNN-SW, KEC-CNN-SW, and SW operating modes. Each set of power bars, from left to right, indicates activity in a different subset of the cluster. KEC-CNN-SW and SW bars show the additional power overhead from running at the higher frequency allowed by these modes.

rounds as specified by KECCAK-f[400]. Reducing the rate and/or increasing the number of invoked permutations decreases the throughput while increasing the security margin.

In Figure 4.11, we present the performance of HWCRYPT in terms of time and energy per byte, while scaling the V_{DD} operating voltage of the cluster. When normalizing these values to the power consumption, we reach a performance of 61 Gbit/s/W for AES-128-2PRG and 137 Gbit/s/W for ISAP-based authenticated encryption, respectively.

4.5.3 Comparison with State-of-the-Art

Table 4.1 compares *Fulmine* with the architectures that define the boundaries of the secure data analytics application space described in Section 4.7. Apart from area, power, and performance, we also use an *equivalent energy efficiency* metric defined as the energy that a platform has to spend to perform an elementary RISC operation[1]. *Fulmine* achieves the highest result on this metric, 5.74 pJ

[1]This is computed as the total energy per instruction on the workload presented in Section 4.6.2, which provides a balanced mix of encryption, convolution, other software-based filters.

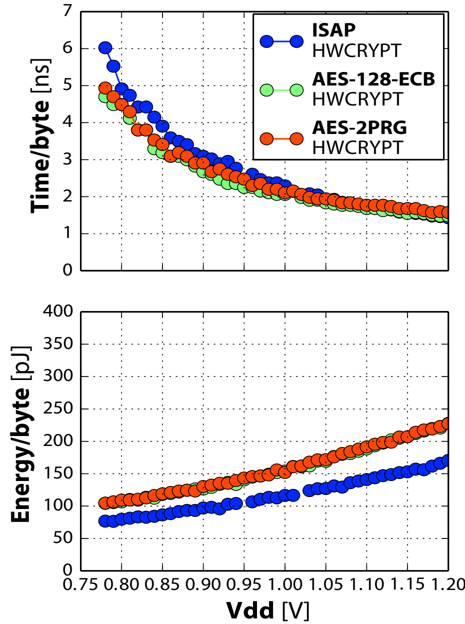


Figure 4.11: Performance and efficiency of the HWCRYPT accelerator in terms of time/energy for elementary output.

per operation, thanks to the cooperation between its three kinds of processing engines. The second-best result is of SleepWalker (6.99 pJ) - but in an operating point where execution takes 89× more time than in the case of *Fulmine*.

Moreover, *Fulmine* provides better area efficiency than what is available in other IoT end-nodes: 32 SleepWalker chips would be needed to achieve the same performance as *Fulmine* in the workload of Section 4.6.2. On the other hand, coupling an efficient IoT microcontroller with external accelerators can theoretically provide an effective solution, but it requires continuous high-bandwidth data exchange from chip-to-chip, which is typically not practical in low-power systems. Conversely, in *Fulmine*, the hardware accelerators are coupled to the cluster cores via the shared L1 memory, and no copy at all is required - only a simple pointer exchange.

For IoT end-nodes, the smaller footprint of a System-on-Chip solution can also provide an advantage with respect to a traditional system on board, which is heavier and bulkier. Taking this reasoning one step further, while it is not always possible to place sensors and computing logic on the same die, the system we propose could be coupled to a sensor in a System-on-Package solution, requiring only a single die-to-die connection. Competing systems listed in Table 4.1 would require the integration of more than two dies on the same package, resulting in a more complex and expensive design.

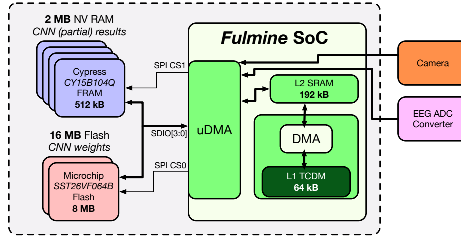


Figure 4.12: A *Fulmine* SoC connected to 16 MB of Flash, 2 MB of Ferroelectric RAM (FRAM), and sensors (the grey area is taken into account for power estimations).

4.6 Use Cases

To evaluate the *Fulmine* SoC in full end-to-end applications, we propose three distinct use cases, which represent a necessarily incomplete selection of possible security- and performance-critical IoT sensor analytics applications. The first use case represents deep-learning-based sensor analytics workloads that are predominantly executed locally on the end-node but require security to access unsafe external memory (secure autonomous aerial surveillance, Section 4.6.1); the second one represents workloads executed only in part on the end-node, which therefore require secured connectivity with an external server (local face detection and remote recognition, Section 4.6.2). Finally, the third use case represents workloads in which, while analytics is performed online, data must also be collected for longer-term monitoring (seizure detection and monitoring, Section 4.6.3).

For our evaluation, we consider the system shown in Figure 4.12. We use two banks (16 MB) of Microchip SST26VF064 bit quad-SPI flash memory to host the weights for a deep CNN as *ResNet-20*; each bank consumes down to 15 µA in standby and a maximum of 15 mA@3.6 V in QPI mode. Moreover, we use 2 MB of non-volatile Cypress CY15B104Q FRAM as a temporary memory for partial results. Four banks are connected in a bit-interleaved fashion to allow access with quad-SPI bandwidth. Both the FRAM and the flash, as well as a camera and an ADC input, are connected to the *Fulmine* uDMA, which can be used to transfer data to/from the SoC L2 memory. The cluster then transfers tiles of the input data to operate on and writes results back to L2 via DMA transfers. We focus on the power spent for actual computation rather than on system power, *i.e.*, we include power spent in transfers from memory used during the computation but exclude data acquisition/transmission[2].

[2]We measured performance on each kernel composing the three applications and for SPI and DMA transfers via RTL simulation, and the related power consumption by direct measurement using an Advantest SoCV93000 integrated circuit tester, encapsulating the target kernel within an infinite loop. Power is measured at two distinct frequencies to obtain leakage and dynamic power density via linear regression. For external memories, we used publicly available data from their datasheets, always considering the worst case.

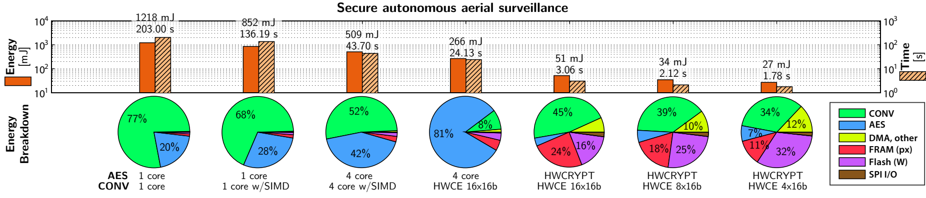


Figure 4.13: Secure autonomous aerial surveillance use case based on a *ResNet-20* CNN [He+16] with AES-2PRG encryption for all weights and partial results. KEC-CNN-SW and CRY-CNN-SW operating modes at V_{DD} =0.8 V.

4.6.1 Secure Autonomous Aerial Surveillance

For the secure autonomous aerial surveillance use case, we consider deploying the system of Figure 4.12 on a low-power nano-Unmanned Aerial Vehicle (UAV) such as a CrazyFlie nano quadcopter [Bit22]. Storms of tens or hundreds of these devices could provide diffused, fully autonomous, and low-energy footprint aerial surveillance. In these vehicles, the power budget for computing is extremely limited (more than 90 % of the battery must be dedicated to the quadrotor engines), and continuous wireless data transmission from cameras is not an option due to its power overhead. Local elaboration and transmission of high-level labeling information provides a more efficient usage of the available power, while also granting greater availability in situations like disaster relief, where wireless propagation might be non-ideal and enable only low-bandwidth communication.

Deployment of state-of-the-art deep CNNs on these devices naturally requires external memory for the storage of weights and partial results. This memory cannot be considered to be secure, as the weights deployed in the flash are an important intellectual property, and UAVs are fully autonomous, therefore, vulnerable to malicious physical attacks. Partial results stored in the FRAM and SPI traffic could be monitored or modified by an external agent, with the purpose of changing the final result classified by the UAV. Strong encryption for weights and partial results can significantly alleviate this issue at the cost of a huge overhead on top of the pure data analytics workload.

Here, we consider a deep *ResNet-20* CNN [He+16] to classify scenes captured from a low-power sensor producing a 224×224 input image. *ResNet-20* has been shown to be effective on CIFAR-10 classification but can also be trained for other complex tasks, and it is in general, a good representative of state-of-the-art CNNs of medium size. It consists of more than 1.35×10^9 operations, a considerable workload for a low-power end-node. External memory is required for both weights (with a footprint of 8.9 MB considering 16 bits of precision) and partial results (with a maximum footprint of 1.5 MB for the output of the first layer). All weights and partial results are en-/decrypted with AES-2PRG; the *Fulmine* cluster is considered the only secure enclave in which decrypted data can reside.

The figure also shows a breakdown of energy consumption regarding kernels (convolution CONV, encryption AES-2PRG), densely connected CNN layers (DENSE), DMA transfers and other parts of the CNN (DMA, other), and external

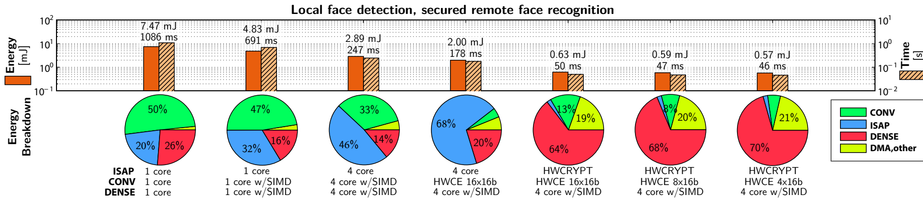


Figure 4.14: Local face detection, secured remote recognition use case based on the 12-net and 24-net CNNs from Li et al. [Li+15] on a 224×224 input image, with full ISAP encryption of the image if a potential face is detected. CRY-CNN-SW operating mode at V_{DD} =0.8 V. We consider that the first stage 12-net classifies 10 % of the input image as containing faces, and that the second stage 24-net is applied only to that fraction.

memories and I/O (FRAM, Flash, SPI I/O)

Figure 4.13 shows the execution time and energy spent at 0.8 V for this compound workload. We exploit the fast frequency switching capabilities of *Fulmine* to dynamically switch from the CRY-CNN-SW operating mode (at 85 MHz) when executing AES-2PRG to the KEC-CNN-SW operating mode (at 104 MHz) when executing other kernels. The figure also shows a breakdown of energy consumption regarding kernels (convolution CONV, encryption AES-2PRG), densely connected CNN layers (DENSE), DMA transfers, and other parts of the CNN (DMA, OTHER), and external memory and I/O (FRAM, FLASH, SPI I/O). In the baseline, where all the workload is run in software on a single core, energy consumption is entirely dominated by convolutions and encryption, with a 4-to-1 ratio between the two. When more features of the *Fulmine* SoC are progressively activated, execution time is decreased by 114× and energy consumption by 45×, down to 27 mJ in total - 3.16 pJ per equivalent operation (defined as an equivalent OpenRISC instruction from [OPE12]). When CNNs use the HWCE with 4 bit weights, and AES-2PRG uses the HWCRYPT, the overall energy breakdown shows that cluster computation is no longer largely dominant, counting for only slightly more than 50 % of the total energy. Additional acceleration would likely require expensive hardware (e.g., more sum-of-products units or more ports in the HWCE) and would yield diminishing returns in terms of energy efficiency.

To concretely estimate whether the results make it feasible to deploy a secure *ResNet-20* on a nano-UAV, consider that a CrazyFlie UAV [Bit22] can fly for up to 7 minutes. Continuous execution of secure *ResNet-20* during this flight time corresponds to a total of 235 iterations in the operating point considered here. This would consume a total of 6.4 J of energy - less than 0.25 % of the 2590 J available in the onboard battery - and the low peak power of 24 mW makes this concretely achievable in an autonomous device.

4.6.2 Local Face Detection with Secured Remote Recognition

Complete on-device computation might not be the most advantageous approach for all applications, particularly for those that can be clearly divided in a lower effort *triggering* stage and a higher effort one that is only seldom executed. A good example is the problem of face recognition. While state-of-the-art face recognition requires a significant workload in the order of billions of operations, e.g., FaceNet [SKP15], the problem can be easily decomposed in two stages: one where the input image is scanned to detect the presence of a face, and another where the detected faces are recognized. The first stage could be run continuously on a low-power wearable device such as a smartwatch, using an external device, e.g., a smartphone or the cloud, to compute the much rarer and much more complex second stage.

We envision *Fulmine* to be integrated into an Ultra-Low Power (ULP) smartwatch platform similar to that presented in Conti et al. [Con+17a]. We consider a similar camera to the one used in Section 4.6.1, producing a 224×224 input image. Face detection is performed locally, using the first two stages (12-net and 24-net) of the multi-stage CNN proposed by Li et al. [Li+15]. If faces are detected by this two-stage CNN, the full input image is encrypted using ISAP and transferred to a coupled smartphone for the recognition phase. The networks are applied to small separate 24×24 windows extracted from the input image; partial results need not be saved from one window to the next. Therefore the CNN does not use any external memory and can rely exclusively on the internal L2.

Figure 4.14 reports the experimental results for the local face detection use case in terms of energy and execution time. Baseline energy is almost evenly spent between convolutions, ISAP encryption, and densely connected CNN layers. Software optimizations such as parallelization, SIMD extensions are much more effective on convolutional and dense layers than they are on the KECCAK-f[400]-based ISAP encryption. Using hardware accelerators essentially reduces the energy cost of convolution and on ISAP to less than 10 % of the total and leads to a 24× speedup and a 13× reduction in energy with respect to the baseline. With all optimizations, face detection takes 0.57 mJ or 5.74 pJ per elementary operation. This face detection could be performed with no interruption for roughly 1.6 days before exhausting the battery charge if we consider a small 4 V 150 mA h lithium-ion polymer battery. Duty cycling, taking advantage of the power management features of the SoC described in Section 4.3, can prolong this time considerably.

4.6.3 Seizure Detection and Secure Long-Term Monitoring

Extraction of semantically relevant information out of biosignals such as Electromyogram (EMG), Electrocardiogram (ECG), and EEG is a potentially huge market for low-power footprint IoT end-nodes. Here, we consider a seizure detection application based on a Support Vector Machine (SVM) trained on energy

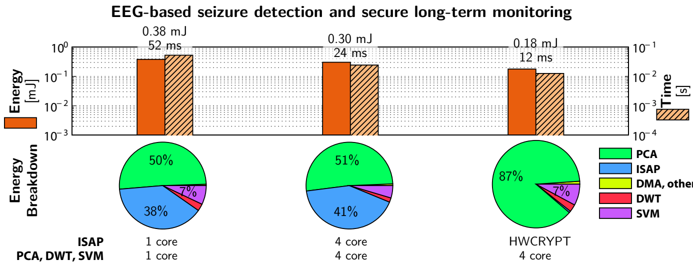


Figure 4.15: Electroencephalogram (EEG)-based seizure detection and secure data collection. CRY-CNN-SW operating mode at V_{DD} =0.8 V.

coefficients extracted from the PCA of a multi-channel EEG signal [Ben+15; Ben+16]. The sampling frequency is 256 Hz with 50 % overlapped windows, *i.e.*, seizure detection is performed every 0.5 s. Starting from a 256-sample window of 23 input EEG channels (represented as 32 bit fixed-point numbers), PCA is applied to extract 9 components, which are then transformed by a Digital Wavelet Transform (DWT) to extract energy coefficients, which are classified by an SVM. For long-term monitoring, the components produced by the PCA have to be collected and sent to the network to be stored or analyzed, which requires encryption due to the sensitivity of this data.

Figure 4.15 shows the results in terms of energy (split down between the various kernels) and execution time. Several components of PCA, like diagonalization, are not amenable to parallelization. Nonetheless, we observe a 2.6× speedup with four cores, excluding the encryption with ISAP. Using the HWCRYPT, the ISAP-based encryption becomes a *transparent* step of the algorithm and essentially disappears from the overall energy breakdown. Therefore, with combined software parallelization and accelerated encryption, an overall 4.3× speedup and 2.1× energy reduction can be achieved. More importantly, the absolute energy consumption of 0.18 mJ (12.7 pJ per operation) means that a typical 2 A h@3.3 V pacemaker battery [MIR04] would suffice for more than 130 million iterations and more than 750 days if used continuously - as for most of the time the *Fulmine* SoC can be in *deep sleep* mode.

4.7 State-of-the-Art and Related Work

In this section, we discuss works related to our contribution, *i.e.*, those proposing low-power hardware Intellectual Property (IP) for encryption or IoT end-node chips that constitute our direct point of comparison. Note that this comparison was done during the time when *Fulmine* was built and was not updated later on.

4.7.1 Low-Power Encryption Hardware IP

Authenticated encryption is a hot topic in the cryptographic community since it adds additional services on top of data confidentiality. AES in the Galois Counter

Mode (AES-GCM) [MV04] is one of the most used authenticated encryption schemes today. For example, Intel added a dedicated finite field multiplication to the AES-NI extension, with a throughput of up to 1.03 cpb [Gue13]. However, solutions of this kind are clearly targeting a different scenario from small, low-power IoT end-nodes.

Only a few IoT-oriented commercial AES controllers are available; an example is the Maxim MAXQ1061 [Max17], claiming up to 20 Mbit/s (power consumption data is not currently disclosed). Research AES accelerators in the sub-100 mW range for the IoT domain have been proposed by Mathew et al. [Mat+14; Mat+15] in Intel 22nm technology, Zhang et al. [Zha+16] in TSMC 40 nm and Zhao et al. [ZHA15] in 65 nm; the latter reaches efficiency up to 620 Gbit/s/W thanks to efficient body biasing and a statistical design flow targeted at reducing worst-case guard bands. A device consuming as little as 0.25 µW for passive RFID encryption has been proposed by Hocquet et al. [Hoc+11]. The main differentiating point between our contribution and these hardware encryption techniques is the tightly coupled integration within a bigger low-power system.

4.7.2 IoT End-Node Architectures

Traditional end-node architectures for the IoT leverage tiny microprocessors, often Cortex-M0 class, to deal with the extremely low-power consumption requirements of applications. Several commercial solutions have been proposed, among the others, by TI [Tex22], STMicroelectronics [STM22], NXP [NXP22], and Ambiq [Amb15], leveraging aggressive duty-cycling and sub-10 µW deep-sleep modes to provide extremely low-power consumption on average. Other recent research platforms also optimize the active state, exploiting near-threshold or sub-threshold operation to improve energy efficiency and reduce power consumption during computation [Bol+13; Mye+15; Pau+16; Roy+16].

Some commercial architectures leverage lightweight software acceleration and optimized DSP libraries to improve performance. The NXP LPC54100 [NXP22] is a commercial platform where a *big* Cortex-M4F core acts as an accelerator for a *little* ultra-low-power Cortex-M0 targeted at always-on applications. From a software viewpoint, some optimized libraries have been developed to efficiently implement crypto algorithms on Cortex-M3 and M4 architectures, given the criticality of this task for IoT applications. Examples of these libraries are SharkSSL [Rea15] and FELICS [Din+19], which are able to encrypt one block of AES-128-ECB in 1066 cycles and 1816 cycles, respectively, both targeting a Cortex-M3. On the other hand, CMSIS [ARM10] is a well-known set of libraries to optimize DSP performance on Cortex-M architectures.

However, even with software-optimized libraries, these tiny micro-controllers are unfortunately not suitable for secure near-sensor analytics applications using state-of-the-art techniques, which typically involve workloads in the orders of billions of operations per second. For this reason, a few recent SoCs couple programmable processors with hardwired accelerators to improve execution speed and energy efficiency in cryptography and other performance-critical tasks. In the field of embedded vision, heterogeneous SoCs of this kind include the one recently

proposed by Renesas [Nak+16], coupling a general-purpose processor with an FPU, a DSP, and a signal processing accelerator. Intel [Wu+15] proposed a 14 nm SoC where a small core with light signal processing acceleration cooperates with a vision processing engine for CNN-based feature extraction and a light encryption engine within a 22 mW power budget. Pullini et al. proposed Mia Wallace, a heterogeneous SoC [Pul+16] coupling four general-purpose processors with a convolutional accelerator. In the field of bio-signals processing, Konijnenburg et al. [Kon+16] proposed a multichannel acquisition system for biosensors, integrating a Cortex-M0 processor and accelerators for digital filtering, sample rate conversion, and sensor timestamping. Lee et al. [LV13] presented a custom bio-signals processor that integrates configurable accelerators for discriminative machine-learning functions (*i.e.*, SVM and active learning), improving energy by up to 145x over the execution on CPU.

Similarly to the presented designs, *Fulmine* is a low-power, heterogeneous Multi-Processor SoC (MPSoC). In contrast to the other architectures presented here, it tackles at the architectural level the challenge of efficient and secure data analytics for IoT end-nodes, while also providing full programmability with sufficient high performance and low-power to sustain the requirements of several near-sensor processing applications.

4.8 Conclusion

This chapter presented *Fulmine*, a 65 nm System-on-Chip targeting the emerging class of smart secure near-sensor data analytics for IoT end-nodes containing the HWCRYPT cryptographic hardware accelerator. HWCRYPT provides different energy-efficient cryptographic modes of operation that provide fault security at the algorithmic level. We achieve this without using aggressive technology or voltage scaling but through the architectural solution of combining cores and accelerators within a single tightly-coupled cluster. The use cases we have proposed show that this approach leads to improvements of more than one order of magnitude in time and energy with respect to a purely software-based solution, with no sacrifice in terms of flexibility. The *Fulmine* SoC enables secure, integrated, and low-power *secure data analytics* directly within the IoT end-node. Without any compromise in terms of security, the proposed SoC enables sensemaking in a budget of a few pJ/op - down to 3.16 pJ/op in one case, or 315 Gop/s/W.

The hardware accelerator within *Fulmine* provides two main engines: an AES-based engine to provide a 2PRG-based stream cipher and a sponge engine for encrypting data with Isap. Both encryption modes provide security against DFA and DPA at the algorithmic level by combining a re-keying function with an encryption mode or even an authentication mode. By using this accelerator, all data that leaves the computing cluster can be encrypted securely, but this can also be used for data that resides within the cluster domain. While this approach still requires manual handling of data, *i.e.*, the programming of the DMA of HWCRYPT, we envision this approach to be used more transparently

in the future. In the following chapters, we follow an enhanced approach that transparently protects certain parts of the system.

5 FIPAC: Control-Flow Protection with ARM Pointer Authentication

While the cryptographic algorithms in the previous chapter are implemented in hardware, they can also be implemented in software. One crucial property or assumption for the correct execution of software is that all program instructions execute in the correct order, i.e., there is a valid control-flow. If the control-flow of a program is violated, *i.e.*, there is a control-flow attack, it can lead to bypasses of arbitrary security countermeasures up to full remote code execution.

Control-flow attacks can be performed purely in software by exploiting a memory vulnerability to modify code pointers or return addresses. This strategy allows an adversary to perform powerful Turing-complete attacks, such as Return-Oriented Programming (ROP) [Sha07] or Jump-Oriented Programming (JOP) [Che+10]. These software-based control-flow attacks have successfully been used to attack many devices, from embedded devices to secure enclaves [HHF09; Jal+20; Lee+17].

When also considering fault attacks, as we do in this thesis, the attack surface of control-flow hijacks increases. Faults can manipulate the control-flow at a much finer granularity rather than only on a course-grained indirect function level, as it is the case for software-based control-flow attacks. Direct branches, calls, or even only a few instructions are a target of fault-based control-flow attacks, allowing an attacker to jump arbitrarily within the program. Consequently, faults on the control-flow are used to bypass security defenses at a much finer granularity [Fre11; Goo15; NCC; Tat+18; TSW16].

To solve this problem and to protect the software execution against control-flow attacks, Control-Flow Integrity (CFI) aims to be a generic solution. Depending on the threat model, CFI can exist at a course-grained indirect function level to

protect against software-based control-flow attacks or on a much finer granularity, e.g., on basic block or even on the instruction level to cover fault-based control-flow hijacks. Section 3.2 details CFI at different granularities for different threat models. However, when it comes to combined attacks, *i.e.*, a software vulnerability is combined with a fault-based control-flow attack, only little research exists. While there are protection mechanisms in this area [Cle+17; Wer+18], they require intrusive hardware changes to the processor architecture and are, therefore, not practical for general use.

This, however, stays in contrast with the rise and large-scale deployment of ARM-based devices [ARMd]. ARM-based microarchitectures can only be modified with expensive architectural licenses, which leaves many devices in hostile environments remaining unprotected against a variety of control-flow attacks. Currently, there are no protection mechanisms against software- *and* fault-based control-flow attacks available that only require minimal hardware changes or even can be designed with a standard unmodified instruction set.

When we started working on this topic, ARM introduced a new hardware primitive named ARM Pointer Authentication (ARM PA) [ARMa], which allows the software to sign and authenticate pointers and was shipping in the first hardware designs, such as the Apple M1 processor [App21]. This hardware primitive is originally intended to protect special code pointers, e.g., the return address on the stack, against unintended manipulations, providing control-flow protection on a very coarse-grained level. We asked ourselves: Can we use this hardware primitive to develop a much stronger protection scheme that protects the control-flow of a program against software *and* fault attacks?

Contribution

We present FIPAC, a software-based CFI scheme protecting the execution at basic block granularity of ARM devices against software *and* fault attacks. FIPAC's threat model considers an attacker hijacking the control-flow on basic block level, independent of the attack methodology. We address this threat model and protect the control-flow by implementing a basic block level CFI protection scheme using a keyed state update that is resistant to memory bugs. FIPAC cryptographically links the sequence of basic blocks at compile-time and verifies the executed sequence at runtime. We exploit ARM Pointer Authentication of ARMv8.6-A for efficient linking and verification. We provide an LLVM-based toolchain to protect programs without user interaction against control-flow attacks on basic block level. We validate the prototype using a simulator supporting ARMv8.6-A. To evaluate the runtime performance of FIPAC, we emulate the overheads of ARM PA instructions and run SPEC 2017 and other embedded benchmarks on existing hardware. Moreover, we provide a security evaluation and discuss different security policies. Summarized, our contributions are:

- We present an efficient basic block granular CFI protection scheme for ARM-based systems protecting the control-flow against fault *and* software attacks.

- We present a prototype implementation exploiting the ARM Pointer Authentication of the ARMv8.6-A.
- We provide a custom open-source[1] LLVM-based toolchain to automatically instrument and protect arbitrary programs.
- We perform a functional and performance evaluation based on SPEC 2017 and other embedded benchmarks and discuss different security policies.

Scientific Contribution

Chapter 5 is primarily based on the following publication that was presented at COSADE 2022 in Leuven (Belgium).

Robert Schilling, Pascal Nasahl, and Stefan Mangard. "FIPAC: Thwarting Fault- and Software-Induced Control-Flow Attacks with ARM Pointer Authentication." In: *Constructive Side-Channel Analysis and Secure Design - 13th International Workshop, COSADE 2022, Leuven, Belgium, April 11-12, 2022, Proceedings*. Springer, 2022, pp. 100–124. DOI: 10.1007/978-3-030-99766-3_5

I am the main author of this paper, wrote the majority of the text, developed the toolchain prototype, and performed all the experiments. Pascal Nasahl contributed to the text and to the evaluation setup of the SPEC benchmark. Stefan Mangard supported the project in many discussions.

Outline

This chapter is structured as follows. Section 5.1 introduces the details of ARM PA, which was originally designed to counteract software-based control-flow attacks. Section 5.2 discusses the threat model we are considering in this chapter and analyzes the state-of-the art. Section 5.3 presents the design of FIPAC and discusses its main requirements. Section 5.4 discusses the prototype implementation of FIPAC exploiting existing architectural features of the ARMv8 instruction set. We discuss the security guarantees of FIPAC and present the performance numbers of the prototype implementation in Section 5.5. Section 5.6 presents existing attacks and shows how FIPAC mitigates them. Section 5.7 discusses how FIPAC can be used to protect compile-time known data. Finally, Section 5.8 mentions possible future work, and Section 5.9 concludes this chapter.

5.1 ARM Pointer Authentication

ARM Pointer Authentication (ARM PA) is a hardware feature introduced with ARMv8.3A [ARMa] and updated in ARMv8.6A [ARMb]. This extension provides new instructions to cryptographically sign and authenticate data. These

[1] Available at https://github.com/Fipac/Fipac

instructions derive a Message Authentication Code (MAC) using a secret key, a 64-bit modifier, and the value of a provided register, e.g., an address stored in a pointer. A fraction of this MAC, called the Pointer Authentication Code (PAC), is then stored in the upper bits of the provided register. By using the authentication instructions, the authenticity of the MAC and the data in the register can then be verified. The size of the PAC depends on the configuration of the virtual address system and can range from 11 to 31 bits.

ARM PA can be used for different purposes, but one of the first use cases was return address signing, developed by Qualcomm [Qua17]. During the function prologue, they sign the function return address that is stored in the link register before storing it to the stack. The function epilogue then uses the authenticate instruction of ARM PA before doing the function return. This instruction recomputes and validates the MAC that is stored in the return address. If the operation succeeds, the MAC is stripped, and the return is performed accordingly. However, if the MAC verification fails, e.g., because the return address was overwritten on the stack due to a buffer overflow, the pointer gets invalidated. When using such an invalid pointer, *i.e.*, when doing the return, the system traps and detects the stack-smashing attack. This mechanism is already built into upstream compilers of GCC [Fre20; GCC23] and LLVM [LLV19; LLV23].

While the previous mechanism of ARM PA is already available in upstream compilers, this primitive is also used in research projects. Most prominently, PARTS [Lil+19] uses ARM PA to sign arbitrary code- and data pointers, providing pointer integrity in the system. Before using a signed pointer, the pointer is authenticated and validated, thus prohibiting the use of a manipulated or forged pointer. PACmem [Li+22] or PACSafe [HZH22] use the primitives from ARM PA to implement spatial and temporal memory safety for C or C++ programs. More recently, ARM PA has been used to protect data that is spilled on the stack, providing data integrity for spilled registers [Fan+22]. All in common, these protection mechanisms use ARM PA to extend the scope of protection within the software threat model.

5.2 Threat Model and Attack Scenario

This section presents the threat model we consider, shows how it bypasses existing Software CFI (SCFI) and Fault CFI (FCFI) protection schemes, and then states the required properties for secure SCFI protection schemes. Eventually, we discuss the required properties for a secure CFI scheme protecting against software and fault attackers.

5.2.1 Threat Model

FIPAC considers an attacker performing software- and fault-based attacks with the goal to redirect the control-flow. This attacker aims to hijack direct or indirect control-flow transfers, *i.e.*, the threat model of FIPAC covers all transfers between basic blocks of the program, *i.e.*, direct, indirect, and conditional branches, direct

and indirect calls, and arbitrary jumps. We consider attacks on the control-flow independent of the methodology, *i.e.*, we cover physical- and software-induced fault attacks on the control-flow as well as software-based control-flow attacks. We expect the CFI protection to detect control-flow deviations to avoid further exploitation. The detection rather than its prevention aligns with threat models of related FCFI protection schemes. The attacker has binary access and can read all instructions and data. This threat model includes software attackers using this information to exploit a memory bug to conduct a control-flow hijack, e.g., manipulating code pointers to perform ROP or JOP. We assume ARM Pointer Authentication to be cryptographically secure and that its keys are isolated from user applications.

We only consider control-flow hijacks on the Control-Flow Graph (CFG)'s edges, so we exclude attacks within a basic block, e.g., instruction skips. However, our assumed threat model aligns with several real-world exploits [Car+19; Goo15; NT] hijacking the control-flow at these edges. Nevertheless, as security-critical code can still require stronger protection, we discuss the usage of FIPAC at instruction granularity in Section 5.8. Data-Oriented Programming (DOP) or faults on the data or the computation are not in the scope, including data used during a conditional branch or data used in cryptographic algorithms. To protect them, it requires orthogonal defenses, e.g., data encoding or instruction replication. We discuss the protection of conditional branches within a state-based CFI protection scheme in Chapter 7. For full fault protection, a combination of both the protection of data and processing and control-flow protection like FIPAC is required.

5.2.2 Attack Scenario

We show how existing CFI protection schemes can be bypassed in the stated threat model. The details of these CFI countermeasures are already discussed in Section 3.2.1 and Section 3.2.2, respectively.

Bypassing SCFI

Most SCFI protection schemes [Aba+05; Eva+15; Lil+19; Lil+21; Tic+14; ZS13], as presented in Section 3.2.1, do not consider faults in their threat model and, therefore, can be bypassed with a single fault. As the programs' code section is immutable, SCFI protection schemes only protect indirect control-flow transfers but not direct calls and other branches. Hence, a targeted fault to the code segment of a program or directly within the execution, e.g., a fault on the program counter or the immediate value of a direct call, cannot be detected by SCFI.

Bypassing FCFI

The threat models of software-based FCFI protection schemes do not consider classical software attackers. Contrary to SCFI [Aba+05] (cf. Section 3.2.1), where memory is considered to be vulnerable, typical FCFI protection schemes,

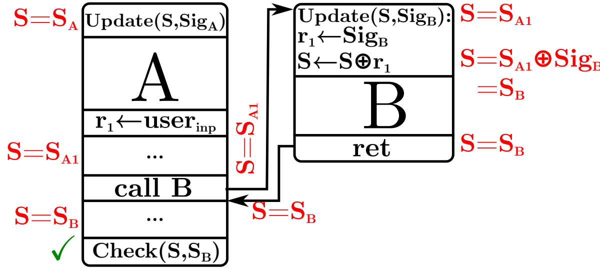


Figure 5.1: Valid control-flow.

as discussed in Section 3.2.2, do not include this in their threat model. An attacker exploiting a memory bug can tamper the CFI state, which is maintained in software. As the state update function is known, an attacker-controlled CFI state can be crafted. Even a naïve combination of SCFI and FCFI, secure in their threat model, can be bypassed (Section 5.5.2) with a combined software- and fault-based control-flow attack. To highlight the conceptual weaknesses of FCFI protection schemes, we demonstrate an attack bypassing FCFI with its state update function. Such a state update function is presented in Algorithm 1 in Section 3.2.2 and is similarly used in many software-based FCFI protection schemes [OSM02]. They compute their CFI states in software and load them into a register at some point in the program. The goal is to exploit this instruction sequence of the state update to manipulate the CFI state to an attacker-defined value, *i.e.*, bypassing CFI.

In the following, we discuss the attacker scenario for a control-flow hijack. Without an attacker, Figure 5.1 shows a valid control-flow transfer, where basic block `A` calls `B`. When entering `B`, the state update function updates the global state S to the beginning state S_B by XORing Sig_B to S. After returning from basic block `B`, a CFI check verifies that S equals the pre-computed state S_B.

In Figure 5.2, we consider an attacker redirecting the control-flow of the call from basic block `B` to `C`. At the beginning of basic block `C`, the state update XORs the current state S with the signature `C`. As this state $S = S_C$ deviates from the pre-computed state S_B, the control-flow hijack can be detected in the final check.

Figure 5.3 shows a successful attack on the control-flow, bypassing FCFI. The attacker controls register r_1, e.g., it is used to store user input, or it is modified due to a memory bug or fault. The adversary again redirects the control-flow from basic block `B` to `C` but omits the signature load to r_1. Since r_1 is controlled by the attacker, who knows all states and signatures, the final state of `C` can be forged to match the end state of `B`. Eventually, the final CFI check in basic block `A` cannot detect the control-flow hijack. Note that the control-flow redirect in Figure 5.2 or Figure 5.3 can either be performed with a software-based control-flow attack or by fault-based one.

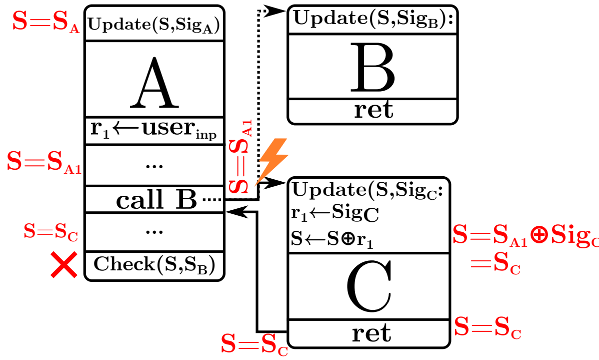


Figure 5.2: Detectable control-flow attack.

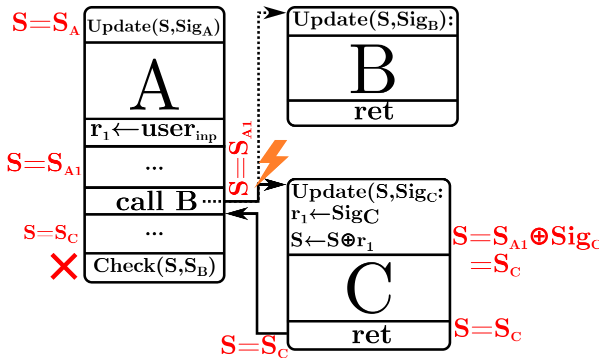


Figure 5.3: Successful control-flow attack.

5.2.3 CFI against Software- and Fault-Based Control-Flow Attacks

To protect the system against software *and* fault attacks and to enable large-scale usage, CFI protection schemes need to fulfill the following requirements:

1. The defense needs to enforce the CFI at a fine granularity, *i.e.*, at least on basic block level, to protect from a fault attacker.

2. The proper selection of the CFI state update function is essential, as it directly influences the security of the CFI scheme. Choosing a weak state update function, e.g., an XOR, allows an attacker to bypass the protection. An attacker reading the binary is able to recompute the CFI states for all locations of the program and can bypass the CFI scheme, as discussed in Section 5.2.2. Thus, it is required that CFI states are not known to the attacker and cannot be recomputed even when having access to the binary. Furthermore, the state update function must be accumulating, meaning that the next CFI state depends on the value of the previous CFI state.

3. The protection should not require hardware changes and can be implemented in software to make the protection deployable for a wide range of devices.

4. To support legacy codebases and to enable easy deployment, the protection must be applied automatically, *i.e.*, during compilation, and must not require source code changes.

Since such a countermeasure provides protection against software- *and* fault-based control-flow attacks, we denote it as a *Software-Fault CFI (SFCFI)* protection scheme. Previous fine-grained CFI protection with keyed update functions,

which can detect software- and fault-based control-flow hijacks, require expensive hardware changes and are unsuitable for commodity devices. For example, Sponge-Based Control-Flow Protection (SCFP) [Wer+18] performs software encryption on the instruction level and dynamic decryption during the software execution. During the compile-time, the toolchain automatically encrypts the program on instruction granularity. At runtime, the system decrypts the program but keeps the previous instruction history in mind, *i.e.*, there is authentic decryption of all instructions. Only when executing all previous instructions in the correct order the decryption of the next instruction succeeds. To deal with the performance overhead, SCFP adds a dedicated pipeline stage to the prototype implementation of their RISC-V processor. This pipeline stage is responsible for decrypting the instruction stream on instruction granularity and performing special handling of control-flow instructions.

A similar approach is called SOFIA [Cle+16], which provides cryptographically-supported CFI protection on instruction-level. However, their design splits up the encryption and verification rather than using an authentic encryption that incorporates the instruction history. To cope with the performance penalty, the prototype implementation again requires intrusive hardware changes on their SPARC processor platform.

Unfortunately, both schemes require intrusive hardware changes in the processor and are therefore inapplicable for large-scale deployment. Hence, there is a need for efficient CFI protection schemes considering software- and fault-based control-flow attacks, which do not require hardware changes.

5.3 Design of FIPAC

This section presents FIPAC, an efficient software-based CFI solution for ARM-based devices, fulfilling the abovementioned requirements. We first show the state-based CFI concept based on the work of Wilken and Shen [WS88; WS90] and then discuss how indirect calls are protected. Finally, we discuss the selection of the state update function and the check placement in the program.

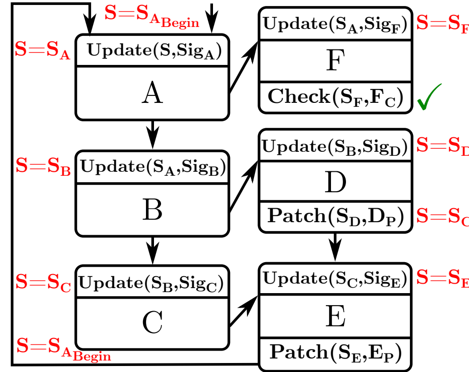


Figure 5.4: Justifying signature for control-flow merges.

5.3.1 Signature-Based Control-Flow Integrity

FIPAC is a state-based CFI protection scheme where every basic block in the program corresponds to a well-defined CFI state. This state is maintained globally through the program execution. The CFI state is checked to match the expected state at certain program locations, indicating that no control-flow error occurred. To consider the history of the execution-flow, the next CFI state is linked with the previous one, allowing FIPAC to enforce the CFG.

Programs do not have a linear control-flow but contain control-flow transfers, such as conditional branches, loops, or calls. Depending on which program path is executed, the CFI state for a certain basic block differs since it has more than one predecessor. When the control-flow merges, *i.e.*, for conditional branches, two different paths of CFI states merge and would turn into a state collision. To avoid that, we adopt generalized path signature analysis from Wilken and Shen and insert justifying signatures for correction. Figure 5.4 shows a conditional branch, where the control-flow merges in basic block `E`, and a loop, which control-flow merges in `A`. At the end of basic block `D`, there is a state patch with $\mathrm{D_p}$, ensuring the CFI state at the beginning of basic block `E` is the same, whether coming from basic block `C` or `D`. These update values are stored in the program code and are applied at runtime. The same approach is used to deal with loops, which effectively boils down to conditional branches. E.g., basic block `E` jumps back to `A`, forming a loop. Thus, a patch $\mathrm{E_p}$ is inserted at the end of basic block `E`, correcting the CFI state to $S_{A_{Begin}}$. At the end of basic block `F`, a check compares the actual state with the expected value $\mathrm{F_C}$.

Direct Calls

Direct function calls require special handling of the justifying signatures. In Figure 5.5, function `A` directly calls function `B`. To support calling `B` from multiple call sites, the beginning state of `B` always needs to be the same. Thus, we apply a justifying signature at the call site before the direct call, transforming the call

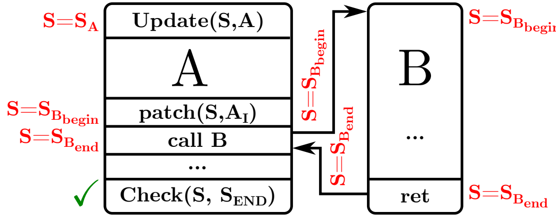


Figure 5.5: CFI state patch for direct calls.

site's CFI state to the beginning state of function B. In this example, the patch operation before the call transforms the CFI state of A to the beginning state of function B. When returning, the CFI state continues with the end state of the called function, here $S_{B_{end}}$. Note this approach of placing justifying signatures is similar to conditional branches, where at a control-flow merge, all edges, except one, have a state update. Although we place a justifying signature before every function call, in theory, one function call would also work without a state update. However, deciding which call site omits the signature update is a challenging problem during compilation and therefore is left out. Additionally, this can also lead to problems when dealing with external libraries.

Indirect Calls

Indirect calls require special handling of signatures, which is not covered by the work of Wilken and Shen. Determining the exact function that is being called during the indirect call is not always possible at compile-time. Indirect calls can also call different functions from the same call site, e.g., a function pointer is given as a parameter to the function and is then called inside. The best that FIPAC can do is to determine a possibly over-approximated set of potential call targets and enforce that the indirect call can only call one of them. Then, FIPAC enforces that the indirect call can only call one of these functions. Figure 5.6 shows the patching for indirect calls and the interaction with direct calls.

To provide the CFI for indirect calls, FIPAC determines an intermediate CFI state S_I for every set of indirectly called functions. This can also lead to merging sets if the same function is called indirectly from different call sites. When performing an indirect call, the call site A, in ①, first patches its state S_A to an intermediate state $S_{I_{begin}}$, the same for all possible call targets of this indirect call. In ②, the indirect call is performed. At the beginning of the indirectly called function B, we transform the state, in ③, from the intermediate state $S_{I_{begin}}$ to the beginning state of $S_{B_{begin}}$. Furthermore, in ④, we set up the patch value used for the function return. We jump over the direct call entry in ⑤ and continue the execution of B until the return patch in ⑥. This patch transforms the end state $S_{B_{End}}$ of B to the common intermediate return state $S_{I_{End}}$ followed by a return. The caller A uses the pre-call signature S_A, which was saved, for a state update in ⑦, to transform the intermediate return state to a unique state for A. Note the call site could simply continue with the execution using the state $S_{I_{end}}$.

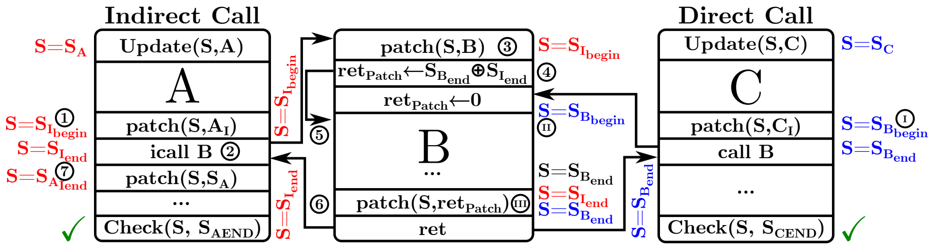


Figure 5.6: CFI state patch for indirect calls.

However, this would introduce undetectable control-flow vulnerabilities between different indirect call sites of the same function. Therefore, the patch with S_A is necessary to avoid different call sites continuing with the same signature and ensure that the function was actually called. The call site continues with the execution using the state $S_{A_{I_{end}}} = S_{I_{end}} \oplus S_A$, different for every call site.

Since any function must be callable with direct or indirect calls, the handling of indirect and direct calls interacts. On the right of Figure 5.6, we show how `C` calls `B` directly. In Ⓘ, a justifying signature is applied to transform `C`'s CFI state to the beginning state $S_{B_{begin}}$ of `B`. The direct call does not jump to the beginning of `B`. Instead, it jumps to a dedicated entry point setting up the return patch ret_{Patch} to be zero (Ⅱ), and continues with the execution of `B`. At the end of the function in Ⅲ, the return patch ret_{Patch} is applied. Since the patch value is zero, this statement does not affect the state which remains $S_{B_{end}}$. After the return, the call site then continues with the execution using the state $S_{B_{end}}$.

5.3.2 State Updates with ARM Pointer Authentication

As discussed in Section 5.2.3, the state must not be computable by the attacker and must depend on all previous CFI states. FIPAC uses a chained cryptographic MAC for the state update function to solve this problem. Thereby, we bind the security of FIPAC to a secret cryptographic key, which is unknown to the attacker and isolated by the Operating System (OS). Only knowing this secret key allows an attacker to recompute the CFI states and mount an attack. As discussed in the threat model, it requires that the state update function is a keyed cryptographic primitive to mitigate the recomputation of the CFI states by an attacker with read access to the binary. To efficiently implement such a cryptographic function, we exploit ARM PA, introduced in ARMv8.3-A and updated in ARMv8.6-A [ARM20] with *EnhancedPAC2* and *FPAC* [ARM20]. It is designed to cryptographically sign pointers with a PAC and verify their integrity before using it [Qua17]. The PAC is computed as the MAC over the pointer and a modifier using the QARMA [Ava17] tweakable block cipher. Although pointers are 64-bit values, the size of the virtual address space limits the actual size of the pointer values. In AArch64 Linux, the virtual address space is typically configured for 39 or 48-bit [Mar20], leaving the upper bits unused. The `PACIA`

instruction of ARM PAhe PAC value in the unused upper bit, thus having no storage overhead.

To use Pointer Authentication, ARMv8.6-A was extended for computing and verifying a PAC. The instructions `PACI*` and `PACD*` use the destination register as input, the source register as a modifier, and XOR the PAC in the upper bits of the destination register. The PAC can be verified by using the `AUTI*` and `AUTD*` instructions. On a successful verification, the PAC is removed from the address, and the pointer can be used. If the verification fails, `AUT*` instructions trap (this is different from ARMv8.3-A, which only sets an error bit).

the `PACIA` instruction of ARM PAused in Apple's iOS [App20], as well as in the Linux kernel [Rut17]. Moreover, upstream compilers such as LLVM [LLV19; LLV23] and GCC [Fre20; GCC23] provide support for PAC. Recently, also RISC-V designs were extended to provide similar mechanisms to ARM Pointer Authentication [Sug20].

This work uses the `PACIA` instruction of ARM PA to implement the state update function rather than sign pointers. This extension fulfills the requirements needed for the state update. It uses a keyed mechanism and brings in the accumulating functionality required to link subsequent states. We further discuss the use of the `PACIA` instruction of ARM PA in Section 5.4.1.

5.3.3 Placement of Checks

Although the CFI check placement is essential for the security of the CFI scheme, there is no general solution for the correct placement. However, at minimum, there needs to be one check at the end of the program. For programs that do not return, *i.e.*, server programs, at least one CFI check in the main event loop is needed. This strategy, however, has the longest detection latency and the worst detection probability. To reduce the detection latency and improve the detection probability of CFI errors, more CFI checks are required. However, the granularity is a trade-off between overheads and security. The more checks inserted, the more overhead, but also better detection probability and lower latency. At worst, a check is placed at the end of every basic block, yielding the best security but the worst runtime and code performance. In between, there exist arbitrary policies with different trade-offs. For example, a generic policy places a CFI check at the end of each function. Even fully custom strategies for placing checks are possible. With the help of dynamic runtime profiling, a compiler can place the checks more efficiently. E.g., a policy can place a check after every 100th basic block.

5.4 Implementation

In this section, we discuss the prototype of FIPAC based on ARMv8.6-A and discuss the custom LLVM-based toolchain. First, we discuss the overall implementation based on ARM Pointer Authentication. Then, we present our custom toolchain based on a modified LLVM compiler and a post-processing tool deriving all signature values of the program.

5.4.1 System Implementation

FIPAC computes a rolling CFI state throughout the program's execution implemented in software on top of ARMv8.6-A without hardware changes. FIPAC exploits the ARM PA instruction set extension to implement the cryptographic state update function. The `PACI*` and `PACD*` instructions cryptographically compute a MAC over a pointer and a modifier register and store the result in the upper bits of the pointer. In ARMv8.6-A, these instructions do not simply replace the upper bits of the pointer with the computed MAC but instead, XOR them to the existing upper bits. Algorithm 2 shows the simplified behavior of the `PACIA` instruction ignoring that the configuration bit 55 is excluded from the PAC.

Key Management

ARM Pointer Authentication includes five keys containing the generic key, the instruction, and data keys `A` and `B`. By utilizing `PACIA`, FIPAC uses the `APIAKey`, which is managed in the kernel (`EL1`) and not accessible from user mode (`EL0`) [Qua17]. To provide CFI protection with FIPAC for the kernel, the key management can be delegated to a higher privilege level, e.g., `EL2`. As ARM PA instructions do not differentiate privilege levels, these instructions can be used in `EL0` and `EL1`. To prevent cross-EL attacks [Aza19], FIPAC-protected user and kernel tasks can either use different keys for each privilege level (e.g., `APIAKey` for `EL0` and `APIBKey` for `EL1`), or the key manager in `EL2` could swap the keys on mode transitions. As the key needs to be known at compile-time, the prototype implementation of FIPAC statically configures the `APIAKey` in a kernel module in `EL1`. We discuss the dynamic configuration of the ARM PA keys in Section 5.8.

Interrupts

FIPAC supports interrupts and CFI interactions without any change. When an interrupt diverts the control-flow to the kernel, it saves all registers of the user application, including the current CFI state. The CFI state is restored after resuming from the interrupt, allowing the program to continue.

Algorithm 2 Simplified behavior of `PACIA` in the 15-bit configuration.

1: **function** PACIA(Xd, Xm)
2: PAC[63:0] $\leftarrow$ ComputePAC(Xd[47:0], Xm, K)
3: Xd[63:48] $\leftarrow$ Xd[63:48] $\oplus$ PAC[63:48]
4: Xd[47:0] $\leftarrow$ Xd[47:0]
5: **end function**

5.4.2 CFI Primitives

We first discuss the CFI primitives based on the ARM instruction set, and then show how they protect different control-flow instructions.

CFI State and Updates

Instead of signing a pointer with `PACIA`, we use it to compute the CFI state. The upper bits of a `PACIA` computation (the size depends on the virtual memory configuration, but we use a 15-bit configuration), the PAC bits, denote our CFI state. To accumulate the CFI state, the `PACIA` instruction is always executed on the same "pointer", in our case, the CFI state is stored in `Xd`. The `PACIA, Xd, Xn` instruction computes a PAC of register `Xd` with `Xn` as a modifier and XORs it to the upper bits of `Xd`. For each basic block, a unique identifier, *i.e.*, the Program Counter (PC), is used as the modifier `Xm` for this instruction. By subsequently XORing the new CFI state to the previous one, we create a dependency link between succeeding basic blocks. We store the global CFI state in the exclusively reserved general-purpose register `x28`, which cannot be used by the rest of the program.

Listing 5.1 shows the CFI state update, placed at the beginning of each basic block. `ADR, x2, #4` first loads a unique constant for the basic block to a temporary register `x2`, in this case, the program counter. We use this constant to compute a new PAC, which gets XORed to the previous CFI state in `x28`.

```
1   adr   x2,  #4
2   pacia x28, x2
```

Listing 5.1: State update with `PACIA`.

Note that FIPAC can also be implemented using the original ARM PA instruction set extension of ARMv8.3-A. However, the ARM PA instructions in this version simply replace the upper bits in the pointer with the PAC, omitting the linking functionality we need for FIPAC. To create a dependency between the previous and the current CFI state, the linking mechanism needs to be implemented manually in software. However, this requires more instructions and thus impacts the code size and runtime performance.

State Patches

As discussed before, it requires justifying signatures for control-flow transfers such as conditional branches or calls. To inject a justifying signature needed for control-flow merges, we use the instruction sequence from Listing 5.2. In Listing 5.2, we show the instructions for such a patch update. We load an immediate constant to a temporary register in `x2`, which gets XORed to the CFI state in `x28`, thus correcting it to a target state. The computation of this immediate constant happens during the post-processing stage, as discussed in Section 5.4.4. Note that an XOR applied to a valid PAC value generates an invalid PAC, which needs to be corrected before verification.

```
1   mov  x2, #patch
2   eor x28, x28, x2
```

Listing 5.2: CFI state patch.

State Checks

A check compares the current CFI state with the expected state at this program location and executes an error handler on a mismatch. Such instruction sequences typically involve conditional branches, which slows down the program execution, as they impact the instruction pipeline. We also exploit the ARM PA instructions for efficiently performing the necessary CFI checks. Similar to generating a PAC, ARM also provides `AUTI*` and `AUTD*` instructions to verify the integrity of PACs. In ARMv8.6-A, these instructions even trap on an invalid PAC verification. Since we use `PACIA` to compute a PAC, it is tempting to use `AUTIZA` directly for verification. However, the CFI state in `x28` is not a valid PAC value in the classical sense. Instead, it is an accumulated XOR-sum of many valid PAC values that combined does not form a valid PAC anymore. Thus, we cannot directly use the `AUTIZA` instruction to verify the CFI state.

```
1   mov    x2, #const
2   eor    x2, x28, x2
3   autiza x2
```

Listing 5.3: CFI check with `AUTIZA`.

At every location in the program, we know the expected CFI state at compile-time. Thus, we can compute a differential constant, which is XORed the CFI state, transforming it to a valid PAC. By applying this constant to the CFI state, we receive a valid PAC that can be verified with `AUTIZA`. This constant is determined in the post-processing tool and explained in Section 5.4.4. In Listing 5.3, we show the corresponding assembly sequence. We first insert an instruction sequence that patches the current CFI state to a valid PAC value using a constant for this program location. Then, we use the `AUTIZA` instruction to verify the integrity of this PAC value. On a control-flow deviation, applying the constant to the incorrect CFI state in `x28` generates an invalid PAC, which the `AUTIZA` instruction detects. If the check fails, `AUTIZA` traps and stops the program.

CFI checks can be placed arbitrarily within the program. FIPAC supports three strategies: one check at the end of a program, a check at the end of every function, or a check at the end of every basic block. The check strategy directly impacts performance and security, which is discussed in Section 5.5.

5.4.3 Protection of Control-Flow Instructions

We now discuss how the CFI primitives are used to protect different control-flow instructions. At the beginning of each basic block, we insert the sequence for an ARM PA-based CFI state update, as shown in Listing 5.1. This instruction sequence uniquely updates the CFI state for the current basic block based on the previous state value. For all CFI primitives, the compiler emits metadata to a custom Executable and Linkable Format (ELF) section that is used by the post-processing step.

Protection of Direct Branches, Jumps, and Conditional Branches

These control-flow instructions create control-flow merges, where state collisions occur. At control-flow merges, our compiler instruments those instructions and inserts the state patches for justifying signatures. Note the actual patch values are left to be zero, and final patch values are determined during the post-processing, as discussed in the next section. To identify the locations of patches, we compute the inverted maximum spanning tree over the edges of the CFG, defining the patch locations.

Direct Calls

Direct calls are instrumented with state patches at the call site, transforming the state to the beginning state of the called function. When returning from a directly called function, the caller's CFI state continues with the callee's end state. Note that functions are instrumented to only have single return nodes.

Indirect Calls and Returns

At the call site, indirect calls are instrumented to stack the current CFI state and patch the state for the intermediate state for this set of indirect calls. When returning, the pre-call state saved on the stack is retrieved and XORed to the CFI state to provide a link over the indirect call.

Indirect calls require more complicated instrumentation besides the call site. As discussed in Section 5.3.1, the function header of an indirectly called function needs to set up the patch value used during the function's return. However, a function generally does not know how it was called and must support being called directly and indirectly. We solve this problem by adding a second function entry point, one for direct calls and the second one for indirect calls.

We add a custom function entry for indirect calls in the compiler, shown in Listing 5.4. This entry patches the intermediate state of the indirect call to the beginning state of the called function (Line 1-3). We then load the CFI update patch (Line 4), used during the function's return, and jump, in Line 5, over the direct call entry point. When the function is called directly, it jumps to the direct call function entry in Line 6, setting up a zero-patch for the return. During the function return, Line 8 uses the previously set up return patch. For direct calls, where the return patch is zero, this statement has no effect, but for indirect calls, it patches the end state to the intermediate return state. The compiler is unaware that the inserted instructions have control-flow and implement a second function entry. Thus, direct calls also use the second entry point, which is exclusively for indirect calls. We correct this during the post-processing, where all direct calls get rewritten to the second entry.

5.4.4 Toolchain

Related work [CFC; Sch+18a; WWM15] either uses the compiler or dedicated binary post-processing tools for the instrumentation. Our prototype toolchain

```
1   mov x1 , #I_PATCH   ; Indirect call entry point
2   eor x28 , x28 , x1  ; Patch to beginning state of function
3   mov x1 , #RET_PATCH ; Load return patch
4   b #8
5   mov x1 , #0         ; Direct Call entry point
6   ...                 ; sets up zero patch
7   eor x28 , x28 , x1  ; Apply return patch
8   ret
```

Listing 5.4: Function entry points for indirect and direct calls.

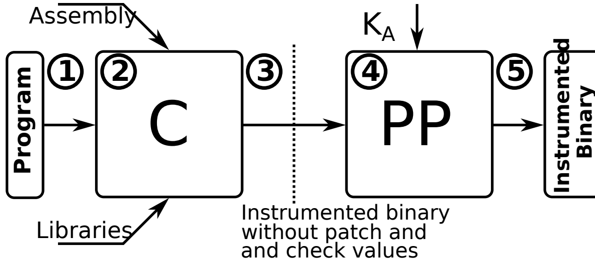


Figure 5.7: Custom toolchain to build protected binaries.

uses a combination of both approaches, shown in Figure 5.7. We use a custom compiler ② based on the LLVM compiler framework [LA04] to insert all necessary state update and patch instructions using two backend passes during the compilation of a program ①. We extend the AArch64 backend and reserve the general-purpose register x28, which is exclusively used to store the CFI state, disable tail calls, and ensure that functions have only a single return point. The compiler emits an instrumented ELF binary ③, but the concrete state patches and check values are set to zero. In the second step, we use a post-processing tool (PP) ④, which has access to the compiled and linked binary to compute all expected states and insert the patch updates.

The toolchain supports instrumented or non-instrumented libraries, but only instrumented libraries have CFI. Instrumented libraries must be linked statically such that the PP tool can replace the patch and check values in the binary. The toolchain also supports inline assembly and external assembler files. However, the programmer's responsibility is to insert the necessary state update and patch sequences into the assembly code. If the assembly code is not instrumented, the code is still fully functional but does not have CFI protection. The toolchain currently supports the instrumentation of programs written in C. However, extending the support to other languages supported by LLVM, e.g., C++ or Rust, only requires more engineering work but no changes to the design of FIPAC.

Post-Processing Tool

The post-processing tool performs the call rewriting, the CFI state computation, the insertion of the patch values, and the computation of the CFI check values. It has access to the ARM PA key and consumes the instrumented binary with zeroed patches and checks. The tool rewrites all direct calls to use the second function entry point (the first one is used for indirect calls). Next, it computes the CFI state for every location in the program. Every function is assigned a random start signature, which is propagated through all PAC-based state updates of the function. At a control-flow merge, the state values of both branches are known such that the tool can compute the justifying signature as the XOR-difference between both states. It finally replaces the patch value `#patch` with the previously computed justifying signature. The post-processing tool knows the CFI state at every location in the program; thus, it can also compute the XOR-differences to form a valid PAC. For `AUTIZA`-based check sequences, it replaces `#const` with the corresponding XOR-difference. Note that the operating system must set up the same ARM PA key before starting the instrumented binary.

5.5 Evaluation

In this section, we discuss the security guarantees of FIPAC and analyze different checking policies. We validate the correctness of our scheme by running the application-grade SPEC 2017 benchmark and Embench on a functional ARMv8.6-A model. Finally, we describe our test setup to measure the runtime overhead for FIPAC and evaluate the overhead of different checking policies.

5.5.1 Security Evaluation

FIPAC considers a software and a fault attacker aiming to hijack control-flow transfers between basic blocks. To protect these control-flow transfers, FIPAC performs a state update of the global CFI state S at the beginning of every basic block allowing FIPAC to detect inter-basic block manipulations triggered by a software- or a fault-attacker. We provide real-world exploits in Section 5.6 and show how FIPAC protects the programs from such attacks.

Protection against Software-Based Control-Flow Attacks

A software-based control-flow attacker is able to hijack the control-flow by modifying indirect calls or returns by exploiting a memory bug. FIPAC mitigates these hijacks, *i.e.*, ROP or JOP, by ensuring that the executed control-flow follows the statically derived CFG. When entering a basic block, FIPAC derives a new state considering the execution history and a unique basic block identifier. On a control-flow hijack, the attacker redirects the control-flow to a basic block that is not in the set of valid targets. Hence, the state update derives a faulty state, which is detectable by the following check. If the attacker omits the update, e.g., by redirecting the control-flow to the middle of the basic block, the check

before the return instruction detects the wrong state, mitigating ROP attacks. Suppose the attacker omits the state update, e.g., by redirecting the control-flow to the middle of the basic block. In that case, the check before the return detects the wrong state, mitigating ROP attacks.

Compared to other CFI protection schemes, which only consider a fault-based control-flow attacker, FIPAC uses a keyed state update to prevent a software-based or combined software- and fault-based control-flow attacker from forging a valid CFI state. Equation (5.1) depicts the state update function ignoring the excluded bit 55 for simplification purposes. This function consists of the secret key K_A, the current state S, and a unique identifier Sig_{BB} for the basic block. The secret key K_A, inaccessible by the adversary, is initialized at boot time and ensures that the attacker cannot forge a specific state.

$$\begin{aligned} S &= \text{Update}(S, Sig_{BB}, K_A) \\ &= S \oplus \text{MAC}_{K_A}\left(Sig_{BB}\right)_{\text{PAC}_{\text{Size}}} \end{aligned} \tag{5.1}$$

Protection against Fault-Based Control-Flow Attacks

While mitigating software-based control-flow attacks only requires protecting a subset of control-flow transfers, *i.e.*, returns and indirect calls, thwarting a fault-based control-flow attack necessitates the protection of all control-flow transfers. Hence, in addition to SCFI schemes, FIPAC also updates the CFI state for direct calls and branches, detecting any faults on code addresses stored in the memory, registers, or during the execution.

Detection of a Control-Flow Violation

FIPAC does not prevent a control-flow hijack; instead, it detects an attack, after the control-flow was violated, at the next check. This is the best that software-based CFI can do, as they cannot verify branches or calls ahead of executing them. If an attacker skips the check at the end of the basic block/function, the hijack is not detected in the first place. However, depending on the checking policy, a new check occurs at the end of the next basic block or function. Since the CFI state is invalid at this point, it requires the attacker to skip all subsequent checks such that the control-flow attack is not detectable. Control-flow attacks, which redirect the execution to the program's end, are not detectable, as there is no check anymore.

CFI State Collision Probability

PAC_{size} is essential for the security of FIPAC. Due to the truncated MAC, state collisions are possible with a probability of $P_{Coll}=\frac{1}{2^{PAC_SIZE}}$, which can lead to a bypass. Figure 5.8 illustrates a control-flow hijack, redirecting the call from `B` to `C` by using a software vulnerability or a fault. When returning to the caller `A`, the state mismatch $S_{C_{exit}} \neq S_{B_{exit}}$ should be detected by the check of FIPAC.

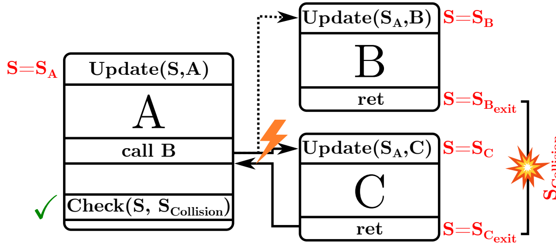


Figure 5.8: Control-flow hijack from `B` to `C`. Due to a state collision, the control-flow hijack is not detected.

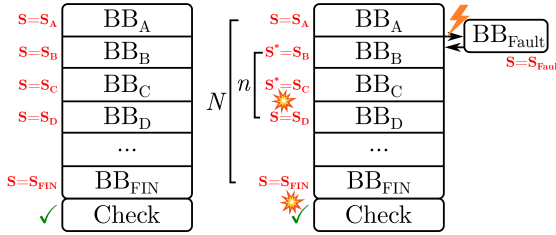


Figure 5.9: A coarse-grained check policy. After n updates, a collision rectifies the faulty state.

However, with probability P_{Coll}, a state collision $s_{C_{exit}} = s_{B_{exit}} = s_{Coll}$ occurs, and the control-flow attack remains undetected.

Checking Policy

To reliably detect state collisions, the sufficient placement of checks, *i.e.*, the checking policy, is crucial for the security of FIPAC. However, properly placing CFI checks is a challenging problem with no general solution. Figure 5.9 shows the problem of a too coarse-grained checking policy. Left, a valid control-flow from basic block BB_A to BB_{FIN} is shown. Right, the attacker manages to redirect the control-flow to BB_{Fault} and therefore alter all subsequent states to S^*.

However, with a probability of P_{Coll}, a state collision occurs after each state update. In this example, after n updates, a collision occurs, and S^* becomes S_D. Thus, the state S is valid again, and the control-flow hijack cannot be detected in further CFI checks. To give a quantitative measure of the security of the check placement, we analyze the probability of undetectable state collisions between subsequent checks. Equation (5.2) denotes the minimum probability that a state collision occurs in one of N state updates. As illustrated in Figure 5.10, after the execution of 50,000 state updates, a state collision probability of 78 % is given. With the execution of 250,000 state updates, and therefore the same number of basic blocks, a collision occurs with almost 100 % for a 15-bit PAC.

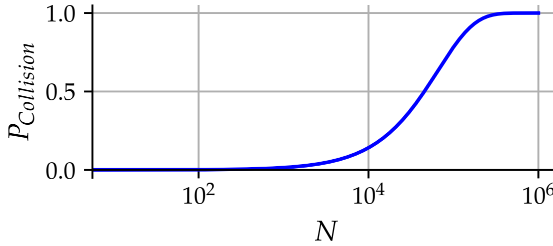


Figure 5.10: Collision probability after N state updates.

$$\mathrm{MP}_{Collision_N} = 1 - \left(1 - \frac{1}{2^{PAC_SIZE}}\right)^N \tag{5.2}$$

Selecting the checking policy is a trade-off between security and performance. Although a precise policy, *i.e.*, a check at each basic block, maximizes the detection probability of a control-flow hijack, the performance overhead also increases. While a loose checking policy, e.g., a check at the program's end, might be sufficient for small programs, programs with a high number of executed basic blocks might be vulnerable. Between these two policies, arbitrary checking strategies can be selected; for example, a check at the end of each function. A more advanced check strategy can incorporate additional information, e.g., runtime profiling. This allows the compiler to better decide where checks are needed to enforce a lower bound of the minimum detection probability of CFI errors.

A check at the end of a function is a good trade-off between runtime overhead and security. For example, SPEC 2017 consists of 28391 functions. 12583 of these functions, or 44 %, contain only a single basic block with a check at the end. Thus, calling such a function is equivalent to performing a CFI state check at the call site. For example, calling this function within a loop containing no explicit checks implicitly performs a state validation at each loop iteration.

We analyzed the number of basic blocks per function for SPEC 2017. Figure 5.11 depicts the occurrence of functions with a certain number of basic blocks. The number of functions with a small number of basic blocks is much larger than functions comprising a large number of basic blocks. Almost 75 % of all functions consist of less than 13 basic blocks, which is in favor of our checking policy since smaller functions perform a CFI check earlier than large ones. Thus, the detection probability of a state mismatch is higher. To summarize, we expect that a CFI check at the end of each function is a good trade-off for a static policy.

5.5.2 Security Comparison

Table 5.1 compares CFI protection schemes addressing software [Aba+05; Eva+15; Lil+19; Lil+21; Mas+15; Tic+14; ZS13] or fault [HLB19; LHB14; OSM02; Rei+05; VHM03] adversaries with FIPAC. Software CFI protection schemes, like

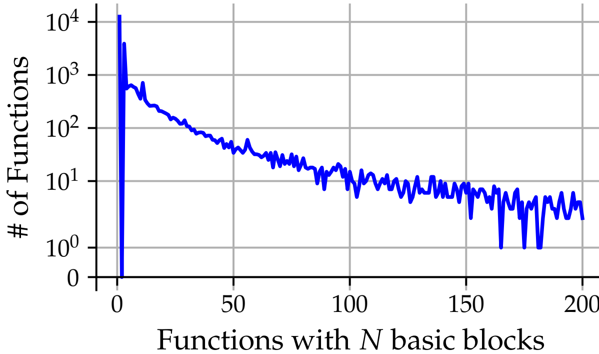


Figure 5.11: Number of functions with N basic blocks.

PARTS [Lil+19] or CPI [Eva+15], enforce CFI at a coarse granularity by protecting a wide range of forward- and backward edges on the function level. Although these approaches mitigate software-based control-flow attacks (🐞) exploiting a memory vulnerability, they fail to protect against a fault-based control-flow attacks (ϟ). FCFI protection schemes enforce CFI at a finer granularity to protect the control-flow from fault attacks, *i.e.*, on basic block or instruction level. In contrast to a software attacker exploiting memory bugs, a precise fault can tamper with direct and indirect control-flow transfers. While software-based FCFI protection schemes protect all control-flow transfers from faults (ϟ), they fail to protect against software-based control-flow attacks (🐞). As the state update of these schemes is based on counters or predictable IDs, an adversary can use a memory bug to modify the state and prevent the detection of a control-flow hijack.

To protect against control-flow attacks from a software- and fault-based attacks, it is tempting to naïvely combine existing schemes such as PARTS with FCFI, e.g., CFCSS. While these schemes are secure in their own threat model, combined software- and fault-based attacks (💣) can bypass them. First, the

Table 5.1: Protection guarantees and vulnerabilities for SCFI and FCFI protection schemes compared to FIPAC.

	SCFI		**FCFI**		**FIPAC**
	Prot.	Vuln.	Prot.	Vuln.	
Return Addresses	🐞	💣	ϟ	🐞	✔
Indirect Calls	🐞	💣	ϟ	🐞	✔
Indirect Branches	🐞	💣	ϟ	🐞	✔
Direct Calls		ϟ	ϟ	💣	✔
Direct Branches		ϟ	ϟ	💣	✔

✔ Full 🐞 Software ϟ Fault 💣 Combined

```
1 eor x28, x28, #2   ; Instruction sequence
2 eor x28, x28, #3   ; spends CPU cycles to
3 eor x28, x28, #5   ; emulate PA overhead
4 eor x28, x28, Xmod
```

Listing 5.5: `PACIA` emulation used for the performance evaluation.

adversary gains control over a register used for the FCFI state update. Then, it redirects the control-flow to a wrong function, e.g., with a fault. Finally, the tampered register is used for the state update, thus, can forge a valid CFI state.

To protect against software- and fault-based control-flow attacks and to support a large-scale deployment, FIPAC fulfills the key requirements stated in Section 5.2.3. First, FIPAC comprehensively enforces CFI for transfers between basic blocks. Hence, our scheme operates on a much finer granularity than typical software CFI protection schemes. Second, FIPAC uses, in comparison to fault CFI protection schemes, a keyed state update function to mitigate attacks targeting to manipulate the global CFI state. FIPAC is implemented in software and is applied automatically during compilation.

5.5.3 Functional Evaluation

To evaluate the functional correctness of FIPAC, we compiled SPEC 2017 [Sta19] and Embench [Pat+] with our LLVM-based toolchain. We executed these instrumented binaries on the QEMU 6.0 [QEM20], which we modified to support ARM PA of ARMv8.6-A. More concretely, to emulate ARMv8.6-A, we modified the `pauth_addpac` function with the *EnhancedPAC2* feature and return the XOR-sum of the upper pointer bits with the computed PAC. To emulate FPAC, we extended the `pauth_auth` function to terminate on a PAC verification failure. In QEMU, we started the 5.4.58 Linux kernel and initialized the ARM PA keys during the boot procedure before starting the benchmarks.

5.5.4 Performance Evaluation

FIPAC exploits Pointer Authentication of ARMv8.6-A. To the best of our knowledge, there is currently no publicly available device supporting ARMv8.6-A with extended ARM PA. To conduct our performance evaluation on hardware, we use the Raspberry Pi 4 Model B [Ras20], and run our experiments on the 64-bit Raspberry Pi OS. Since the ARM Cortex-A72 Central Processing Unit (CPU) is based on ARMv8-A without ARM PA, we emulate the runtime overhead of the ARM PA instructions in software by replacing them with their PA-analogue, *i.e.*, four consecutive XORs. PARTS [Lil+19] evaluated this sequence to model the timing of native ARM PA instructions, which is also used in related work [Lil+21].

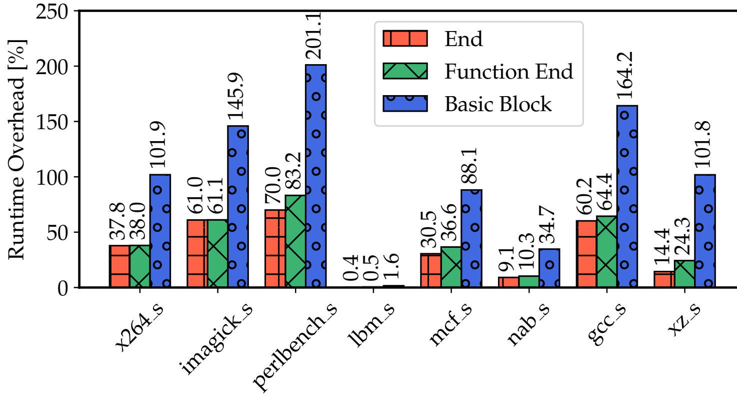


Figure 5.12: Runtime overhead for SPECspeed 2017.

SPEC 2017

To measure the performance overhead of FIPAC, we compiled all C-based benchmarks with OpenMP support disabled of SPECspeed 2017 Integer. We enabled three different checking policies, from coarse-grained to fine-grained checks, to compare the performance penalty introduced by them. More concretely, we configured FIPAC to insert a CFI check at the end of the program, at the end of every function, or at each basic block. Table 5.2 summarizes the code overhead with different checking policies compared to the baseline with no instrumentation. As expected, the checking policy with a single check at the end of the program has the lowest overhead. Verifying the CFI state at the end of every basic block has the largest geometric mean penalty in code size of 90.6 %, as it requires three additional instructions per basic block. Interestingly, placing a CFI check at the end of every function only has a geometric mean overhead of 52.5 %, slightly higher than a single check at the program end with a geometric mean penalty of 50.6 %. Due to this small increase in code size but its stronger security guarantees, this policy is a good trade-off.

Figure 5.12 shows the runtime overhead of FIPAC compared to the baseline without protection. The coarse-grained checking policy with a single check at the program end introduces the smallest geometric mean runtime overhead of 18.8 %. The fine-grained checking policy with CFI checks at the end of every basic block has the largest geometric mean runtime penalty of 62.9 %. Interestingly, the intermediate policy with a check at the end of each function introduces a geometric mean runtime overhead of 22.1 %. This is only a small increase compared to a single check at the end, but it provides much better security. Note that since the control-flow of the `lbm_s` benchmark is mostly linear and this test performs a large number of expensive floating point operations, the impact of FIPAC is rather small. These runtime overheads are outperforming related work with overheads between 107–426 % [Gol+03].

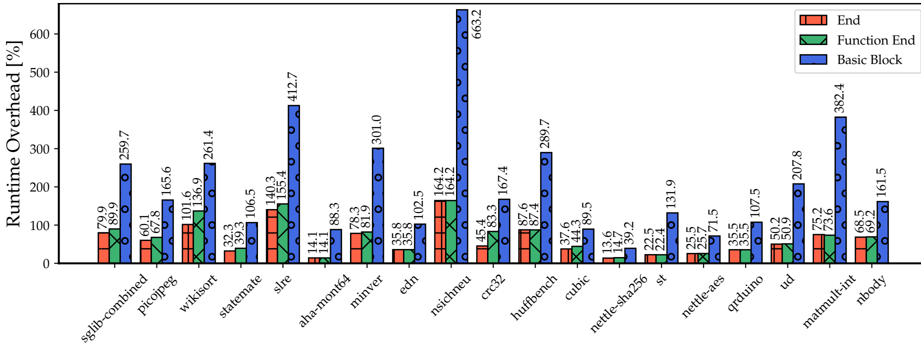


Figure 5.13: Runtime overhead for Embench.

Embench

To evaluate FIPAC on embedded workloads, we use Embench. The geometric mean code overheads are between 55–95 %, and the runtime overheads are between 49–168 % (Figure 5.13), depending on the checking policy. This increased overhead is due to Embench's small codebase with a larger number of control-flow transfers compared to application-grade benchmarks like SPEC. In Table 5.3, we show the detailed code overhead for all benchmarks of Embench for the three checking policies of FIPAC.

CoreMark

CoreMark [EEM] also represents a widely used embedded benchmark. Instrumented with FIPAC, we observed a code overhead of 69.8 % for checking at

Table 5.2: Code size overhead for SPECspeed 2017.

Testcase	**End** [%]	**Function End** [%]	**Basic Block** [%]
lbm_s	19.19	20.85	33.30
sgcc	74.31	77.62	130.71
xz_s	48.08	50.65	85.78
perlbench_s	66.00	67.22	120.18
nab_s	63.96	65.38	117.15
x264_s	41.56	42.60	73.40
imagick_s	58.44	59.99	101.91
mcf_s	61.29	62.78	115.92
Geomean	**50.62**	**52.51**	**90.65**

program end, 72.7 % for checking at every function end, and 127.1 % for checking at the end of every basic block. Corresponding to that, we measured runtime overheads of 76.7 %, 79.6 %, and 274.0 % for the different checking policies. Similar to Embench, also CoreMark is dominated by control-flow, and, thus, has increased runtime overhead compared to SPEC 2017.

Table 5.3: Code size overhead for Embench.

Testcase	**End** [%]	**Function End** [%]	**Basic Block** [%]
huffbench	70.04	76.03	119.19
slre	81.72	86.37	146.78
cubic	31.43	34.04	56.48
nbody	70.19	77.39	116.09
minver	74.24	80.79	126.64
sglib-combined	81.24	86.47	147.62
st	69.74	76.49	114.74
matmult-int	39.50	43.49	64.18
statemate	61.68	65.08	112.24
crc32	48.16	53.97	76.17
aha-mont64	70.82	77.66	117.22
qrduino	53.38	56.27	99.46
ud	71.66	78.07	119.44
edn	43.78	47.93	71.36
nettle-sha256	46.07	50.65	81.62
wikisort	60.25	64.86	105.86
nettle-aes	19.59	21.46	32.95
picojpeg	50.73	53.49	88.24
nsichneu	54.07	55.27	113.76
Geomean	**54.84**	**59.26**	**95.08**

5.6 Example Exploits

This section illustrates example control-flow exploits and shows how FIPAC detects them.

NaCl Sandbox Escape via Rowhammer

In [Goo15], Rowhammer is used to perform a sandbox escape exploit out from NaCl. The Rowhammer effect is used to manipulate a jump instruction in the memory to an attacker-controlled destination. In Listing 5.6, the final jump instruction is faulted for modifying the operands of the instruction. For example, the jump instruction is modified to use the attacker-controlled register `ecx` as the jump target instead of `rax`.

```
1   andl $~31, %eax     ; Truncate and align to 32 bits.
2   addq %r15, %rax     ; Add %r15, the sandbox base address.
3   jmp *%rax           ; Indirect jump.
```

Listing 5.6: NaCl attack gadget.

After jumping to the attacker-defined code position, FIPAC detects the control-flow manipulation at the subsequent check instruction. As discussed in the design of FIPAC, fine-grained checking policies, such as the end of every basic block or at the end of every function, are supported. When executing such a check instruction, FIPAC detects the control-flow hijack since the CFI state is invalid.

RCE on an Electronic Control Unit

In [NT], a combined software- and fault-based control-flow attack was used to gain Remote Code Execution (RCE) on an automotive Electronic Control Unit (ECU).

```
1   ldr r1, [r2, #4]
2   ldr pc, [r2, #4]
```

Listing 5.7: Instruction corruption to load `r2` into `pc`.

Here, as depicted in Listing 5.7, a targeted fault was used to corrupt a load instruction to modify the program counter. As register `r2` either is directly attacker controllable or can be modified using a memory vulnerability, the attacker can redirect the control-flow to an arbitrary code position. While the concrete attack is not directly applicable to AArch64, as the program counter cannot be manipulated via a load instruction, [Tim21] highlights alternative methods for corrupting instructions to redirect the control-flow. Independently of the actual targeted instruction to corrupt, a fault is used to modify the program counter in an attacker-controlled way, yielding an arbitrary jump primitive. FIPAC is able to protect the program from such control-flow hijacks, as the global CFI state does not match the expected CFI state determined at compile-time for this

position in the code. Hence, at the next FIPAC check instruction, e.g., at the end of the function, the attacker redirected the control-flow to, AUTIZA traps, and FIPAC detects the attack.

Defeating ROP with FIPAC

The idea of ROP [Sha07] is to redirect the control-flow to small code-snippets, so-called gadgets, with a return at the end. By chaining these gadgets together, Turing-complete code execution is possible.

```
1  <libc>:
2    state_patch                        ; S = StateL
3    ...
4    ...                                ; gadget_start
5    ...
6    state_patch                        ; S = StateF XOR PatchL
7    state_check                        ;  S!= StateL
8    ret                                ; gadget_stop
9  <function>:
10   ...
11 <main>
12   state_update                       ; S=StateM
13   ...
14   ldr x8, function                   ; gadget_start
15   state_patch                        ; S = StateF
16   blr x8                             ; indirect call to <function>
```

Listing 5.8: Detection of ROP attack.

Listing 5.8 depicts a classical ROP exploit and highlights how FIPAC detects the attack. In this example, the adversary exploits a memory vulnerability in Line 13 to overwrite an address stored in memory. This address, which is loaded from memory to the register x8, now points to the start of the ROP gadget in Line 4 in libc, to which the control-flow is redirected. Depending on the checking policy, FIPAC automatically inserts a state check instruction before each return. As the expected state S does not match the actual state, the control-flow hijack is detected at Line 8.

Attack on CT-RSA of Mbed TLS

In [Car+19], they analyze the fault exploitation against the CT-RSA algorithm of Mbed TLS [Lin23] cryptographic library. They used a targeted fault-based control-flow attack on Mbed TLS, with the goal of breaking the RSA signature algorithm. This attack highlights that attacks on the control-flow can also be used to attack cryptographic algorithms.

In one of their experiments, they used a fault during a direct call to manipulate the control-flow to a different function. At the manipulated call target, they modify the stack, similar to a ROP attack, for their further exploitation. FIPAC

does not prevent the first control-flow manipulation in the first place. Instead, since the control-flow was manipulated to a different call location, the CFI state does not match anymore, which is detected by the following state check.

```
1  <rsa_ossl_mod_exp>:
2    state_patch                        ; S = StateL
3    ...
4    state_patch
5    bl BN_free
6    ...
7    state_patch
8    state_check
9    ret
10
11 <BN_free>
12   state_update                       ; S = StateM
13   ...
14   ...
15
16 <BN_clear_free>
17   state_update                       ; S = StateM
18   ...
19   ...
20   sub sp, r11, #4
21   pop r11, pc
22   ...
23   state_check
24   ...
25   ret
```

Listing 5.9: Detection of control-flow attack on Mbed TLS.

In Listing 5.9, the function `rsa_ossl_mod_exp` originally calls `BN_free` via a direct call. In Line 5, this call to `BN_free` is faulted for redirecting the control-flow to Line 20. Although this redirect is not prevented, the subsequent CFI check in Line 23 in `BN_clear_free` detects the control-flow deviation. Thus, FIPAC prevents any further exploitation of the control-flow attack and aborts the program.

5.7 Data Protection with FIPAC

Although FIPAC is a CFI protection scheme, it can also be used to protect certain data of software. In particular, the state-based CFI design allows the software to inject arbitrary data into it. If the data is known at compile-time, the post-processing tool can pick that up during the state computation. The state computation then uses that compile-time known data and performs an update with that, similar to an ordinary CFI patch operation.

```c
int cnt;
for(cnt = 0; cnt < 10; ++cnt)
{
  // do something
}
asm volatile("injv %0, #10", : : "r"(cnt));
```

Listing 5.10: Data injection using `injv` instruction.

During runtime, the software uses an `eor` operation to inject or XOR the register that contains the data to be protected to the CFI state. Only if that register contains the correct value at that time and program location the CFI state is updated accordingly and has the expected value. If the data is manipulated, *i.e.*, due to a fault attack, the state gets updated with the wrong value and manifests there. Subsequently, the next CFI check detects the state mismatch as a CFI error and aborts the program. Using this approach, we translate data errors to CFI errors that are detectable via FIPAC.

To comfortably use this protection mechanism, we extend our compiler with a custom `injv` instruction. This new instruction takes the register to inject and the expected value for that register as two operands. Internally to the compiler, this instruction is just a pseudo instruction that expands to an `eor` instruction, XORing the register from the operand to the global CFI state in register `x28`. The expected value is emitted to a metadata section of the ELF file to be used by the post-processing tool.

In Listing 5.10, we show the usage of this protection mechanism for a C-based `for` loop. After the constant amount of ten iterations, the loop counter is injected into the CFI state. At this program location, the variable has the expected value of `10`, since there are no early exits from the loop. Only if all loop iterations are completed, and the loop counter `cnt` has the value `10`, the subsequent CFI check succeeds, e.g., at the end of the function.

While currently, this approach requires a manual adaption of the source code, this can even be integrated into the compiler in a future work. A compiler pass can insert the necessary injection operations for all known data or at a user-defined policy.

5.8 Discussion

This section discusses the hardware requirements of FIPAC, how it can be implemented on other architectures, and future improvements.

5.8.1 FIPAC Hardware Requirements

FIPAC requires Pointer Authentication from ARMv8.6-A with EnhancedPAC2 and FPAC. At the time of writing, there is no open hardware available implementing ARMv8.6-A yet. Although it is not yet widely available in existing

processors, ARM has already announced the successor ARMv9-A [ARMc]. Hence, we expect new designs, e.g., Apple's new processors, to feature ARMv8.6-A or even ARMv9-A.

5.8.2 FIPAC on ARMv8.3-A

FIPAC can also be implemented on ARMv8.3-A with the following adaptions. ARMv8.3-A ARM PA instructions only compute a new PAC without accumulation, which must be done manually using an additional `eor` instruction per state update. This increases the overhead of an update to 3 instructions and requires one more register. `autiza` in ARMv8.3-A cannot be used as a check as it does not trap. However, ARMv8.3-A features `blraa`, a branch with link operation with Pointer Authentication, which traps if the jump-target contains an invalid PAC. This instruction can be misused to perform a CFI check. First, we transform the known CFI state to a valid PAC with the address of the next instruction. When executing this branch, it first verifies the target address and, if valid, jumps to the next instruction. If the PAC, and therefore also the CFI state, is invalid, the verification traps. Both solutions increase code size and runtime overheads compared to the prototype of FIPAC based on ARMv8.6-A.

Similar to ARMv8.6-A, there is currently no open hardware available for ARMv8.3-A yet. Although Apple offers cores, such as the M1 and A14 [App21], they restrict the usage of this feature. iOS applications are not allowed to load kernel modules; thus, FIPAC cannot configure the ARM PA keys. FIPAC may run on the Apple M1 core with ARM PA of ARMv8.3-A. However, we currently do not have access to such a device, and it requires future research to clarify if ARM PA key access is possible in the EL1 kernel mode or if Apple restricts it.

5.8.3 FIPAC on Other Architectures

The design of FIPAC is generic and could also be implemented on other architectures. It is tempting to implement FIPAC on x86 with the AES-NI [Rot12], supporting partial encryption with one instruction. However, we see limitations with this approach. First, AES-NI operates on a 128-bit state, also requiring to embed 128-bit patches. Second, one AES-NI operation only computes one round, just providing scrambling and no cryptographic strength. Third, it requires the encryption keys to be held in general-purpose registers. Thus, there is no key isolation between the user and the kernel. Hence, we do not envision FIPAC to be implemented with AES-NI.

Recently, the reverse engineering and patching of the microcode of a certain Intel microarchitecture introduced instruction for Pointer Authentication [Bor+; Bor23]. While this emulated approach only provides instructions to sign and verify a pointer, which is similar ARM PA of ARMv8.3, the emulation can be extended for the updated ARM PA instructions. However, it still needs to be determined how the signing keys can be isolated by the operating system required for the security of FIPAC. If this challenge can be solved, FIPAC can be fully implemented on existing x86 CPUs.

5.8.4 Dynamic Key Handling

FIPAC uses a static ARM PA key configured by the OS. However, ARM Pointer Authentication supports up to five keys for different domains. By using different keys, FIPAC could isolate the control-flow of the kernel and user programs. Future work can extend the instrumentation to support all available encryption keys to also provide CFI isolation. For better isolation between applications, FIPAC could embed the ARM PA key in the binary, allowing applications to use different keys. Existing key exchange algorithms are then used to protect the embedded ARM PA key. The OS has access to a private key for the key exchange, can read the ARM PA key, and configure the system before starting the binary. The previous approaches use a static key per binary, such that all executions use the same ARM PA key. To dynamically change the ARM PA key, the post-processing can be integrated into the OS. Before starting the application, the OS chooses a random key and performs the post-processing step, *i.e.*, the computation of the CFI states, patches, and check values. Thus, every invocation of the application is different in terms of FIPAC-related patch and check values, which also hardens the attack surface.

We tackle this challenge and enhance FIPAC with this approach with SFP, which is discussed in Chapter 6. Furthermore, we use FIPAC to extend the scope of protection and secure the system call flow.

5.8.5 Instruction Granular Protection

FIPAC does not protect the linear instruction sequence within a basic block, as it only performs state updates at the beginning of a basic block. If more fine granular protection is required, *i.e.*, intra-basic block security, FIPAC supports the placement of state updates within security-critical basic blocks. For such pieces of code, the state update from Listing 5.1 is placed after every instruction to emulate instruction-granular CFI. Instruction granular CFI increases the overhead and adds two additional instructions per instruction to protect. Automatically identifying such critical pieces of code is a challenging task and not in the scope of this work. Instead, it requires the developer to manually place a check, e.g., via inline assembly. In the future, a compiler could insert the necessary state updates automatically for annotated scopes that require further protection.

5.8.6 Compatibility

FIPAC uses the instruction address to compute a unique CFI state update for every basic block. To compute the justifying signatures and the values needed for the CFI checks, these addresses need to be known at compile-time. However, when Address Space Layout Randomization (ASLR) is enabled, the code address space is randomized, which leads to randomized signatures not being compatible with the static computation. This problem can be solved by using static numbers to compute the signatures or by integrating dynamic key handling in the OS, which we integrate in Chapter 6.

FIPAC is a software-based CFI protection scheme that comes with certain degrees of flexibility compared to hardware-centric approaches. As FIPAC supports arbitrary checking policies on the same system, critical applications, e.g., running within a Trusted Execution Environment (TEE) or an enclave, can have a stronger checking policy than a non-critical application. FIPAC can also be deployed partly within an application, e.g., the protection schemes is only deployed on security-critical functions or code. Furthermore, FIPAC is backward compatible and fully supports non-instrumented applications that execute alongside the ones with protection.

5.9 Conclusion

With the rise of new attack methodologies, fault and software attacks are omnipresent on all commodity devices and require protection. While there exist different defenses, they are not sufficient when considering both fault and software attacks

We presented FIPAC, a fine-granular software-based CFI protection scheme for upcoming ARM-based hardware. FIPAC offers fine granular control-flow protection on basic block level for both software- and fault-based control-flow attacks. The design exploits a cryptographically secure state update function, which cannot be recomputed without knowing a secret key. FIPAC utilizes ARM Pointer Authentication of ARMv8.6-A to efficiently implement the keyed CFI state update and checking mechanism. We provide a toolchain to automatically instrument and protect applications. The evaluation of FIPAC with the SPEC 2017 benchmark with different security policies shows a geometric mean runtime overhead between 19–63 % and is slightly larger for small embedded benchmarks. FIPAC is a software-based CFI protection scheme that requires no hardware changes and outperforms related work.

Having a cryptographic state available within the software turns out to be versatile protection primitive. While FIPAC uses this approach to primarily implement CFI, this mechanism can also be used to protect other assets. As discussed in Section 5.7, the cryptographic state can be used to protect arbitrary compile-time known data. However, this approach can also be used to provide fine-granular software licensing. This would allow a system to only call special library functions if a *paid* key is loaded at the right time. If there is no software license available, calling a paid library function will yield a control-flow error and stops the program. Such a mechanism could be used for virtual Programmable Logic Controllers (PLCs) in the cloud [Sof23] that are used for industrial environments.

Nevertheless, FIPAC still has limitations, *i.e.*, missing a dynamic key handling. In the next chapter, we extend FIPAC, solve the aforementioned limitation, and expand the scope of protection for system calls.

6 System Call Flow Protection and Dynamic CFI

In the previous chapter, we introduced FIPAC, a new countermeasure that protects the control-flow of user-space applications against software- and fault-based control-flow attacks. While FIPAC increases the security against control-flow fault attacks in the user-space, it misses one critical part. As FIPAC is designed to operate on application-class processors, the system usually hosts a feature-rich operating system that manages the resources. While the control-flow of Control-Flow Integrity (CFI) instrumented programs is protected, the interface to the kernel, *i.e.*, the system call or syscall interface, remains unprotected and vulnerable against faults. A control-flow hijack, independent of how it is performed (cf. Section 3.1), can skip a syscall call or can even change which system call gets executed. Both cases may have a critical security impact on the system. Furthermore, precise faults can directly manipulate which system call gets executed by manipulating the system call register containing the system call number.

While traditionally, CFI was only used to protect user-space applications, different CFI protection schemes can also protect the kernel [CDA14; Ge+16]. However, currently, there are no CFI protection schemes available providing protection between different security domains, *i.e.*, the transitions between the user-space program and the kernel. Thus, the large attack surface, the transitions between user programs to the kernel still remain unprotected against fault attacks, even if CFI is deployed in the user-space and/or in the kernel. Hence, there is a need for new countermeasures that protect the software interface to the kernel and provides system call flow integrity for commodity devices.

Contribution

In this chapter, we solve the problem of the unprotected system call interface and provide system call flow protection on top of CFI, protecting the interface to the kernel against both software and fault attacks. SFP cryptographically links the system call itself and its origin to a global CFI state that is verified at runtime in the operating system. We built SFP on top of FIPAC, but in general, this protection mechanism is designed to be generic and compatible with other CFI protection schemes. A second-stage linking mechanism within the kernel dynamically applies a second link to ensure that the correct system call was selected and executed.

To automatically protect arbitrary programs, we develop an LLVM-based toolchain to provide CFI and instrument all system calls. We provide an instrumented standard library, where all system calls are instrumented with our system call protection. Furthermore, we modify the Linux kernel to dynamically verify at runtime that the correct system call was executed.

We implement SFP on top of FIPAC, the software-based CFI protection scheme from Chapter 5. To increase CFI protection of subsequent runs of a program, we extend FIPAC to randomize the CFI protection at every start of the program. We evaluate the performance of SFP based on a microbenchmark to measure the impact of SFP on the system call latency, leading to an overhead of 1.9 %. To show the applicability to real-world programs, we perform macrobenchmarks using the SPEC 2017 application benchmark. On average, we measure a runtime overhead of 20.6 % for protected applications. Summarized, we make the following contributions:

- We provide system call flow protection by linking the syscall and its origin to a global CFI state and verifying it at runtime.
- We perform dynamic instrumentation of user programs to yield different CFI signatures for subsequent program calls and provide a new CFI checking policy.
- We provide a prototype implementation comprising an LLVM-based toolchain, an instrumented C-standard library, and a modified Linux kernel.
- We evaluate the performance based on a microbenchmark and on the application-grade SPEC 2017 benchmark.

Scientific Contribution

Chapter 6 is primarily based on the following publication that was presented at HASP 2022 in Chicago (Illinois, USA).

Robert Schilling, Pascal Nasahl, Martin Unterguggenberger, and Stefan Mangard. "SFP: Providing System Call Flow Protection against Software and Fault Attacks." In: *CoRR* abs/2301.02915 (2023). DOI: 10.48550/arXiv.2301.02915

I am the main author of this paper and developed the technical idea. I wrote the majority of the text, developed the prototype, and performed all the experiments. Pascal Nasahl contributed to the background section of the text. Martin Unterguggenberger helped in editing the Tikz drawings. Stefan Mangard supported the project in many discussions.

Outline

The remainder of this chapter is structured as follows. Section 6.1 introduces the Linux system call interface and existing countermeasures to protect it. Section 6.2 specifies the threat model for this work and shows how existing attacks can bypass it. Section 6.3 presents the design SFP, and Section 6.4 details the prototype implementation. Section 6.5 provides the security and performance evaluation. Section 6.6 discusses possible prototype improvements, Section 6.7 presents related work, and finally, Section 6.8 concludes this chapter.

6.1 Background

This section provides background to the Linux system call interface and discusses existing countermeasures to provide system call flow protection. Existing protection schemes in the context of CFI are already discussed in Section 3.2 or in Chapter 5.

6.1.1 Linux and the System Call Interface

Linux [Tor22] is a monolithic kernel used in billions of devices [Var22] and embedded systems. To retrieve a particular service or get a specific resource, e.g., reading and writing a file, or to get dynamic memory, the user program needs to request this from the kernel, *i.e.*, via a system call. A system call changes the privilege and transfers the execution from the user-space program to the kernel of the operating system, which then grants or denies the requested service. A user-space program aiming to execute a certain system call invokes the corresponding system call wrapper routine provided by a library. This wrapper then initiates a control-flow and privilege transfer into the kernel-space by using a dedicated instruction, *i.e.*, the `svc` instruction for AArch64. The system call instruction

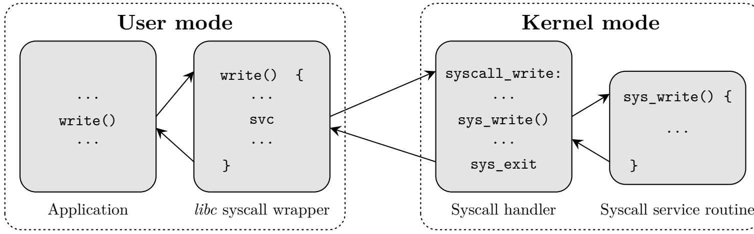


Figure 6.1: Linux system call invocation.

requires the system call number of the requested service and additional optional parameters as arguments.

Figure 6.1 illustrates the system call interface [Ker22b] of the Linux kernel. A user-space program aiming to execute a certain system call invokes the corresponding system call wrapper routine provided by a library. This wrapper then initiates a control-flow and privilege transfer into the kernel-space by using a dedicated instruction, *i.e.*, the `svc` instruction for AArch64. The system call instruction requires the system call number of the requested service and additional optional parameters as arguments.

6.2 Threat Model and Attack Scenario

Our threat model considers a powerful adversary capable of performing software attacks, fault-based attacks, or combined software- and fault-based attacks. This attacker can exploit memory vulnerabilities to arbitrarily read or modify data in memory. However, we assume that a software adversary cannot modify the code segment of the program via a memory vulnerability, as the operating system maps that segment as read-only, and we do not consider Just-in-Time (JIT) compiler environments. In addition, by inducing faults, the attacker can flip bits in memory, the registers, the code segment, or the instruction pipeline of the processor. We assume that the control-flow of executed programs *and* the kernel is protected using a Software-Fault CFI (SFCFI) protection scheme, such as FIPAC.

Note that faults in the data, except the system call register, are out of the scope of this work. It requires orthogonal schemes, e.g., redundancy encoding schemes for data [Bro60], for their protection. We assume the `PACIA` instruction of ARM Pointer Authentication (ARM PA) to be cryptographically secure, and the attacker does not have access to the encryption keys. Furthermore, the operating system is assumed to be secure, providing isolation of the kernel task structure to the user program.

6.2.1 Attack Scenario

Within this threat model, the adversary aims to hijack the program's interface to the Linux kernel. In the example shown in Figure 6.2, the user program invokes

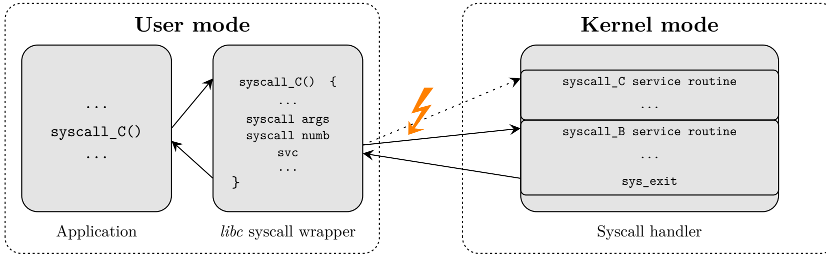


Figure 6.2: Redirecting a system call using fault attacks.

```
1 basic_block:
2   ...
3   ldr w8, memAddress   ; load data from memAddress to w8
4   ...
5   mov x0, #...         ; arguments for C system call
6   mov w8, #syscall_C   ; system call number for C
7   svc #0
```

Listing 6.1: Invoking system call **C** on AArch64.

the system call **C** using the Linux system call interface. However, by using a fault-based attack or a combined software- and fault-based attack, the adversary can either *(i)* redirect the system call to **B** or *(ii)* entirely skip the system call.

Listing 6.1 shows the instruction sequence to invoke the system call **C** on AArch64. The system call number is stored in register `w8`, and the system call arguments are stored in the remaining registers. By flipping bits in register `w8` using faults, the adversary can redirect *(i)* the execution to a different system call.

Moreover, the system call gadget in Listing 6.1 is susceptible to combined software- and fault-based attacks. A combined software- and fault-based attacker utilizes a memory vulnerability to overwrite data in memory at address `memAddress`. Afterward, in Line 4, the adversary hijacks the execution of the program by flipping bits in the program counter to redirect the control-flow to the `svc` instruction in Line 7, responsible for switching to the kernel. This attack enables the adversary to invoke arbitrary system calls. In addition to these attacks, a fault attacker can also corrupt the `svc` instruction to skip *(ii)* the execution of the entire system call.

SFCFI protection schemes, such as FIPAC from Chapter 8, currently *cannot* mitigate these attacks as these countermeasures do not consider transitions between user-space and kernel-space in their threat model. Since our protection scheme only instruments the user-space application, it does not protect the interface to the kernel, which poses a large threat surface for critical vulnerabilities. Even if CFI is deployed independently in the user- and kernel-space, the transitions between those domains still remain vulnerable to fault attacks.

Furthermore, current SFCFI protection schemes use static control-flow instrumentation, which is the same for subsequent calls to the program. As a result, an attacker with access to the code segment or general-purpose registers can learn from previous program executions. Thus, it would be possible for an attacker to attempt multiple control-flow attacks until the hijack succeeds.

6.2.2 FIPAC Intra-Basic Block Protection

In Section 5.8.5, we describe a mechanism to extend the protection guarantees of FIPAC from inter to intra-basic block security. By applying a state update after every instruction within a basic block, we subsequently also update the CFI state continuously. Although this mechanism can be applied around syscalls, it does not add any protection for them. With a state update before and after the system call, an attacker can still fault the syscall number or manipulate the `svc` instruction to perform a *nop* instruction. Although this attack manipulates the execution of the system call, FIPAC's extended intra-basic protection does *not* detect these attacks. Consequently, it requires a different protection scheme to provide call flow protection for system calls.

6.3 Design of SFP

In this section, we present SFP, a mechanism that provides system call flow protection by exploiting a stateful CFI protection scheme. While SFP is generic and compatible with different CFI protection schemes, our design builds on FIPAC as the underlying CFI protection scheme. FIPAC is presented in Chapter 5, and the details of ARM Pointer Authentication (ARM PA) are discussed in Section 5.1. Section 6.6.3 further discusses the compatibility aspects and how SFP can be applied to different CFI protection schemes.

6.3.1 Requirements for System Call Protection

The goal of SFP is to protect the system call interface to the kernel against software-based, fault-based, and combined software- and fault-based attacks. Based on the attack scenario from Section 6.2, the protection of SFP must fulfill the following requirements.

R1 *System Call Number*. Prevent an attacker from manipulating the system call number to a different system call.

R2 *System Call Execution*. Ensure that a syscall cannot be skipped.

R3 *System Call Protection*. Ensure the system call dispatcher in the kernel executes the correct system call function.

R4 *Dynamic CFI Instrumentation*. Provide a dynamic CFI instrumentation to ensure protection between consecutive program executions.

6.3.2 System Call Protection

To fulfill requirements **R1** to **R3**, SFP introduces a two-step approach to cryptographically linking the syscall to the state of the deployed SFCFI scheme. First, at the system call caller site, we cryptographically link the system call origin and which system call we want to execute to the cryptographic CFI state. Second, at runtime, we perform a second-stage linking operation during the system call operation, confirming that the correct syscall gets executed. Furthermore, we break up the toolchain of existing CFI toolchains and shift the CFI instrumentation into the kernel to allow a dynamic computation per program call.

First-Stage System Call Linking

We statically identify at compile-time which system call is getting executed for all locations in the program. To protect the system call, SFP binds the syscall to the CFI state, e.g., by performing a CFI state update with the system call number. The system call number is a monotonically increasing number, thus not providing a significant Hamming distance between different system calls. A single bit-flip on the system call number changes the system call to a different one. For example, the write and read system call on AArch64 only differ in two bits. As a result, the system call number cannot safely be used to bind it to the CFI state since faults can easily manipulate the system call to a different one.

To overcome this limitation and perform a safe and secure state update, we need to compute a syscall-dependent update value with a sufficiently large Hamming distance. In SFP, we exploit the cryptographic properties of the `PACIA` instruction of ARM PA fur this purpose. We use the computation of a `PACIA` operation, with the system call and a randomly selected modifier at instrumentation-time as input, and compute a cryptographic 15-bit patch value for the particular system call, which is stored in the binary. Due to the cryptographic Message Authentication Code (MAC) operations of the ARM PA, the patch values of different syscalls have a large Hamming distance and cannot be computed without having access to the secret ARM PA key. The computation of those patch values occurs at instrumentation-time, which can happen during the compilation or loading of the program through the Operating System (OS), and replaces the empty patch values in the binary.

Before executing a system call and jumping to the kernel, we patch the CFI state with the aforementioned statically computed system call patch, thus performing the *first-stage* linking. At this point in time, we bind the future execution of the particular system call to the CFI state ahead of executing it. Moreover, executing the correct system call in the kernel is required to properly unlink the patch to retrieve the correct CFI state gain. Only when executing the correct system call in the kernel the CFI state gets properly unlinked to retrieve the correct CFI state gain. Performing first-stage linking already provides protection for requirements **R1** and partly **R2**.

Second-Stage System Call Linking

After linking the system call to the CFI state in the user-space of the program, the system call is executed, and the execution switches into the kernel. Via dispatching code and the selected system call in the general-purpose register `w8`, the kernel selects the correct system call function and executes it. At the beginning of every particular system call function, we verify if the CFI state matches the value for the executed system call function. At the end of each system call function, we apply a second patch, *i.e.*, the *second-stage* linking to the CFI state, confirming that the previously selected system call was really executed. This patch value is computed dynamically during the execution of the syscall with the help of `PACIA`, the system call number, and a modifier value, which is different than the modifier from the first-stage linking. By using a different modifier and computing a different patch value, we avoid canceling out the impact on the CFI state when applying it to it. Immediately after applying the computed patch value to the CFI state, we perform a check operation to ensure a valid CFI state. The second-stage linking step ensures that both requirements **R2** and **R3** are fulfilled.

In Figure 6.3, we summarize SFP's system call protection. A user program performs the first-stage linking and patches the CFI state with a statically computed syscall patch to link the execution of a system call. The execution transitions to the kernel, which executes the desired system call function. At the end of the system call, the kernel performs the second-stage linking operation, followed by a CFI check operation. The second-stage linking operation only succeeds when the correct system call is linked to the CFI state. As a result, SFP's approach translates system call errors, independent of how they occur, to CFI state errors, which eventually are detected through the checking policy of the selected CFI protection scheme. Note that Figure 6.3 includes CFI checks at the beginning and end of the syscall to immediately detect a wrong syscall when entering the kernel and after the syscall's execution. Consequently, we reduce the detection latency, and no harmful syscall can be performed. The user-space application then continues the execution with the CFI state at the end of the executed syscall. The CFI state in the user-space is only correct if the kernel executes the right syscall; thus, the backward edge from the kernel to the user-space is protected.

6.3.3 Dynamic Instrumentation

Existing SFCFI protection schemes such as FIPAC or related work [Cle+17; NSM21; Wer+18] use a static post-processing or encryption phase. A dedicated post-processing tool recovers the control-flow, computes the patch and check values, and modifies the program. The static approach with a single encryption key leads to the fact that all executions of the same program use the same CFI values, e.g., patches, updates, or checks. By observing the used CFI-related values, attackers can more easily craft valid CFI states to bypass the control-flow protection.

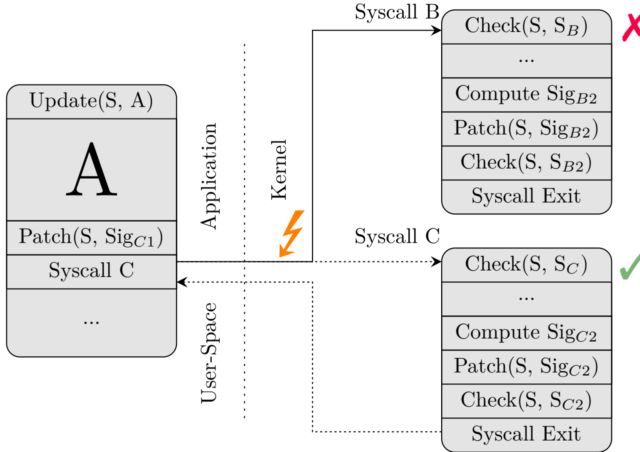


Figure 6.3: System Call protection in SFP. Before a syscall, we cryptographically bind the syscall to the CFI state for later verification and second-stage linking in the kernel.

In SFP, we overcome this limitation by splitting up the toolchain and integrating the CFI instrumentation into the kernel. Instead of statically performing the post-processing step at compile-time, we integrate the post-processing tool into the operating system. When starting a program, the Executable and Linkable Format (ELF) loader of the OS identifies a CFI-instrumented program. It generates a random ARM PA encryption key and stores it in the process task structure. The ELF loader then performs the per-program call unique CFI instrumentation and computes the expected CFI state and all patch values needed to handle the control-flow. The CFI states are stored along with the process task structure within the kernel. With this mechanism, subsequent calls to the same program create different encryption keys. As a result, it guarantees that different CFI values are generated on each new program start, *i.e.*, fulfilling requirement **R4**.

Kernel Checking Policy

The instrumented programs of SFP do not contain any CFI check operations. Instead, in SFP, we develop a novel CFI checking policy at the edge of the operating system. Due to dynamically instrumenting the program when starting it, the operating system knows exactly the expected CFI state for every location of the program. When a user program now enters the kernel, e.g., due to a system call instruction, the kernel, which has access to both the user program state and the expected CFI states, can verify them. If the current CFI state matches the expected state, the system call continues. However, if the CFI state deviates from the expected state, a CFI error is detected, and the operating system aborts the program execution. A CFI check at the of the syscall confirms the execution of the right syscall. Apart from system calls, a user program can

enter the operating system also via different execution paths. We include the same checking policy when a timer interrupt is raised and the kernel is entered.

6.4 Implementation

The prototype implementation of SFP consists of two parts. First, we develop a toolchain to automatically compile and instrument arbitrary C-programs with CFI, including a custom runtime library. Second, we modify the kernel of the Linux operating system to include the system call verification, the new checking policy, and the dynamic instrumentation on the program start.

6.4.1 Toolchain

This section describes our toolchain capable of compiling arbitrary C programs with SFP.

Compiler

We base the toolchain on the modified compiler of FIPAC [SNM22a], which is based on the LLVM [LA04] compiler framework. We adopt the AArch64 backend of the compiler to instrument the control-flow and embed control-flow meta information in a custom section of the ELF binary. The compiler inserts the updates for every basic block, inserts patches for control-flow merges, and also deals with call instructions. Our modified compiler emits a running ELF binary but leaves all patch values for control-flow merges, and system calls to be zero. The necessary post-processing step is shifted to the operating system, which computes all patches at the program start. Note that the instrumented program does not contain any check instructions, as they are part of the transition to the operating system and are performed in the kernel.

C Standard Library

System calls are typically invoked via wrapper functions provided by the standard library of the programming language. This prototype toolchain uses a CFI-instrumented version of the *musl* [Fel22] C standard library. The standard library provides wrapper functions for all system calls or uses system calls directly in different library functions. We identify every system call in the musl standard library and insert the necessary patch sequence containing an immediate load and the XOR-based state update ahead of executing the system call. Listing 6.2 summarizes the first-stage linking, where the immediate value for the `mov` instruction is zero. When starting the binary, the operating system computes the actual patch value for this system call and fills out the correct load value.

```
1 basic_block:
2   ...
3   mov x0, #...         ; arguments for B system call
4   mov x15, #0          ; Zero system call patch
5   eor x28, x28, x15    ; Perform a CFI state update
6   mov w8, #syscall_B   ; system call number for B
7   svc #0               ; Jump to kernel
```

Listing 6.2: Patched system call in the musl standard library.

6.4.2 Kernel Support

SFP requires minor modifications to the operating system. We base the prototype of SFP on the Linux kernel in version 5.15.32 [Ker22a].

Dynamic Instrumentation on Program Start

On program start, when an instrumented ELF binary is started, SFP performs the per-program instrumentation of the program. First, the kernel generates a random encryption key used for the ARM PA instrumentation. With the help of control-flow metadata, which is stored along with the ELF binary in a metadata section, we compute the CFI state throughout the program and fill the necessary patch values for justifying signatures. Furthermore, we compute the syscall- and key-dependent patch values that are used to protect the system call interface. For every system call in the program, we compute its Pointer Authentication Code (PAC) based on the system call number and user-space program unique modifier. The resulting PAC value, which is not guessable by the attacker, is filling out the immediate patch value before the syscall.

As discussed, the instrumented program does not contain dedicated CFI check operations, as they are performed when entering the kernel. Instead, we store the expected CFI state for each program location in the task's kernel structure. To reduce the storage overhead, we use a `RangeMap`, to only have one entry for a contiguous range of states, where it does not change.

System Call Verification

During the system call, the user program updates the CFI state with a statically computed cryptographic patch value that depends on the system call number. The verification that the correct system call gets executed happens in the kernel. After the system call jumps into the kernel, a dispatcher code selects the correct system call function to be executed. At the end of every system call function in the kernel, we perform the second-stage linking. Based on the system call number, we dynamically compute a second patch value dependent on the currently executed system call. In Listing 6.3, we summarize this operation sequence, where we perform the second-stage linking within the kernel. To retrieve a cryptographically secure patch value, we exploit the `PACIA` instruction of ARM PA, which takes the

```
1 syscall_A:
2   ...
3   mov x16, #1           ; Load kernel modifier
4   pacia x8, x16         ; Compute system call patch
5   eor x28, x28, x15     ; Perform 2nd stage linking
6   and x28, x28, #0xffffffff00000000 ; Clear syscall
7   ret                   ; number from CFI state
```

Listing 6.3: Dynamically computing the system call patch and removing it from the CFI state at the system call end.

system call and a modifier as input operands. Note that the modifier used for the kernel update of the CFI state is different from the one used for the first-stage linking in the user program. This property is essential to avoid attackers being able to skip system calls entirely since patching the CFI state twice with the same value would cancel out and has no permanent effect on the CFI state. We finally apply the computed patch to the CFI state and clear the lower bits from the system call.

Checking Policy at the Kernel Boundary

Whenever a user program enters the kernel, SFP performs a CFI check to validate if the current CFI state still matches the expected state. We perform CFI checks on two entering points: During a system call and when a timer interrupt is raised. With the help of the CFI states stored in a `RangeMap` within the process structure and the knowledge of the program's current program counter, we look up the expected CFI state for the program location. If the current CFI state, stored in the register `x28` of the user program state, diverges from the expected state, a CFI error is raised, and SFP stops the program execution. For syscalls, we perform a second CFI check at the end of the syscall function in the kernel to ensure the syscall was really executed.

6.5 Evaluation

In this section, we first evaluate the security of SFP and show how it provides protection and the defined threat model. Second, we evaluate the functionality and the performance overhead of the prototype implementation.

6.5.1 Security Evaluation

We analyze the security guarantees of SFP and show how different attacks within the threat model are mitigated.

Control-Flow Hijacks in the User-Space or Kernel

SFP provides CFI protection for the user-space application based on the selected underlying CFI protection scheme. The prototype uses FIPAC, a basic block granular CFI scheme, protecting all direct/indirect branches as well as direct/indirect calls. The protection domain includes the C standard library, which is fully CFI instrumented. Consequently, an attacker cannot redirect syscalls in the user-space application by redirecting the control-flow to a different wrapper function of the standard library. Control-flow attacks in the kernel are detected via the kernel's internal CFI protection scheme.

Skipping a System Call

When skipping a system call instruction, *i.e.*, the `svc` instruction, the first-stage linking already occurred. Subsequently, the skipped system call misses the second-stage linking from the kernel, which yields a wrong CFI state, which is detectable through the CFI checking policy.

However, if the entire system call instruction sequence is skipped, *i.e.*, first-stage patching and the syscall instruction are omitted, the hijack is still detectable. As both patch operations are missing on the CFI state, the state is wrong again, and a subsequent CFI check, e.g., when the program gets scheduled, detects the invalid state. In both cases, SFP transforms the skipped system call into a CFI error, which manifests itself in a wrong CFI state, which is detectable.

Changing a System Call

A fault on the register containing the system call number, or a combined software- and fault-based attack, in which the attacker controls the register used to execute the system call, redirects the system call to a different one. SFP protects against both attacks. By applying the first-stage linking to the CFI state, the correct system call is already bound to its future execution. Manipulating the system call register, e.g., due to a fault or software vulnerability, leads to applying the wrong system call patch to the CFI state. When the system call is executed, the CFI state for that program differs from the expected state, and the CFI check in the kernel detects the problem and aborts the program.

To bypass a system call, the attacker only has a single chance to change the system call number and manipulate the previous system call patch to the correct one for this location. However, the system call patch is protected via the secret ARM PA key, which the attacker cannot access. Guessing the PAC leads to a probability of $p = \frac{1}{2^{15}} = 0.0031\,\%$ for getting the correct patch value, where 15 is the configured PAC size of our prototype implementation. Furthermore, due to the dynamic instrumentation on the program startup, the system call patches always differ between subsequent calls of the same program. As a result, the attacker cannot learn new patch information between subsequent program calls.

6.5.2 Functional Evaluation

To validate the functional correctness of SFP, we emulate the execution on the functional simulator QEMU [QEM20] in version 7.0.0. Since this simulator currently only supports ARM PA from ARMv8.3-A, we extend it to include ARM PA of ARMv8.6-A to support the CFI protection. The functional evaluation runs the modified Linux kernel from the prototype and can start and execute instrumented programs where all system calls are protected. Within the kernel, the functional simulator performs the second-stage linking and a CFI check to verify the execution of the correct syscall.

Fault Simulation

To verify the functionality of the countermeasure, we emulated skipping a system call and modifying the system call number. In both cases, SFP detects the attack through the next CFI when entering the kernel. Due to the manipulation, the CFI state became invalid, and the CFI checking policy stops the program execution.

6.5.3 Performance Evaluation

At the time of evaluation, there is no publicly available system supporting ARMv8.6-A needed to run FIPAC. However, to conduct the performance evaluation and to measure the performance impact of SFP, we emulate the runtime overhead of ARM PA instructions. Therefore, we base the performance evaluation on a Raspberry Pi 4 Model B [Ras20] with 8 GB RAM configured with a fixed Central Processing Unit (CPU) frequency of 1.5 GHz. The Raspberry Pi contains an ARM Cortex-A72 CPU based on ARMv8-A but without Pointer Authentication. To emulate the overhead of ARM PA instructions, we replace them with a PA-analogue instruction sequence, *i.e.*, four consecutive XORs. Related work [Lil+19; Lil+21] evaluated this instruction sequence to mimic the timing behavior of an ARM PA instruction.

Microbenchmark

To evaluate the overhead of SFP executing system calls, we perform a simple microbenchmark. Our benchmark measures the syscall latency of the `getpid` system call, which is a side-effect-free syscall and is used in related works to benchmark the syscall execution path [Bue19; Can+21]. We execute the system call 10 million times and measure the system call latency via the processor's inbuilt cycle counter. Figure 6.4 summarizes our evaluation results, showing the syscall latency in different kernel configurations. On the plain, unmodified Linux kernel, we measure an average system call latency of 2131 cycles. When integrating the system call verification alone, the latency rises to 2144 cycles. Furthermore, with the CFI checks alone enabled, the latency increases to 2175 cycles. When both are active, we measure a system call latency of only 2185 cycles, impacting the system call latency by only 1.9 %.

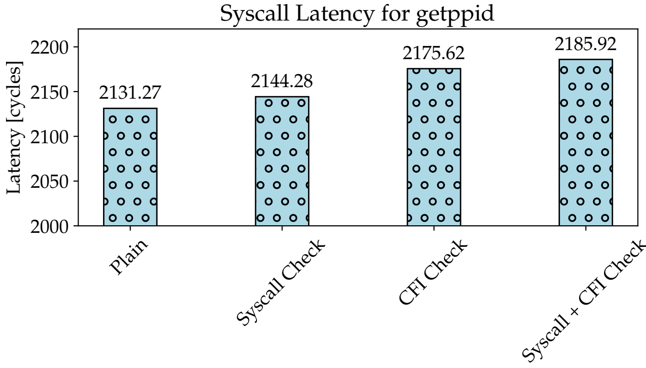


Figure 6.4: The microbenchmark shows the system call latency of the `getpid` system call for different kernel configurations. SFP increases the system call latency by 1.9 %.

Macrobenchmark

To demonstrate the applicability of SFP on a larger scale, we perform a macrobenchmark on real-world applications. We compiled the SPECspeed 2017 [Sta19] benchmark with our toolchain, including only C-based programs. In Figure 6.5, we plot the runtime overheads in two different configurations compared to the plain uninstrumented code. First, we only include the dynamic verification, including the new CFI checking policy, that verifies the CFI state of user programs when entering the kernel. Second, we include the syscall protection based on the two-stage linking approach together with the previously evaluated CFI checking policy.

During the evaluation, we measure a geometric mean overhead of 18.8 % for the new CFI checking policy and 20.6 % with the system call protection and CFI checking policy in place. Based on the evaluation of the SPEC 2017 benchmark, we only measure a difference in the overhead of 1.8 % between the pure CFI protection and the full system call protection of SFP. This result shows that the dominating part of the overhead comes from the CFI instrumentation, not from the system call protection. Thus, reducing the overheads of the CFI protection directly influences the performance of SFP.

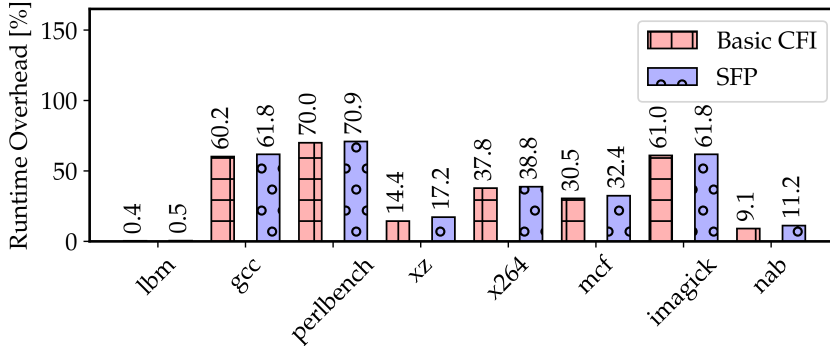


Figure 6.5: Macrobenchmark shows the performance impact of SFP on the SPEC 2017 benchmark. We evaluate the impact of CFI only and SFP, including the system call protection.

6.6 Discussion

This section discusses prototype limitations and shows how SFP is compatible with other CFI protection schemes.

6.6.1 Dynamic System Call Instrumentation

In our prototype, we manually instrument all system calls of the C standard library with the necessary patch instructions, consisting of a load of an immediate patch value followed by applying the patch value to the CFI state. The immediate value is zero and is set to its concrete value during the dynamic instrumentation of the startup phase of the program. In a future version of SFP, we could instrument the compiler to detect syscall instructions, *i.e.*, `svc`, and then automatically insert the necessary patch sequence. This enhancement would also include cases where syscalls are invoked manually without the wrapper functions of the standard library.

6.6.2 CFI Checking Policy Extension

SFP currently performs CFI checks when entering the kernel through a syscall or a timer interrupt. A future version of this work can extend the CFI checking policy to include all interrupts of the system. Our microbenchmark shows adding new CFI checks adds minimal overhead to the syscall latency. Thus, adding additional CFI checks for all interrupt handlers are expected to have minimal impact on the system performance.

6.6.3 Compatibility

Although SFP uses FIPAC as the underlying CFI protection scheme, the design or the protection mechanism of SFP is generic and compatible with different CFI

protection schemes. To apply the protection of SFP to a different protection scheme, two things are required. First, the CFI protection scheme must be stateful, and there must be a possibility to manipulate the state, e.g., via standard or custom instructions, to inject the system call patch. Second, it is necessary to be able to dynamically compute a second system call patch required for the second-stage linking in the kernel. With these requirements, SFP is compatible with existing CFI protection schemes such as Sponge-Based Control-Flow Protection (SCFP), SOFIA, or any other state-based CFI protection scheme.

6.7 Related Work

There already exists related work in the context of CFI protection schemes or systems that enforce the correct system call sequence.

For example, SCFP [Wer+18] and SOFIA [Cle+17] are hardware-assisted control-flow integrity protection schemes on the instruction level. They encrypt the program's instruction stream at compile-time and perform a fine-granular decryption during runtime to retrieve the correct instruction sequence with the help of intrusive hardware changes. However, these protection schemes are designed for embedded use cases without running a rich operating system such as Linux. Thus, these schemes don't provide protection for the system call interface since it does not exist. However, the design of SFP is generic; thus, system call protection can be implemented on SCFP or SOFIA, as discussed in Section 6.6.3.

SFIP [Can+22] implements coarse-grained syscall flow protection for user-space applications. They statically identify the possible transitions between different syscalls at compile-time and then enforce that at runtime. SFIP adds an average overhead of around 1.9 % on a set of macrobenchmarks, which is very similar to the overheads of SFP. However, the overheads of SFP provide a much larger scope of protection when a CFI protection scheme is already deployed. Since SFIP only considers software attackers in their threat model, their applicability to fault attacks is limited.

In the context of software-only attacks, there exists other work that tries to learn and enforce the correct sequence of system calls. For example, machine learning approaches are used to predict the syscall sequence and to detect intrusion [Lv+18; ZPZ08]. Linux also has inbuilt support to minimize the kernel surface that a user application can reach. The Secure Computing interface, or Seccomp BPF [The21] of Linux, allows the user program to install filters that limit which syscalls the application can use. When a system call is executed, the kernel first verifies if the system call is allowed and then executes it or not. However, within the limited set of system calls, faults can still bypass them or hijack the execution to a different system call in the allowed set.

6.8 Conclusion

In this chapter, we presented SFP, a protection mechanism that provides system call flow protection on top of ordinary CFI, protecting the interface to the kernel against both software and fault attacks. We show that an already employed CFI protection scheme can be used as a versatile tool to protect the system call interface to the kernel. Furthermore, we present a new CFI checking policy at the edge of the kernel to verify the CFI state for all transitions to the kernel. Combined with a dynamic CFI instrumentation on program startup, the attacker cannot learn CFI or system call-related information from subsequent program executions. We showed a prototype implementation comprising an LLVM-based toolchain to automatically instrument arbitrary programs and protect all system calls. A modified Linux kernel running on a Raspberry Pi evaluation setup to show the applicability to real-world programs. Our evaluation based on a microbenchmark and on the SPEC 2017 application benchmark shows an average runtime overhead of 20.6 %, which is only an increase of 1.8 % compared to plain CFI protection. This slight increase in the performance impact shows the effectiveness of SFP for protecting all system calls of a program.

The mechanism of SFP is generic in the sense that it can be protected with different CFI protection schemes such as SCFP or SOFIA. Compared to other protection schemes that provide protection for the order of executed syscalls, SFP only adds minimal overhead on an already deployed protection mechanism, *i.e.*, CFI.

7 Secure Comparisons and Conditional Branches for CFI

Control-Flow Integrity (CFI), as discussed in Chapter 5, is one pillar for secure software execution in the presence of fault attacks. While FIPAC is one concept for fine granular CFI for commodity systems, there are plenty of other CFI protection schemes with different trade-offs. For example, there are other pure software-based protection schemes such as CFCSS [OSM02] or hardware-based approaches like Sponge-Based Control-Flow Protection (SCFP) [Wer+18] and SOFIA [Cle+17], which all protect against fault-based control-flow attacks. Unfortunately, all these schemes have one common unsolved problem. While they provide fine-granular protection of the Control-Flow Graph (CFG), *i.e.*, on the basic block or even instruction level, they lack the protection of conditional branches, the point where data and control-flow merge. We discussed the details of the control-flow and conditional branches in Chapter 3.

A conditional branch determines the program flow of the executed software based on a flag or based on the comparison of two values. While the basic functionality of a conditional branch is quite simple, the correct execution is highly critical for the security of computer systems. In the end, it is a conditional branch that decides whether or not an entered password is considered correct, a system update is performed, a signature check is considered successful, or a user is granted access to a privileged function. The security implications are huge if a critical program flow decision is not taken correctly and, for example, unauthenticated software is executed [Ris17].

Under normal conditions, conditional branches execute correctly, *i.e.*, the branch is performed according to the comparison. However, when considering faults inside the threat model, this assumption is not necessarily true anymore,

as discussed in Chapter 2. A fault on the data, the comparison, or on the actual branch can redirect the execution to the wrong program location.

Even the presence of strong CFI protection only ensures that one of the two possible execution paths and no completely different path is taken after a conditional branch. However, CFI protection schemes do not protect the decision of which path is taken against fault attacks. For conditional branches also, the data influences the outcome of its execution. Thus, it is important that also the processed data is protected with redundancy mechanisms. For example, [FSS09] shows how to protect variables during arithmetic operations using AN-codes. However, as also pointed out in [Hof+14], such schemes only protect data values and their processing and no branching operations based on the data.

Today, there is a gap. There exist CFI and data protection schemes against fault attacks. However, there is no method of linking them efficiently such that the decision on which execution path to take is protected at the same level as the control-flow and the processing of the data.

Contribution

In this chapter, we close the existing gap by providing protection for conditional branches which is encoding-based, like the redundancy schemes for data and CFI. We develop a generic approach to link a redundant comparison operation to the CFI state. Thus, we merge the domain of data redundancy with CFI and protect conditional branches.

We base our concrete instantiation of protected conditional branches on arithmetic AN-codes. In this context, we develop new comparison algorithms for all comparison predicated that use the data redundancy to compute a redundant condition value. Our algorithms preserve the Hamming distance of the encoding scheme, thus, computing condition values that are secure against fault attacks. By linking this protected condition value to the CFI state, we bind the control-flow and execution to the previously computed comparison and protect the actual conditional branch.

To showcase the applicability of the design, we develop an LLVM-based toolchain that automatically protects conditional branches. On annotated functions, we automatically provide branch and data protection by transforming all branch-related data and operations to the AN-code domain and inserting protected comparison algorithms, which result is linked to the CFI state. Finally, we evaluate the design isolated and present a use case based on a secure boot scenario. Summarized, our concrete contributions are as follows:

- We present a generic solution that closes the gap of unprotected conditional branches in the presence of a CFI protection scheme. Conditional branches are protected by linking a redundant comparison result with the redundancy of the CFI protection scheme.
- We show that we can use AN-codes to efficiently perform a redundant comparison of encoded values which preserves the redundancy.

- We present an LLVM compiler extension to automatically identify and protect conditional branches based on the concept of AN-codes. We provide experimental results showing that the overhead in terms of code size and runtime is lower than state-of-the-art duplication schemes. Furthermore, a bootloader application can efficiently be protected with 2.4 % code overhead and with less than 0.1 % runtime overhead.

Scientific Contribution

Chapter 7 is primarily based on the following publication that was presented at DATE 2018 in Dresden (Germany).

Robert Schilling, Mario Werner, and Stefan Mangard. "Securing conditional branches in the presence of fault attacks." In: *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018.* IEEE, 2018, pp. 1586–1591. DOI: 10.23919/DATE.2018.8342268

I am the main author of this paper, developed the algorithm, integrated that into a compiler, and performed all the evaluation steps. Mario Werner contributed to this work by writing the background section. Stefan Mangard supported this work through technical discussions and by writing parts of the introduction.

Outline

The structure of this chapter is as follows. We define the threat model and discuss existing countermeasures for conditional branches in Section 7.1. In Section 7.2, we present how we protect conditional branches in this setting. We discuss a novel approach to compute a redundant comparison result used for protected conditional branches in Section 7.3. In Section 7.4, we present a compiler extension to protect conditional branches automatically and evaluate the overhead. Section 7.5 analyzes the security of the countermeasure, and finally, in Section 7.7, we conclude this chapter.

7.1 Threat Model and Related Work

In this section, we define the threat model, discuss existing methods to protect branches, and show the limitations of existing CFI protection schemes in the context of protecting conditional branches.

7.1.1 Threat Model

We consider an active attacker that is capable of inducing faults in the system. Faults can occur once or multiple times with multiple bits modified. We assume the presence of a fine-granular state-based CFI protection scheme, *i.e.*, on the

basic block or even on the instruction level. This can be achieved with the FIPAC from Chapter 5 or hardware-based CFI protection schemes [Cle+16; Wer+18; WWM15]. Furthermore, we assume the data of the program to be encoded redundantly.

7.1.2 Conditional Branch Protection via Re-checking

One way of protecting a conditional branch against fault attacks is to check the condition for the branch again after the branch has been taken [Ris23b]. This duplication approach increases security and can be scaled to an arbitrary order. However, this duplication approach leads to significant overheads on one side, and it can be attacked by inducing multiple times the same fault. The options for creating diversity by using different branches to make attacks harder are limited. Typically, the same hardware multiplexer for all branches decides which address is loaded next and remains as a single point of failure.

7.1.3 Conditional Branches in the Context of CFI

CFI protection schemes in the context of fault attacks rely on an internal state S, which gets updated at a certain granularity, e.g., on basic block or instruction granularity. For FIPAC, this occurs once for every basic block. Independent of the concrete CFI protection scheme, control-flow transfers like conditional branches require special treatment. On control-flow transfers, the internal state S diverges because the instruction stream diverges. When the control-flow graph merges at a later point in the program, the CFI state S also needs to merge. To support this, CFI protection schemes either use correction values or replace the state.

Although CFI protection mechanisms can deal with conditional branches, they cannot protect them. Such a scheme only ensures that one of the correct successor's blocks is executed after the branch, but the correct selection is completely unprotected, leaving a single point of failure. A single bitflip, for example, during the comparison operation of a conditional branch, can redirect the control-flow to the wrong successor basic block, which is still valid in terms of CFI.

7.2 Protecting Conditional Branches against Fault Attacks

A conditional branch consists of two operations: a comparison step and a branch operation. The comparison takes two inputs, x and y (in the general case, two register values), compares them with a predicate P (e.g., $<$), and results in a 1-bit signal indicating if the comparison is true or false. Typically, this signal is part of the Central Processing Unit (CPU) flags. The branch operation takes this signal and decides how to update the Program Counter (PC), which can end up with two different values PC_1 and PC_2, depending on whether the branch

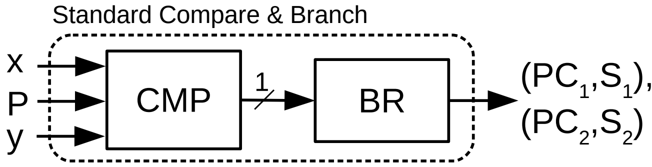


Figure 7.1: Conditional branch with CFI state.

was taken or not. Depending on the CPU architecture, the compare and branch operation is implemented as a single instruction or as two dedicated instructions.

In the presence of a fine-granular CFI protection mechanism, *i.e.*, on the basic block or even on the instruction level, conditional branches work slightly differently. Again, there is a comparison and branch operation, as shown in Figure 7.1. However, the CFI protection mechanism contains a dedicated internal state S for each value of the program counter, which is updated when executing the conditional branch. Here, the output of a conditional branch is two different PC values PC_1, and PC_2, with their corresponding CFI states S_1 and S_2.

However, even in the presence of a CFI protection scheme, there are three different error sources, which are not protected and can lead to a wrong execution:

1. *Faulting the operands.* Modifications on the branch operands or any data that leads to the comparison can result in a wrongly executed conditional branch.

2. *Faulting the comparison.* The value deciding whether a conditional branch is taken or not, the condition signal, is a 1-bit signal. An attacker being able to control this signal precisely can change the execution of the conditional branch.

3. *Faulting the branch.* A fault modifies the execution of the branch such that the branch is taken, although the condition value says otherwise or vice versa.

To protect conditional branches, we assume that data and all performed operations on it are encoded redundantly, e.g., via AN-codes. We generically address the latter two points as follows: We first use a redundantly encoded condition computation to ensure the integrity of the condition value. This encoded comparison takes two encoded values, x_c, and y_c, a comparison predicate P, and outputs a redundantly encoded condition result, which Hamming distance is large enough to maintain the same security level throughout the whole conditional branch. The comparison predicate P does not require redundancy by means of encoding since a different predicate uses a different expected condition value. We then use the standard compare and branch mechanism that compares the redundant comparison result computed in software with one of the expected condition values.

Without further measure, this would introduce an intermediate unprotected 1-bit signal to decide if the branch should be taken. To mitigate this problem, we

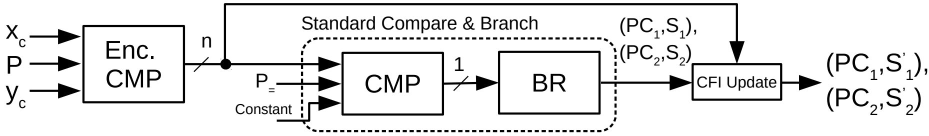


Figure 7.2: Protected conditional branch with state update and n-bit redundantly encoded comparison.

exploit the redundancy of the encoded comparison result and merge this value as part of the CFI state update into the redundancy of the CFI protection scheme (Figure 7.2). Only if the condition is computed correctly and the branch is executed correctly the states for S_1' and S_2' are correct. This approach eliminates the single point of failure present in state-of-the-art CFI protection schemes by not relying on a 1-bit condition value but on a redundantly encoded condition value linked with the CFI state. The comparison is protected by using an encoded comparison operation that yields a redundant result. The final conditional branch is protected by linking the redundant condition value with the CFI redundancy. Fault attacks in both cases yield an invalid state S, which is detectable.

7.2.1 Requirements for CFI Protection Scheme

Using an encoded comparison operation ahead of an ordinary conditional branch makes this design modular and flexible allowing different encodings with different security levels to be used at various program locations. The *only* requirement for the CFI protection is the ability to merge a value from the register file into the internal state CFI state. Consequently, our protection mechanism makes it attractive and compatible with many different CFI protection schemes, including pure software-based designs such as FIPAC or hardware-based approaches such as SCFP [Wer+18]. We further discuss the compatibility with these protections schemes in Section 7.6.

7.3 Protected Comparisons with AN-Codes

In this work, we focus on AN-codes [Bro60], which are well-suited for fault protection [FSS09] and natively support different arithmetic operations. The details for this encoding scheme, including their arithmetic properties, are discussed in Section 2.5.2. However, as discussed by Hoffmann et al. [Hof+14], AN-codes alone is not sufficient because conditional branches are still a single point of failure.

In this section, we discuss a redundant comparison framework which is exploiting the arithmetic properties of AN-codes, which adheres to the interface definition in Equation (7.1). The inputs, the internals, and the output are encoded such that there is no single point of failure. The two possible outputs of the encoded comparison operation should have a Hamming distance larger or equal than a constant D, where D denotes the minimum security level in

bits of the data encoding and the CFI redundancy. Furthermore, we want to avoid the all-zero and all-one condition results because faulting to these values is easier than to others due to the hardware implementation (e.g., the reset line of a register can initialize its value to zero).

$$\begin{aligned} \text{condition} \leftarrow & \text{EncodedCompare}\,(P, x_c, y_c) \\ & \text{with condition} \in \{C_1, C_2\} \text{ and} \\ & \text{Hamming distance}\,(C_1, C_2) \geq D \end{aligned} \tag{7.1}$$

AN-encoded data can be compared using a standard compare instruction. However, such a simple comparison removes all redundancy, and the result is stored as a 1-bit signal inside the CPU. Hoffmann et al. [Hof+14] found this issue during fault simulation. Instead, we compute the comparison and preserve the redundancy of the AN-codes avoiding this single point of failure.

To compute the $x_c < y_c$ comparison, where x_c and y_c are AN-coded variables, we first start with a subtraction, as this is the basis for all comparisons, including standard comparison operations. Based on the sign of this subtraction result, we get the information shown in Equation (7.2). However, we cannot directly use the sign bit because it is not redundant. The challenging task is performing an entropy compression, where we map the encoded positive difference values to C_1 and all encoded negative values to C_2. Additionally, we want to maximize the Hamming distance between C_1 and C_2, yielding a redundant comparison result.

$$x_c - y_c \begin{cases} \text{positive if} & x_c \geq y_c \\ \text{negative if} & x_c < y_c \end{cases} \tag{7.2}$$

Our approach *arithmetically* computes this entropy compression yielding a comparison result that preserves the redundancy of the AN-code. When looking at the difference in Equation (7.2), the congruence $0 \equiv (x_c - y_c) \bmod A$ is valid because AN-codes are closed under subtraction in a signed representation. However, when interpreting the AN-code congruence in an unsigned representation, this destroys the congruence for negative differences. For a positive difference, on the other hand, the unsigned representation does not change anything. By intentionally destroying the AN-code congruence for negative numbers due to casting to unsigned, we are able to separate the two cases of Equation (7.2), yielding two different values. Using 32-bit data types, the unsigned interpretation x_u of a signed negative value $x_s < 0$ in the twos-complement representation is computed as $x_u = 2^{32} + x_s$. We exploit this property of twos-complement encoded negative numbers for the required entropy compression. First, the difference is cast to an unsigned value. This does not change the value of the difference if it originally was positive. Negative values change according to the twos-complement, where the AN-encoded difference becomes invalid. In Equation (7.3), we show the conversion from the signed AN-code to the unsigned representation for negative values of the difference.

Algorithm 3: AN-encoded < comparison.

Data: $x_c, y_c \in$ AN-code, $0 < C < A$.
Result: $cond \in \{C_1, C_2\}$.
begin
    diff ⟵ (unsigned) $x_c - y_c + C$
    cond ⟵ diff % A
end

$$(x_c - y_c)_u = 2^{32} + (x_c - y_c) \tag{7.3}$$

$$= 2^{32} + A \cdot (x - y) \tag{7.4}$$

When applying the AN-code congruence to that value by using a modulo operation with A, we obtain a dedicated value for the negative difference, as shown in Equation (7.5).

$$\left(2^{32} + A \cdot (x - y)\right) \% A = 2^{32} \% A \tag{7.5}$$

The relation described before only holds true for the negative difference. For a positive difference in Equation (7.2), the AN-code congruence still returns zero. However, as discussed before, having a comparison result that is zero is not favorable. We avoid this zero comparison result for the true case by adding constant $0 < C < A$ to the difference before we compute the remainder (this also changes the comparison result for the false case).

Algorithm 3 summarizes how the encoded less-than comparison is computed. The comparison result *cond* holds the value $2^{32}\%A + C$ if x_c is less than y_c or the value C if x_c is larger or equal to y_c. Any modification (e.g., due to a fault) to the operands such that their AN-code gets invalid results in a different comparison result, making it invalid.

The same scheme can be applied to compute a $\leq$, $>$, and $\geq$ comparison by swapping the operands in the first subtraction and swapping the symbols for the true and false cases. In Table 7.1, we summarize the comparison result values and the subtraction order for all comparison predicates (except $=, \neq$) for 32-bit data types.

7.3.1 Protected Equal and Not-Equal Condition Computation

Computing the $=$ and $\neq$ condition using AN-codes can be assembled by combining the $\leq$ and $\geq$ condition. The $=$ condition is true if both conditions are true and false if only $\leq$ is true or $\geq$ is true. Both conditions cannot be false at the same time. We combine these conditions using an addition operation. Using the

Algorithm 4: AN-encoded = and ≠ comparison.

Data: $x_c, y_c \in$ AN-code, $0 < C < A$.
Result: $cond \in \{C_1, C_2\}$.
begin
```
    diff1 ⟵ (unsigned) x_c − y_c
    diff1 ⟵ diff1 + C
    rem1 ⟵ diff1 % A
    diff2 ⟵ (unsigned) y_c − x_c
    diff2 ⟵ diff2 + C
    rem2 ⟵ diff2 % A
    cond ⟵ rem1 + rem2
```
end

condition values for $\geq$ and $\leq$ from Table 7.1, the sum of both true values is $2 \cdot C$. The false case is the sum of one true and one false case resulting in the condition value $2^{32}\%A + 2 \cdot C$. The algorithm to compute the $=$ or $\neq$ condition is shown in Algorithm 4.

7.3.2 Parameter Selection

For the comparison algorithms, we used 32-bit registers and chose A to be 63877 (a *super*-A according to Hoffmann et al. [Hof+14]). This A maximizes the functional value for 16-bit data and has a minimum Hamming distance of *six* between all code words, allowing the code to detect up to 5-bit errors. We then chose C such that it maximizes the Hamming distance between the true and false symbol for one comparison. For the $=$ and $\neq$ comparison, we select $C = 14991$, and for the $<, \leq, >, \geq$ comparison, we select $C = 29982$. With both constants, we reach a maximum Hamming distance D of 15-bit between the comparison values.

Table 7.1: Condition values for encoded $<, \leq, >, \geq$ condition values.

Predicate	Subtraction	True Value	False Value
$>$	$y_c - x_c$	$2^{32}\%A + C$	C
$\geq$	$x_c - y_c$	C	$2^{32}\%A + C$
$<$	$x_c - y_c$	$2^{32}\%A + C$	C
$\leq$	$y_c - x_c$	C	$2^{32}\%A + C$

7.4 Implementation and Evaluation

To demonstrate that our protection mechanism for conditional branches is sufficiently performant for real-world applications, we implemented the required transformations into a C compiler and evaluated the concept using a simulator for the ARMv7-M Instruction Set Architecture (ISA), *i.e.*, used by ARM's Cortex-M3 processors. Although this countermeasure is fully compatible with FIPAC from Chapter 5, we use a software-centered CFI protection scheme that is based on the Generalized Path Signature Analysis (GPSA) implementation from [WWM15]. We opted for a pure software implementation of our branch protection scheme, which does not require hardware modifications on top of the used CFI protection scheme.

7.4.1 Implementation

We implement the protection mechanism for protected conditional branches as an extension to the LLVM [LA04] compiler framework. The resulting modified compilation pipeline is depicted in Figure 7.3.

The compiler front end contains a new LLVM-specific function attribute (*i.e.*, `protect_branches`) to annotate functions that require AN-code protection for their conditional branches. The AN-code instrumentation is performed in the middle end. There, the optimized Intermediate Representation (IR) is preprocessed by a custom *Loop Decoupler* pass which separates loop induction variables from the use in arithmetic expressions or memory accesses, and a *Lower Select/Switch* pass simplifying the IR for the subsequent *AN Coder*. The *AN Coder* pass transforms all instructions, which end up in the comparison operation of a conditional branch to the AN-domain. The affected instructions are identified by slicing the program with respect to the operands of the conditional branches. Moreover, the AN-code-based *encoded compare* algorithm is added here. Up to this point in the compiler pipeline, all transformations are independent of the target architecture and CFI protection scheme.

The *CFI Instrumentation* pass, which is located in the back end of the compiler, is, in fact, the only architecture and CFI-specific part of our implementation. This pass performs the CFI instrumentation itself and adds the *state updates* (as discussed in Section 7.2) to the conditional branches.

7.4.2 Cost Analysis

Qualitatively speaking, the overhead of our implementation comprises three parts: the cost of computation on encoded data yielding into a branch, the costs of the branch protection scheme, and the costs of the CFI protection scheme. Given that we solely propose a branch protection, we do not focus on analyzing the cost of the used data protection or CFI protection scheme. In general, these costs are highly application specific and, therefore, hard to predict. Still, our evaluations indicate that the expected costs for enforcing CFI and for protecting data values

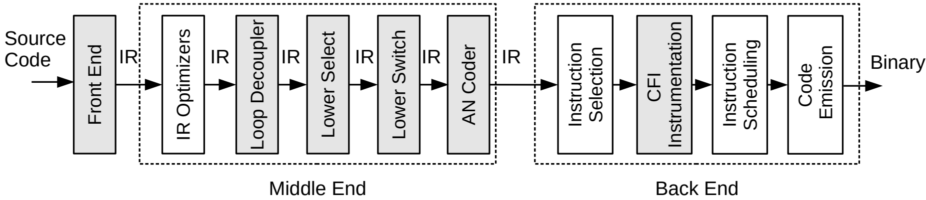


Figure 7.3: Modified LLVM compiler pipeline. Grey boxes indicate modifications or additions of/to the regular compilation flow.

are quite reasonable when mostly requiring simple arithmetic operations (e.g., loop counters or accumulator variables).

Analyzing the cost of the encoded compare, and the state update operations (Table 7.2) is possible precisely. The generic implementation[1] of the proposed AN-code-based encoded compare comprises additions, subtractions, and modulo operations. Every ISA typically supports addition and subtraction, but modulo is not necessarily supported directly and, therefore, often is more costly. With the used ARMv7-M ISA, modulo has to be implemented using a combination of a slow division (UDIV) and a multiply+subtract (MLS) instruction. As a result, depending on the comparison predicate, between 12 and 26 bytes of memory overhead and 6-33 cycles runtime overhead is generated for one encoded compare. Hardware support for a fast modulo instruction would considerably reduce this overhead.

The cost for state update operations is again highly dependent on the employed CFI protection scheme. In the software-centered design, they are implemented using one address load and a store of the comparison result to the CFI unit.

[1]Special encoding constants may have optimized implementations but different code properties.

Table 7.2: Qualitative overhead analysis of the building blocks.

Predicate	Required Operations	Our Prototype		
		Instructions	Size / B	Runtime / c[a]
$>$ $\geq$ $<$ $\leq$	1 + 1 - 1 %	1 ADD 1 SUB 1 UDIV 1 MLS	12	6-16
$=$ $\neq$	3 + 2 - 2 %	3 ADD 2 SUB 2 UDIV 2 MLS	26	13-33

[a]Division on ARMv7-M requires between 2 and 12 cycles.

These instructions are added to the beginning of the successor basic blocks of the protected conditional branch and introduce 4 bytes code and 4 cycles of runtime overhead per instantiation. An optimized CFI and branch protection design can fully omit these costs.

7.4.3 Performance Evaluation

We use two micro-benchmarks to measure the overhead in terms of runtime and code size. These benchmarks (*integer compare* and *memcmp*) test the branch protection in isolation by exercising a single integer equal comparison and a secure memory comparison with 128 elements. We compare this overhead with a duplication approach, where we duplicate the conditional branch six times consecutively to have a comparable single-bit fault tolerance to the AN-code-based implementation (*i.e.*, 6-bit Hamming distance for the encoded values). However, this duplication approach does not protect any data or arithmetic operation leading to the branch opposed to the AN-code-based scheme. As a macro-benchmark, we implement a fault-protected version secure bootloader, similar to the one in [Atm17]. Only programs which feature a valid ECDSA signature over the program's hash get executed. In this example, the memory comparison of the signature verification and all subsequent conditional branches are protected. This mitigates the single point of failure of a secure boot mechanism, which was already a target of fault attacks.

The costs, as shown in Table 7.3, also include the overhead of computing on the AN-encoded values. Based on the micro-benchmark results, we observe that the performance in terms of code size and runtime is on par with the duplication approach or even better. However, we do not only protect the conditional branch but also protect the data and the arithmetic operations on it. When applying this protection mechanism to the protected bootloader, the overhead is neglectable since the crypto implementation dominates code size and runtime. The code size overhead of less than 2.5 % and a neglectable runtime overhead makes this countermeasure applicable to real-world applications.

Table 7.3: Size and runtime overhead of different branch protections.

Benchmark	Metric	CFI	Duplication		Prototype	
		abs	abs	+ / %	abs	+ / %
integer compare	Size / B	12	128	967	86	617
	Runtime / c	20	91	355	63	215
memcmp	Size / B	68	272	300	276	306
	Runtime / c	1689	10210	504	8905	427
bootloader	Size / B	17252	—	—	17672	2.435
	Runtime / c	51888k	—	—	51888k	0.001

7.5 Security Analysis

To state the security of the countermeasure, we analyze its fault resistance. If there is a fault on a single location but with multiple bits flipped, the error is transparent and detectable, relying on the code properties of the selected A [FSS09]. For our parameter selection, we can detect up to 5-bit errors in a single word during the calculation. In the final condition result, the error detectability is even higher because only two symbols are valid. At this place, we reach a Hamming distance of 15-bit between the two condition values.

However, if errors are spread over multiple locations/operations, the fault detection capabilities of the AN-code decrease, and the code cannot detect as many bits as before. To investigate this behavior, we performed a simulation with faults at different locations. Simulations show that for our parameter selection, the error detectability is reduced to 3-bits, arbitrarily placed over the whole computation of the condition value. With four bits flipped over the whole computation of a condition value, the error rate where an attacker can flip the final condition value from true to false or vice versa is 0.0002 %, which increases having more bits flipped.

7.6 Compatibility

In this work, we a the software-centric GPSA-based countermeasure [WWM15] as the underlying CFI primitive. However, our approach is fully compatible with other CFI protection schemes. In general, the only requirement to implement secure conditional branches is a mechanism to inject data, *i.e.*, the protected condition value, ahead of executing the branch. This can either be done via a dedicated instruction or can be combined into a special conditional branch instruction.

7.6.1 Compatibility with FIPAC

Our method of protecting conditional branches is fully compatible with FIPAC, which is purely implemented in software, as presented in Chapter 5. In FIPAC, an `eor` instruction is used to perform a state update on the CFI state. This instruction XORs a value that is stored in a general-purpose register to the CFI state, which is also a general-purpose register. We can use this instruction to inject or XOR the secure condition value to the CFI state ahead of executing the ordinary and unprotected conditional branch.

7.6.2 Compatibility with SCFP

We also included the protection mechanism presented in this chapter to a hardware implementation of the SCFP [Wer+18] CFI protection scheme. SCFP, with our conditional branch protection, got integrated into a 4-stage open-source RI5CY/CV32E40P [Ope23a; PULb] core and taped out to the *Patronus* [Sch+16]

Algorithm 5: Pseudocode of the protected conditional branch instruction `bpdeq` for the RISC-V architecture.

```
CFIState ← CFIState ⊕ rs1
if rs1 = rs2 then
    CFIState ← CFIState ⊕ Patch
    PC ← PC + offset
else
    PC ← PC + 4
```

Application Specific Integrated Circuit (ASIC). For this purpose, we added a new conditional instruction named `bpdeq`, targeted for our countermeasure. The instruction, as detailed in Algorithm 5, performs in the first step a state update with the first register operand ahead of the execution of the conditional branch. This register holds the comparison result from the secure software-based comparison algorithm, as discussed before. Later on, it performs an equal comparison, performs the necessary state patch using the justifying signature, and updates the program counter. Similar to the prototype implementation of Section 7.4, the CFI state is only correct if the correct comparison result gets injected *and* the subsequent conditional branch is executed the right way.

7.7 Conclusion

In this chapter, we close the gap of unprotected conditional branches in CFI countermeasures in the presence of fault attacks. We eliminate the single point of failure by adding an encoded comparison operation that yields a redundant condition value. Using a standard compare and branch mechanism together with the ability to merge the redundant comparison result with the CFI protection mechanism allows us to protect the execution of a conditional branch. Our approach is highly flexible, allowing us to use different encoded comparison operations based on different encoding schemes with different security properties at different places in the program. We exploit the properties of arithmetic AN-codes and present novel comparison algorithms to compute the condition values arithmetically but preserve the redundancy. We integrated this countermeasure in the LLVM compiler to automatically protect conditional branches. Experimental evaluation shows little overhead to security-critical programs, such as the signature verification of a secure bootloader, making it applicable for real-world usage.

The modified compiler automatically transforms all branch-dependent data and its dependence chain of operations to the protected AN-code domain. However, in certain situations, such as when the program employs operations not supported in the AN-code domain, this transformation may not be fully achievable. Additionally, the program may utilize redundancy mechanisms that are incompatible with AN-codes. Despite these limitations, the proposed countermeasure could potentially e used solely to protect conditional branches. Right before

a conditional branch, both branch operands are encoded to the AN-code domain, which are then used to carry out the secure conditional branch as proposed in this chapter.

In summary, we believe this is a first step towards improved security for conditional branches in the presence of CFI protection schemes. As discussed at the beginning of this chapter, conditional branches are heavily used in security-critical operations, thus, their protection has importance. The proposed scheme is generic in the sense that it could be extended for data encoded with other encoding schemes but also in terms of compatibility with the CFI protection scheme.

8

Secure Memory Accesses in the Presence of Fault Attacks

In the previous chapters, we looked at the protection of the control-flow against fault attacks and designed software-based countermeasures using existing architectural features. But even with a protected control-flow, the attack surface is still large enough to compromise the system, e.g., via memory accesses. A memory access is a highly critical operation. Many decisions inside a program rely on the correct execution of a memory access. Password checks, signature verification, or grants to a privileged function, they all depend on the genuine execution of a memory access.

Under normal operating conditions, a memory access reads/writes from/to the desired location. However, the situation changes dramatically as soon as intentionally induced faults, *i.e.*, fault attacks, are considered. A fault on the processor's internal bus during a read or write operation either reads the wrong data from memory or writes it to the wrong location. Both cases are not trivially detectable with data redundancy, given that the data remains unmodified. Similar effects can be triggered by injecting faults into pointers, which are typically not prevented by these schemes. Since we are focusing on addressing errors, e.g., with faults on the pointer or on the memory bus, faults in the memory, e.g., via Rowhammer, are not the main concern of this work.

While there exist data encoding schemes that also aim to protect against addressing errors, *i.e.*, ANB-codes (cf. Section 2.5.2), they are very costly and impose severe restrictions on the protected code. ANB-codes introduce a tremendous runtime overhead of 90 % on average on top of already expensive AN-codes solely to solve the memory access problem. Furthermore, they can only protect a limited set of variables with well-known memory alignment and

size. More efficient and less restrictive approaches are needed to protect memory accesses against address tampering.

Contribution

This chapter addresses the issue of unprotected memory accesses in the context of fault attacks by proposing a hardware-software co-design. We introduce a practical solution to detect address tampering in pointers and on memory buses with an extended hardware design. Our generic approach works independently of the used code and data protection schemes and, therefore, can effectively be combined with state-of-the-art techniques in the context of hardening general-purpose computing against fault attacks.

First, we present a new approach to protect pointers against faults with negligible overhead in terms of runtime and storage requirements. We encode pointers using a multi-residue arithmetic code, which allows us to detect faults on encoded pointers during both storage and computation. The redundancy information of the code word is hereby stored in the unused upper bits of a pointer to fully utilize the available register space and yield zero overhead for storing an encoded pointer. This design decision is inspired by ARM Pointer Authentication (ARM PA), which uses a similar trade-off to embed a Message Authentication Code (MAC) into a pointer. Furthermore, by transforming the pointer arithmetic into the encoded multi-residue domain, the protection of the pointer is also maintained also when performing arithmetic operations on the pointer, e.g., when adding an offset to the stack pointer. We add new instructions to the processor that efficiently operate on encoded pointers.

Second, we propose an efficient way to protect memory accesses from tampering by linking the stored data in memory with the address of the access. We establish this link whenever data is written to the memory and remove the link as soon as the data is read back into the processor. When considering fault attacks, countermeasures like data encoding are already necessarily employed. By linking the redundant address information with the encoded data, faults during addressing manifest as errors in the redundantly encoded data, where they can be detected. As a result, data integrity checks implicitly also check for address tampering and make explicit addressing error checks unnecessary.

Finally, to evaluate the concept, we integrated our protection mechanism into a Field Programmable Gate Array (FPGA) hardware implementation of an open-source RISC-V processor. Furthermore, to avoid the tedious manual encoding of all pointers and addresses inside the program, we integrated this concept directly into an LLVM-based C compiler, which is capable of automatically protecting complex codebases without manual interference. The resulting prototype induces 10 % code size and less than 7 % runtime overhead on average. Summarized, our contributions are:

- To protect the memory subsystem, we encode all pointers to the multi-residue domain and perform protected pointer arithmetic with zero overhead in terms of storage costs.

- To secure the memory access itself, we link the pointer's redundancy through the memory bus down to the memory cell.
- We provide an LLVM-based toolchain to automatically encode all pointers and pointer arithmetic to the residue domain. All memory accesses are replaced with its linked counterpart, which uses an encoded pointer.
- To show the feasibility of the design, we enhance the hardware design of a RISC-V core with new instructions and evaluate the concept on different benchmarks.

Scientific Contribution

Chapter 8 is primarily based on the following publication that was presented at ACSAC 2018 in San Juan (Puerto Rico).

Robert Schilling, Mario Werner, Pascal Nasahl, and Stefan Mangard. "Pointing in the Right Direction - Securing Memory Accesses in a Faulty World." In: *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, pp. 595–604. DOI: 10.1145/3274694.3274728

I am the main author of this paper, developed the technical idea and the prototype toolchain, and performed all experiments and evaluations. Mario Werner contributed to the text of this paper. Pascal Nasahl contributed to the prototype RISC-V hardware implementation. Stefan Mangard supported this work in many discussions and giving feedback to the paper.

Outline

The remainder of this chapter is structured as follows. Section 8.1 discusses related work on memory access protection or pointer protection. Section 8.2 presents the thread model and attack scenario. In Section 8.3, we describe how we protect pointers against fault attacks. The approach to link the pointer protection with data encoding is presented in Section 8.4. Section 8.5 details how we extend the RISC-V instruction set to support encoded pointers and discusses our compiler modifications. Finally, Section 8.6 evaluates the overhead, and Section 8.7 concludes this chapter.

8.1 Background of Memory Access

In this section, we describe the related concepts, which aim to secure pointers or a memory access in general.

8.1.1 ANB-Codes for Memory Access Protection

There already exist mechanisms to protect the memory access against tampering. One method is the use of ANB-codes, as detailed in Section 2.5.2. By assigning/adding a variable-dependent signature B_x, each code word gets a unique checking property, which is used to detect wrong memory accesses.

After loading a value from the memory back to the register, the variable is identified to ensure the memory access was not tampered with. Unfortunately, this encoding scheme has significant challenges to be used in practice. As already discussed, the ANB-codes add significant overhead – around 90 % – on top of ordinary AN-codes- Furthermore, a compiler always needs to track which data is used where to correctly identify wrong memory accesses. In practice, this is not possible for arbitrary programs. Finally, AN- and ANB-codes limit the range of values significantly, thus only providing protection to a limited set of computations.

8.1.2 ARM Pointer Authentication

Protecting pointers against tampering is not only relevant in the context of fault attacks but is also used to counteract software attacks. As discussed in Section 5.1, ARM added a Pointer Authentication [Qua17], to cryptographically sign and authenticate pointers.

Even though the general approach is similar to our work, the provided capabilities and the resulting protection is vastly different. ARM Pointer Authentication aims to only protect special pointers against software attacks. In Pointer Authentication Code (PAC), authenticated pointers cannot be protected during pointer arithmetic since there is no homomorphism for the MAC. Furthermore, ARM PA only aims to protect the pointer. The memory access, which uses an authenticated pointer, is completely unprotected, and there are no protection mechanisms to ensure that the accessed memory actually originates from the correct address.

8.2 Threat Model and Attack Scenario

This section presents the threat model we consider and shows how fault attacks can hijack a memory access in different ways.

8.2.1 Threat Model

In this work, we shift the threat model from the control-flow part of the system to the memory subsystem of the processor and System-on-Chip (SoC). We assume a powerful attacker which performs fault attacks in order to compromise a system by redirecting memory accesses. Faults can be induced into instructions and data at various places, for example, in registers, during computation in the Arithmetic Logic Unit (ALU), on buses, and in memory. Many of these attack vectors can be covered by existing and established countermeasures, which we assume to be

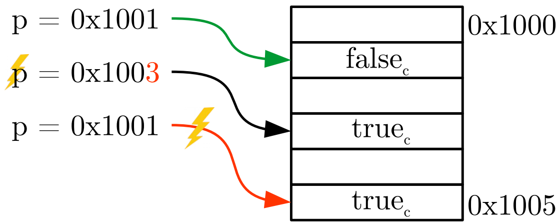


Figure 8.1: Attack vector: Modified pointers and manipulated memory accesses.

in place. Control-Flow Integrity (CFI) protections schemes against fault attacks such as FIPAC from Chapter 5 with the conditional protection from Chapter 7 or related work [Cle+17; Wer+18] can enforce the authenticity of instructions as well as their execution sequence, thus, can be used to protect code against faults. Furthermore, such a CFI protection scheme already protects function pointers, which do not require further protection. Data, on the other hand, can be protected during computation and storage using data encoding techniques like, for example, arithmetic codes like residue or ANBD-codes. These redundancy schemes also cover fault attacks on the memory, e.g., via Rowhammer. Details of various of those countermeasures are discussed in Chapter 2.

However, as soon as data is transferred via a memory bus, these encoding schemes are insufficient. While the value itself is protected via the code, the corresponding address information remains vulnerable to fault attacks. Furthermore, pointers typically remain unprotected by the data encoding schemes considering that eventually, the plain value of the pointer is used to address the memory.

8.2.2 Attack Scenario

To illustrate the problem, Figure 8.1 visualizes a simple memory access. On the left side, there is the pointer used for a memory access. On the right side, there is the memory, and the arrow in between denotes the memory access. The data in the memory is redundantly encoded, denoted by the c-subscript of the variables. Originally, the pointer p points to the address 0x1001 to read out the value false_c from the memory. However, a fault can manipulate the memory access to read out a wrong value. In particular, there are two error sources, which can lead to a wrong memory access. First, the attacker can modify the pointer, as shown in the middle example in Figure 8.1. If a pointer gets modified, then all subsequent memory accesses lead to a wrong location. An attacker could, e.g., modify two pointers used for a signature comparison to point to the same location, which always bypasses the memory comparison. This can occur anywhere in the program, also during pointer arithmetic. The second source of a manipulated memory access is the memory operation itself. When assuming the pointer is correct and not manipulated, the memory access can still be manipulated. A fault on the address bus can redirect the memory access to a wrong location, as indicated in the third example.

Both of these attack vectors can lead to a wrong memory access. Today, there is no efficient way to protect them, leaving frequently used memory operations completely unprotected against fault attacks.

8.3 Pointer Protection with Residue Codes

The manipulation of a memory access is possible by attacking two different parts of the access. The first one is the pointer itself, which is used to perform the memory access. This section details how we use multi-residue codes to protect every data pointer inside a program against fault attacks. Furthermore, we present how to integrate the multi-residue code into our pointer representation and elaborate on the additionally needed hardware support.

8.3.1 Overview

Pointers are ubiquitous. Every memory access, e.g., accessing a variable on the stack, uses a pointer to address the memory. However, when considering fault attacks, pointers may be manipulated to point to a different memory location.

To counteract this threat, we encode all pointers to a redundant representation where faults are detectable. As presented in Section 2.5.2, there are two classes of suitable codes: *systematic* and *non-systematic* codes, which can have similar properties in terms of error detection capabilities and support for computation. However, a systematic code has advantages of protecting a pointer. Namely, it supports direct access to the functional value and can, therefore, immediately be used to address memory. On the other hand, using a non-systematic code to protect the pointer requires performing a potentially expensive decoding operation before the actual address is available. AN-codes, as an example for non-systematic codes, require a costly integer division during the decoding operation. Hence, this division would be required for every memory access.

We encode pointers using a *systematic* multi-residue code with a scalable number of moduli. Details about the properties of multi-residue codes are discussed in Section 2.5.2. Here, an encoded pointer p_c is denoted as a tuple (p, r_p), where p is the original value of the pointer, and r_p denotes the redundancy part comprising the residues of p given a moduli set M. Using a multi-residue code to protect the pointer gives two advantages. On the one hand, the strength of the code, *i.e.*, the number of detectable bitflips, is scalable with the number of residues. On the other hand, residue codes are arithmetic codes and, therefore, also support arithmetic instructions, like addition and subtraction, natively. This allows us to perform pointer arithmetic, for example, the stack pointer manipulation in function prologues and epilogues, directly inside the encoded domain without decoding the pointer.

8.3.2 Pointer Layout and Residue-Code Selection

Adding separable redundancy to data implies that the additional information needs to be stored somewhere in order to provide a value. In the context of protecting a processor register, various possibilities exist to provide this storage.

For example, an additional parallel register file can be added to the processor, which only holds the redundancy part and gets updated in lockstep with the actual values [MM11]. However, this approach is quite costly for our use case, considering that only a small number of registers typically hold pointers at a certain point in time. Alternatively, pairs of regular registers can be used to store the data and its redundancy. Unfortunately, doing so increases the register pressure and lowers the overall performance. Moreover, without adding costly access ports to the register file, multiple instructions have to be performed on every pointer operation, even for simple ones like an increment. Finally, at least for modern RISC Instruction Set Architectures (ISAs) [Wat+14], adding additional operands into the instruction encoding is difficult without increasing the instruction size and adding new read ports to the register file.

In this work, we followed a different approach and store the redundancy information directly into the upper bits of the pointer. Similar to PAC, *i.e.*, ARM's Pointer Authentication feature, this approach introduces zero overhead in terms of storage for the redundancy at the cost of some bits of address space. Additionally, this dense representation of an encoded pointer allows us to add new combined residue arithmetic instructions, which operate on the functional value and on the residues in parallel, rather than requiring separate instructions to handle both. By storing the redundant pointer in one register, we can, therefore, use the same instruction format as regular instructions and do not require extensive modifications of the ISA or hardware to maintain performance.

Considering that the directly accessible address space is limited, embedding the residues into the pointer works best for modern 64-bit architectures. Therefore, the following design considerations, as well as our prototype, which is presented in Section 8.5, is built upon such an architecture. The overall concept can still be applied to 32-bit architectures with reduced error detection capabilities or via a different storage option.

Parameter Selection

When selecting the parameters of an error detecting code, it is always a trade-off between error detection capabilities and the overhead introduced by the code. However, since the functional value, including the redundancy, is stored in a single register, also the remaining address space has to be considered. For our prototype, we focus on a 64-bit architecture and partition our pointers into 24-bit redundancy and a 40-bit functional value. The resulting pointers can still address one terabyte of memory, which is sufficient for most applications. This partitioning is in line with related work, *i.e.*, ARM PA, which is described in detail in Section 5.1.

As a concrete code, we instantiate a multi-residue code with the moduli set

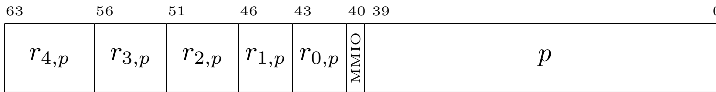


Figure 8.2: Encoded pointer representation. The actual 40-bit pointer value p, the MMIO tag bit, and 23 bits of redundancy r_p comprise an encoded 64-bit pointer.

$M = \{5, 7, 17, 31, 127\}$, which is an extension to the one presented in [MM11]. This moduli set yields a code with a Hamming distance of $D = 5$ and is capable of detecting up to four bitflips in the encoded 64-bit pointer value. Storing the residues for these moduli requires a total of 23 bits, *i.e.*, $3, 3, 5, 5, 7$ bits, respectively. The last remaining bit is used as a tag bit and specifies if data accessed via the pointer have to perform data linking/unlinking, as presented later in Section 8.4.3. The resulting register layout of such an encoded pointer is shown in Figure 8.2.

8.3.3 Pointer Operations

Pointers are used not only to perform a memory access but also to perform pointer arithmetic. To maintain good performance, it is therefore vital that the encoded pointers support these computations as efficiently as possible. Notably, as the term pointer arithmetic already hints, arithmetic computations, like addition and subtraction, are the most common operations that are performed on pointers. For example, accessing larger sequential memory chunks via a pointer involves a large number of additions between the pointer and the access stride in a loop. Similarly, next to every function call, the respective stack frame size is added and subtracted to/from the stack pointer in the function's prologue and epilogue. Precisely these types of operations are natively supported by the used multi-residue code and can therefore be performed in the encoded domain.

On the other hand, more work is required for operations that are not directly supported by the multi-residue code. The simplest approach is probably to perform the operation on the plain functional value only and restore the encoding afterward. To ensure the correctness of the computation, then additional countermeasures like replication have to be used. Alternatively, such operations can be performed by first converting the pointer to a different code, in which the computations are straightforward, followed by converting the differently encoded result back into multi-residue representation. Still, such operations comprise only a very small number of pointer operations compared to arithmetic operations.

Software vs. Hardware

In a multi-residue code, the addition operation is performed on the functional value and on all its residues. This operation can be executed in hardware or software. However, performing this operation in software is challenging, as it involves a modulo reduction for each residue.

Looking only at a single modulo operation, there exist several options for implementing the reduction in software: First, a normal modulo instruction from the ISA can be used. Although such an instruction does not have much code overhead, a modulo operation involves a costly integer division which usually takes multiple clock cycles to finish. Second, instead of a modulo operation, a conditional subtraction can be used for the modular reduction. Third, there are optimized modulo algorithms available [Jon], but their overhead is still large. A single modular addition with an optimized reduction with the modulus five takes at least 18 instructions on our RISC-V target architecture.

Considering that the runtime of these solutions additionally has to be multiplied with the number of used residues makes a software solution even less attractive. Furthermore, even if the performance penalty is acceptable, additional registers have to be reserved for implementing the reduction functionality. Summarizing, a software-based approach to perform residue operations, while feasible, is not very practical. Therefore, hardware-based approaches to implementing the residue operations have been investigated.

In particular, in our prototype, we add new instructions that permit to perform addition and subtraction of multi-residue encoded pointers. Section 8.5.1 discusses the new instructions in detail, focusing on the target architecture. Furthermore, an instruction for performing the expensive encoding operation is added, which computes the modulus for each residue. For convenience reasons, also a dedicated decoding operation is added to the ISA.

8.4 Evolved Memory Access Protection

Apart from faulting the pointer, the second source to manipulate a memory access is the memory operation itself. If the attacker is able to induce faults on the address bus, the memory access can be redirected to a different location. In this section, we present a method to link the data with its respective address, where addressing errors are transformed into data errors which can subsequently be detected using a data-protection scheme.

8.4.1 Overview

In order to be able to detect address tampering, a way to uniquely identify incorrectly accessed memory is needed. A common approach to establish this link between the data and the address is augmenting the data-protection scheme, which is anyway needed to protect the data against faults.

For example, ANB-codes embed the identity of the variable, in the form of a unique residue B_x, into a required underlying AN-code-based data encoding. However, this approach has several drawbacks. For example, working on variable granularity requires concise data-flow information, which is, in real-world applications, hard to acquire for arbitrary memory operations, and limits the applicability of the approach. Furthermore, maintaining these identities during

calculation is quite costly. Finally, the approach is strongly linked with AN-codes and cannot easily be applied to other data-protection schemes.

Our scheme takes an entirely different approach to prevent address tampering. Instead of constructively embedding the address of the data into the data-protection code, our scheme destructively overlays data that is written to the memory with the respective memory address. As a result, addressing errors are transformed into data errors that get detectable as soon as the overlay is removed again.

In more detail, before data is written from a register to the memory bus by the processor, the data gets encoded with respect to the target address. Conceptually, this kind of linking is similar to encrypting the data in an address-dependent way. However, since we do not strive for confidentiality with our approach, the use of a cryptographically secure cipher is not needed. The resulting encoded data is then stored simply into memory like in a regular system.

When data is read back from memory into a processor register, the decoding with respect to the target address is performed. Considering that the performed decoding operation is the inverse of the encoding, a genuine data value is restored only when the read has been performed from the correct address. Otherwise, an incorrect data value is generated, which can be detected via the used data-protection scheme. Note that the detection of address tampering during memory writes is possible like this as well. However, the detection is delayed to the point where the incorrectly written value is read back into the processor.

8.4.2 The Linking Approach

As already mentioned, the general idea behind our memory access protection approach is to link the data that is stored in memory with its respective address. Instead of directly writing a register value D_{Reg} into memory at a certain address p (*i.e.*, $\texttt{mem}\left[p\right] = D_{Reg}$), a little more work has to be performed in our scheme. Namely, as shown in Equation (8.1), the linking function l has to be evaluated in order to determine the value that is actually written to the memory at address p.

$$\texttt{mem}\left[p\right] = l\left(p, D_{Reg}\right) = l_p\left(D_{Reg}\right) \tag{8.1}$$

The purpose of this linking function is to combine the address p with the data value D_{Reg}. However, not every function can be used for this purpose. At the very least, the following two requirements have to be fulfilled in this context. First, for each address p, the linking function l_p has to be a permutation. Having this property means that l_p performs a bijective mapping and that an inverse function l_p^{-1} exists, as shown in Equation (8.2).

$$\forall p, D_{Reg} \rightarrow l_p^{-1}\left(l_p\left(D_{Reg}\right)\right) = D_{Reg} \tag{8.2}$$

Subsequently, memory read operations can be implemented using this inverse function, as shown in Equation (8.3). As a result, from the software perspective, encoding data when storing to memory and decoding data when loading

from memory is completely transparent, yields the expected result, and can be performed for every memory access.

$$D_{Reg} = l^{-1}\left(p, \mathtt{mem}\left[p\right]\right) = l_p^{-1}\left(\mathtt{mem}\left[p\right]\right) \tag{8.3}$$

Second, to ensure that addressing faults are detectable, data encoded under one address should yield a modified value when being decoded under a different address, as shown in Equation (8.4). Furthermore, the modified value should not be a valid code word in terms of the used data-protection code.

$$\forall p, p', D_{Reg} : p \neq p' \rightarrow l_{p'}^{-1}\left(l_p\left(D_{Reg}\right)\right) \neq D_{Reg} \tag{8.4}$$

Function Selection

Various functions, such as cryptographic ciphers, fulfill these requirements and are therefore suitable to link the data and the address information as required by the memory access protection scheme. However, given that we aim for a low-overhead design, less resource-demanding functions have been investigated.

Interestingly, already a simply XOR operation, as shown in Equation (8.5) and Equation (8.6), is sufficient as the linking function for our use case. In more detail, in our scheme, addresses are encoded using arithmetic multi-residue codes, and the data encoding can be selected arbitrarily. On the one hand, when the same multi-residue code is also used for the data, e.g., an encoded pointer is written to memory, using the XOR operation is a good choice, given that multi-residue codes are not closed under the XOR operation. Subsequently, it is also unlikely that combining multiple valid code words yields a valid result and therefore facilitates error detection. On the other hand, even when a data protection code that is closed under the XOR operation is used, still similar error detection capabilities are expected. After all, combining code words from different codes is highly unlikely to yield sensible results.

$$\mathtt{mem}\left[p\right] = p \oplus D_{Reg} \tag{8.5}$$

$$D_{Reg} = p \oplus \mathtt{mem}\left[p\right] \tag{8.6}$$

Linking Granularity

Theoretically, the previously described linking approach can be applied with arbitrary granularity. Therefore, applying the technique to the processor's native word size, e.g., 64-bit in our prototype, may appear natural. However, performing XOR-based linking on such a coarse granularity does not yield the desired amount of diffusion. Namely, bytes that are close to each other, *i.e.*, with a stride of 8 bytes when operating on 64-bit, are highly likely to have the same address pad. Furthermore, in many real-world applications, also misaligned data accesses with arbitrary sizes have to be supported efficiently. Situations like this, for example, commonly arise when arbitrarily aligned data is copied via the *memcpy* function.

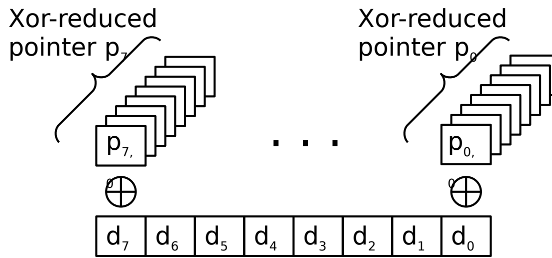


Figure 8.3: Byte-wise data linking of a 64-bit word. Each byte gets XORed with its respective XOR-reduced encoded address.

Therefore, to fix the problem of the low diffusion and the arbitrarily aligned data accesses, we perform the linking of data and address with byte-wise granularity. Each byte, even when it is part of a larger memory transfer, is independently linked with its respective address. Hereby, each individual byte-address pointer is still multi-residue encoded to provide the desired diffusion during linking. Furthermore, the actual linking is again performed via an XOR similar to Equation (8.5). However, considering that the data and its address have different bit sizes, an additional compression is applied on the address before linking. Namely, each 64-bit address $p = [p_0, p_1, \ldots, p_7]$ gets reduced to one byte value p' by XORing the individual address bytes as shown in Equation (8.7).

$$p' = \bigoplus_{i=0}^{7} p_i \tag{8.7}$$

Applying this approach to a full 64-bit word is visualized in Figure 8.3. Considering the number of needed multi-residue operations for such a word-sized access, using this linking scheme effectively requires hardware support. In this work, we integrated the needed transformations directly into special load and store instructions. From the software perspective, encoding data when storing to memory and decoding data when loading from memory is completely transparent and can be performed for next to every memory access.

8.4.3 Memory-Mapped I/O

Memory-Mapped I/O (MMIO) is a common communication interface in embedded processors to access peripherals. In MMIO, the peripheral registers are mapped into the standard memory layout of the processor. This allows the processor to use ordinary load and store instructions to access the peripheral.

However, in order to protect the memory access, our architecture uses redundant pointers and links them with the data before executing the memory access. Since a standard memory-mapped peripheral is not aware of this data linking, wrong data would be written to the device. Therefore, we cannot apply data

linking when accessing a memory-mapped peripheral. However, we still can use an encoded pointer to access the memory-mapped peripheral, as this does not influence the data. In order to use an encoded pointer but not perform the data linking, we would need special instructions for load and store for this purpose. We avoid this overhead by encoding this information directly into the encoded pointer. The load and store instructions detect this and do not perform the data linking accordingly.

As shown in Section 8.3, we redundantly encode the pointer using a multi-residue code. In Figure 8.2, we show the pointer layout where the 41st MMIO-bit indicates whether the pointer is for an MMIO access without data linking. The residues, which form the redundancy of the pointer, are computed over the 40-bit functional pointer value and the MMIO-bit to protect both against tampering.

8.5 Architecture

The concept of protected pointers and linked memory accesses is integrated into a prototype implementation based on a 64-bit RISC-V architecture. In this section, we first discuss the new instructions, show how we integrated them into the architecture, and finally show a compiler prototype to automatically protect all memory accesses in a program.

8.5.1 New Instructions

As previously described, it requires hardware support to efficiently perform the residue arithmetic such that the performance penalty is acceptable. In this work, we extend the instruction set of the processor with instructions that operate in the encoded residue domain. In particular, the following custom instructions are added to the instruction set.

***renc*, *rdec*.** To efficiently encode a value into the multi-residue domain, a dedicated encoding instruction (*renc*) is added. The encoding operation computes the residues over the 41-bit functional value of the pointer, which also includes the *MMIO*-bit in the protection domain. As a second instruction, we add support to decode a multi-residue encoded register (*rdec*). Both instructions are idempotent, meaning they can repeatedly be executed (encoding an already encoded value does not change the encoding).

***radd*, *raddi*, *rsub*.** To support pointer arithmetic on encoded pointers, hardware support for the most commonly used operations is added. Concretely, we support adding two multi-residue encoded register values (*radd*), adding a multi-residue encoded value to an immediate value (*raddi*), and subtracting multi-residue encoded values (*rsub*). The immediate value in the *raddi* instruction is not yet multi-residue encoded. However, these values are part of the instruction encoding and are already protected via the CFI code protection scheme. Note

that before the immediate can be used in a residue operation, it gets encoded as part of the instruction execution.

***rlxck*, *rsxck*.** Since we now use encoded pointers and require data linking/unlinking, dedicated memory instructions are added to the ISA. Therefore, a family of new load (*rlxck*) and store (*rsxck*) instructions is added. Herby, the x denotes the access granularity of the memory operation. Concretely, we support byte (b), half-word (h), word (w), and double-word (d) accesses with and without sign extension, which corresponds to the original memory access instructions in the RISC-V 64-bit ISA. The new instructions have the same operand interface as the original load and store instructions of RISC-V. However, they now take an encoded pointer for addressing the memory. The memory instructions also contain a plain immediate value to add an offset to the pointer, which is protected by the CFI code protection. Furthermore, these instructions perform the data linking and unlinking on a byte-wise granularity. However, if the 40th-bit, the *MMIO* bit, is set to one, no data linking and unlinking is performed. This approach allows us to use a protected pointer when accessing a memory location that does not support data linking, e.g., a memory-mapped peripheral.

Since every memory access is replaced with its protected counterpart, the protection mechanism could already be implemented in the original load and store instructions of the processor. However, for the sake of still supporting the original RISC-V instructions, they are left unmodified, and new instructions are added separately.

8.5.2 Hardware

The instruction set is only one part of our protected architecture. We also implemented the modified instruction set in hardware. As a foundation, we use the open-source 32-bit RISC-V core *RI5CY* [PULb], which was meanwhile renamed to CV32E40P [Ope23a]. This core is extended to a 64-bit processor meaning that the register file, datapath, and load-and-store unit are modified, and all necessary instructions are added to be compliant with the RISC-V RV64IM instruction set. Furthermore, we added new instructions to deal with multi-residue encoded pointers, as defined in Section 8.5.1. Figure 8.4 shows the modified processor pipeline, which includes a dedicated ALU for residue operations. Furthermore, immediate values, which are part of the instruction, get encoded during the instruction decode stage of the processor pipeline. The load-and-store unit is extended to support data linking and unlinking to protect all memory accesses.

The new residue ALU is shown in detail in Figure 8.5. The ALU supports encoding and decoding of values to/from the multi-residue domain as well as adding and subtracting two encoded values. The design of the ALU is optimized to require only one residue adder and one encoder in the execution stage of the processor. Decoding is free since it only requires rewiring, where the upper bits are set to zero. After performing an addition, the functional value of the adder

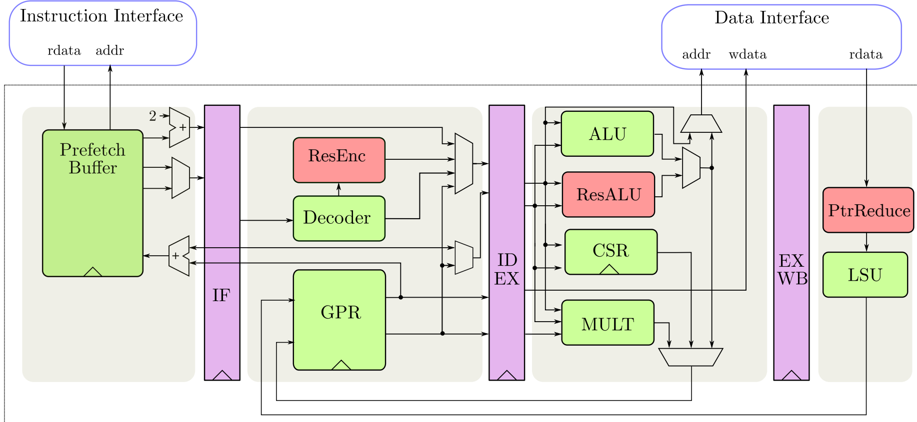


Figure 8.4: Modified processor pipeline. The instruction decode stage is extended with a 12-bit residue encoder, the execution stage with a residue ALU, and the write-back stage with a pointer-reduction data-linking unit.

result is re-encoded and compared with the independently computed residues in order to perform error-checking after each residue instruction. If the computed residues and the newly re-encoded residues mismatch, a redundant error signal is generated to force the processor into a safe state. Since this adder is also used for computing the final pointer address during a memory access (the encoded immediate value is added to the encoded base pointer), every pointer is also checked before performing a memory access. With frequent checks for every result, we minimize the probability that error masking occurs and errors are not detectable anymore.

Currently, the residue encoder uses special algorithms from [PB09] to encode data. However, the residue adder is implemented without any further optimizations. By using optimized arithmetic operations, e.g., the one from [Zim], the hardware overhead can be further reduced.

8.5.3 Software

To make the countermeasure practical and protect every memory access in the program, the new instructions and the protection mechanism also need to be integrated into the compiler. In the following, we integrate our countermeasure into the LLVM-based C compiler [LA04].

An LLVM-based compiler is partitioned into three parts, the front end, the middle end, and the back end. While the middle end optimizes target-independently on an intermediate code representation, the back end transforms the universal intermediate representation to a target-dependent code. To protect every memory access in the program, the countermeasure needs to be inserted in the back end stage of the compiler. Any earlier transformation can potentially miss a memory access leaving some accesses possibly unprotected (e.g., the stack

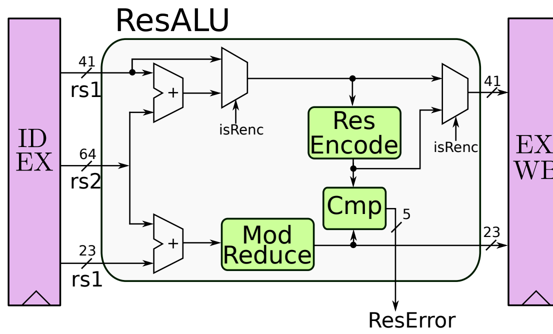


Figure 8.5: Residue ALU with a 41-bit adder and a shared residue encoder. The addition result is automatically checked after the operation by re-encoding the result and comparing it with the computed residues and generating a redundant error signal.

is created in the target-dependent part of the back end). Even in the back end, the protection happens right before the final instruction scheduling.

LLVM's back end uses a Directed Acyclic Graph (DAG) representation, the *Selection DAG*, for the code generation. The intermediate representation is transformed in a series of steps to finally emit the machine code. However, the back end has no information about pointers and addresses. Therefore, this information is created and propagated manually on the Selection DAG. Dedicated pointer nodes are added to the Selection DAG where pointers are created, e.g., when creating a *FrameIndex* node used for a local stack memory access. This information is then propagated on the Selection DAG, and all dependent operations are replaced with their corresponding residue counterpart. If we obtain an instruction, which is not supported by the residue code, the pointer is decoded, the operation is performed in the unencoded domain, and then, the pointer is re-encoded. However, this sequence of instructions is not used in the majority of the transformations. Finally, protected load and store instructions are emitted, which use an encoded pointer for addressing the memory.

If the program uses a constant address, e.g., the address of a global variable, this information needs to be encoded to the multi-residue domain. However, the compiler does not have this information yet. Therefore, it creates a relocation such that the linker can fill in the correct address information. Since this information requires multi-residue encoding, the linker is also modified. In our work, we use a custom RISC-V back end of LLVM's *lld* linker. In addition to resolving regular relocations, our linker also applies multi-residual encoding to pointers in the binary. This includes pointers synthesized in code as well as pointers stored in the memory, which additionally get linked with address information. Similar to that, data stored in the read-only section of the binary is also linked with its address. As soon as these values are loaded into a register, the unlinking operation is performed, and the correct value is restored.

8.6 Evaluation

In order to make a countermeasure usable in practice, the overhead must be reasonable. In this section, we first show the introduced hardware overhead and then evaluate different benchmark applications on the target architecture. Finally, we analyze the software overhead, discuss the overhead sources, and describe future optimization possibilities.

To quantify the hardware overhead, we synthesize the hardware architecture for a Xilinx Artix-7 series FPGA. By adding the new instructions, a dedicated ALU for multi-residue operations, and a modified load-and-store unit, the required number of Lookup Tables (LUTs) increases by less than 5 %, and the number of flip-flops increases by less than 1 %. However, this prototype design is implemented without optimizations leaving space to improve the design further.

The custom LLVM toolchain based on LLVM 6.0 is used to compile different benchmark applications for the RISC-V-based target architecture. The benchmarks were taken from the *PULPino* repository [PULa], which were used to originally evaluate the performance of the RI5CY core. Simulation is performed using a cycle-accurate Hardware Description Language (HDL) simulation of the target processor. As a baseline, we simulate the benchmark applications solely with enabled CFI protection [Wer+18] but without an application-specific data protection scheme. On top of that baseline, we determine the exclusive overhead of our countermeasure in terms of code size and runtime.

As shown in Table 8.1, on overage, the code overhead is 10 %, and the runtime overhead is less than 7 %. This is a comparable better performance to ANB-codes, which have an average runtime overhead of 90 % compared to AN-codes solely to provide memory access protection. Instead, our countermeasure has considerably lower overheads, making it attractive for many real-world applications.

Table 8.1: Code and runtime overhead for different benchmark programs from an HDL simulation.

Benchmark	Code Overhead		Runtime Overhead	
	Baseline [kB]	Overhead [%]	Baseline [kCycles]	Overhead [%]
fir	4.26	8.54	39.22	6.35
fft	6.52	6.57	58.01	4.65
keccak	4.79	10.11	255.55	11.31
ipm	4.84	12.81	10.80	3.94
aes_cbc	7.25	8.77	60.91	9.10
conv2d	3.26	13.12	5.92	2.70
Average		9.99		6.34

8.6.1 Future Work

The overhead numbers are already competitive for practical usage, even for a prototype implementation. Still, some improvements regarding code size or performance have not been performed yet.

For example, pointer comparisons in the encoded domain are currently only implemented for *equal* and *not equal*. Although seldomly used, comparisons with other predicates are still performed on the functional value without the protection of the residue code. Similarly, there are rare cases when pointer arithmetic uses unsupported logical operations. In this case, the operations is performed only on the functional value. Adding support for these operations would further increase the protection domain.

Furthermore, our current toolchain has not been highly optimized for our prototype architecture yet. We expect that, with a more optimized compiler, even better results can be achieved in the future.

8.7 Conclusion

Memory accesses are frequently used operations, and many different security policies, as well as safety mechanisms, rely on their correct execution. However, when dealing with faults, a correctness of a memory access cannot be guaranteed. While there are dedicated methods to protect the control-flow of a program and to protect the data in memory and registers, there is no efficient protection mechanism to protect the memory access against address tampering.

In this chapter, we closed this gap and presented a new mechanism to protect memory accesses inside a program. The countermeasure is employed in two steps. First, all pointers, including pointer arithmetic, are protected by employing a multi-residue code. The redundancy is hereby directly stored inside the unused upper bits of the pointer, which does not add any memory overhead. The second step links the redundant pointer with the data. Subsequently, addressing errors manifest as data errors and get detectable as soon the data is loaded into the register. This linking approach is universally applicable and can be used on top of any data protection scheme.

To demonstrate the practicability of our countermeasure, we integrated the concept of protected memory accesses into a RISC-V processor. We extended the instruction set to deal with multi-residue encoded pointers and added new memory operations which perform the linking and unlinking step. Furthermore, we extended the LLVM compiler to automatically transform all pointers of a program to the encoded domain. Our evaluation showed an average code overhead of 10 % and an average runtime overhead of less than 7 %, which makes this countermeasure practical for real-life applications.

Our design provides a first step towards the efficient protection of memory accesses against fault attacks. By using a hardware-software co-design, we are able to efficiently protect pointer arithmetic and direct memory accesses against faults. This work targets embedded use cases where the processor does not

contain a Memory-Management Unit (MMU) and only performs direct memory accesses. However, since the complexity of devices is increasing and nowadays, systems are often based on application-class processors, there is a gap. Memory accesses coming from the virtual domain and supporting shared memory accesses are not yet supported. In Chapter 9, we extend the scope of protection and solve these challenges.

9 Protected Memory Accesses in the Virtual Memory Domain

In Chapter 8, we discussed a protection scheme to arbitrarily protect memory accesses of bare-metal applications against fault attacks. However, Internet-of-Things (IoT) nodes emerged in the last decade from small microcontrollers to more powerful application-class processors running commodity Operating Systems (OSs). These systems use a Memory-Management Unit (MMU) to implement virtual memory and efficiently isolate the memory of different programs. In virtual memory architectures, the concept of *shared memory* is used to provide inter-process communication via the memory subsystem. However, shared memory limits the use of the protection scheme from Chapter 8 for application-class processors, as it is designed for bare-metal applications only. Unfortunately, even on larger application-class or commodity desktop processors, fault attacks on the virtual memory systems are exploited to gain kernel privileges [Goo15; Tro+21]. Consequently, new and efficient mechanisms are required to protect larger applications with arbitrary memory accesses from the virtual memory domain.

Contribution

In this chapter, we present SecWalk, an efficient countermeasure to protect virtual memory accesses against fault attacks. Our approach allows us to protect all memory accesses of a program against fault attacks, even for large application-class processors. This work closes an open gap and protects the page table walk against fault attacks by linking the redundancy properties of virtual addresses to physical addresses. We built upon the protection scheme of Chapter 8, where we

encode pointers and addresses in the virtual memory domain using a multi-residue code. The proposed secure page table walk uses the redundancy of a pointer to securely translate the virtual address to an encoded and protected physical address. We exploit the arithmetic properties of encoded pointers to retrieve the correct page table entries during the address translation. The translated encoded physical address is then used to perform the actual secure memory access. This mechanism allows us to support all common memory allocations, such as dynamic or shared memory.

To evaluate the design, we integrate SecWalk into a hardware implementation of an open-source RISC-V processor. The evaluation of our prototype implementation shows that the hardware overhead increases the size of the processor design by less than 0.5 % in terms of flip-flops and 10 % in terms of lookup tables. To evaluate the software overhead, we develop a custom LLVM-based toolchain to automatically instrument programs with SecWalk. On a set of microbenchmarks, SecWalk yields an average code overhead of 11.05 % and an average runtime overhead of 7.17 %. To showcase the applicability to larger programs, we integrate SecWalk to the commodity microkernel seL4 and automatically protect all memory accesses of the kernel and user threads (virtual and physical accesses) against fault attacks. Instrumenting all pointer arithmetic and every memory access increases the code size by 13.1 % and the runtime overhead by 11.6 %.

Summarized, our contributions are:

- We propose SecWalk, a generic method to protect the page table walk against fault attacks. Combined with encoded pointers and pointer arithmetic, we protect all memory accesses of a program against fault attacks.
- We integrate SecWalk to the open-source RISC-V processor CVA6 and evaluate its overhead based on a Field Programmable Gate Array (FPGA) implementation.
- To automatically protect arbitrary programs with SecWalk without user interaction, we develop a custom LLVM-based toolchain.
- To evaluate the software overhead and to show the practicability, we evaluate SecWalk on a set of microbenchmarks we port the microkernel seL4. We automatically replace all pointers, addresses, and memory accesses with their protected counterparts using our toolchain.

Scientific Contribution

Chapter 9 is primarily based on the following publication that was presented at the HOST 2021 in Washington D.C. (Washington D.C, USA).

Robert Schilling, Pascal Nasahl, Stefan Weiglhofer, and Stefan Mangard. “SecWalk: Protecting Page Table Walks Against Fault Attacks.” In: *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2021, Tysons Corner, VA, USA, December 12-15, 2021*. IEEE, 2021, pp. 56–67. DOI: 10.1109/HOST49136.2021.9702269

I am the main author of this paper, developed the technical idea, implemented the prototype toolchain, and performed all experiments and evaluations. Pascal Nasahl contributed to the text of this paper. Stefan Weiglhofer contributed to the prototype RISC-V hardware implementation. Stefan Mangard supported this work in many discussions and gave feedback on the paper.

Outline

The remainder of this chapter is structured as follows. Section 9.1 introduces page-based virtual memory. Section 9.2.1 discusses the threat model and existing fault attacks on virtual memory. Section 9.3 presents SecWalk, an efficient mechanism to protect virtual memory accesses against fault attacks. Section 9.4 describes the prototype implementation of SecWalk based on a RISC-V processor and discusses the toolchain, and in Section 9.5, we evaluate the performance of the implementation. Section 9.6 discusses related work and shows how SecWalk is superior. Finally, Section 9.7 concludes this chapter.

9.1 Page-based Virtual Memory

Page-based virtual memory [Lav78] or paging, is a well-known and widely used architecture to decouple the physical memory layout from the application and OS. A Memory-Management Unit (MMU) decouples the virtual address space from the constraint physical address space. The memory of a program is fragmented into smaller, fixed-size pages. The operating system creates a mapping between pages in the virtual address space and the pages in the physical address space. These mappings, *i.e.*, the Page Table Entries (PTEs), are stored in the page tables located in the page directory in the main memory. When running the program, the MMU uses the page tables to translate a virtual address to a physical address, called the *page table walk*. The physical address is eventually used for the actual memory access. As this translation is expensive, modern processors have a small cache in the MMU for storing the most recent translations, *i.e.*, the translation Translation Look-Aside Buffer (TLB), to have faster access to the physical address.

Apart from managing the memory, paging also provides isolation between different user-space applications. Since the OS sets up a different paging structure for every user-space process, thus every process can only see its own memory. Consequently, paging provides isolation between different user-space processes. However, the OS can still provide a memory-based communication interface between different processes, *i.e.*, it provides shared memory. Here, the same physical memory is mapped into the virtual memory space of two or more processes.

9.2 Threat Model and Attack Scenario

This section first presents the threat model and then shows how existing attacks in this threat model hijack virtual memory accesses. Finally, we discuss the required properties for protected memory accesses in the virtual memory domain.

9.2.1 Threat Model

In this work, we consider a powerful attacker capable of inducing faults with the goal of redirecting a virtual memory access. Thereby, we extend the threat model from Section 8.2 for application-class processors. The attacker aims to hijack the memory access by attacking the register file where a pointer is stored, pointer arithmetic, the memory access itself, or by manipulating the translation between the virtual and physical address. Consequently, we include the MMU, the TLB, and PTEs stored in memory to the threat model. Furthermore, we assume that the payload data of the application in memory is protected with a data encoding scheme.

Note that fault attacks on other parts of the processor, e.g., the instruction pipeline, the instruction pointer, the actual computation, or on other data, are not in the scope of this work. It requires orthogonal countermeasures, e.g., a Control-Flow Integrity (CFI) protection scheme such as FIPAC from Chapter 5 in combination with the conditional protection from Chapter 7 or hardware-enforced CFI tailored to fault attacks [Cle+16; Wer+18], which ensures the authentic and genuine execution of the instruction stream and its control-flow graph. The computation can either be protected with instruction replication or by using a data encoding scheme that supports encoded arithmetic operations. For complete protection against fault attacks, a combination of the protection of memory accesses such as SecWalk, the control-flow, and the computation is required. We now show how faults are used to hijack memory accesses in the virtual memory domain.

9.2.2 Faults on Virtual Memory

When dealing with larger application-class processors with virtual memory and MMUs, no efficient protection mechanism exists, leaving virtual memory accesses vulnerable to fault attacks. Especially, the page table walk, which translates a

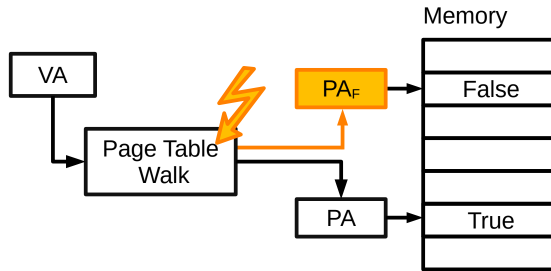


Figure 9.1: Attack vector: A faulted page table translation leads to a wrong memory access.

virtual to a physical address, is prone to fault attacks, which eventually leads to wrong memory accesses. Figure 9.1 illustrates the unprotected page table walk leading to a wrong address translation due to a fault. The virtual address VA is translated to a physical address PA during the page table walk. A precise fault in this page table translation can redirect the page table walk to return a different physical address PA_F, thus redirecting the subsequent memory access to a different location. This attack vector exists even if virtual or physical addresses include redundancy mechanisms such as data encoding. There is no efficient way of protecting the page table walk, and thus, memory accesses from the virtual domain against fault attacks. Similarly to that, also the MMU internal optimization buffer, *i.e.*, the TLB, suffers from the same attack vector. A fault can redirect the TLB to return a different and wrong page table entry, thus, redirecting a memory access to a wrong location.

Such an attack is presented in [Tro+21], where they use electromagnetic fault injection to induce faults to the MMU of a System-on-Chip (SoC). In their experiments, they are able to fault the virtual to physical mapping, therefore, redirecting the memory access to a different location.

[Goo15] describes a kernel privilege escalation, where the Rowhammer effect is used to manipulate the PTE stored in memory. By inducing faults to the PTE, the attacker is able to redirect the virtual to physical mapping of an attacker-controlled page. Eventually, this results in having read and write access to the attacker process's own page tables, yielding access to all physical memory and allowing the attacker to escalate privileges.

9.2.3 Requirements for Protected Virtual Memory Accesses

To protect memory accesses in the virtual memory domain against fault attacks with an easy application and to mitigate attacks, as discussed above, a protection scheme needs to fulfill the following requirements.

1. Pointers and addresses require an efficient protection mechanism against fault attacks, which also covers pointer arithmetic.

2. A link between the accessed data and the protected memory address is required to ensure the correct memory element was accessed.

3. In order to protect the virtual memory domain, the translation between virtual and physical addresses, including the TLB, must propagate the address redundancy.

4. To support arbitrary applications, the protection mechanism of virtual memory must support shared memory. Therefore, any linking between payload data and addresses must only operate on physical addresses.

5. To support legacy codebases and to enable easy deployment, the memory protection must be applied automatically, *i.e.*, during compilation, and must not require source code modifications.

Existing protection mechanisms for memory accesses are either not efficient [Sch+10] or, like the countermeasure presented in Chapter 8, do not support the protection of virtual and shared memory. Hence, there is a need for new and efficient protection schemes, protecting *all* memory accesses against fault attacks.

9.3 Design of SecWalk

This section presents SecWalk, an efficient protection scheme against fault attacks for all memory accesses in the virtual and physical memory domain, fulfilling the key requirements discussed above. We first introduce the design of protected pointers and then discuss the protected page table walk and TLB protection needed for virtual and shared memory.

9.3.1 Protected Pointers and Memory Accesses

Residual codes have been proven to be an efficient countermeasure to protect arithmetic operations against fault attacks. These codes can also be used to protect pointers and their respective pointer arithmetic. Similar to the protection mechanism for direct memory accesses, as presented in Chapter 8, we embed the redundancy of the residue code in the upper bits of the pointer by reducing its address space.

Our design uses the moduli set $M = \{5, 7, 17, 31, 127\}$ to protect pointers and addresses, which yields a Hamming distance of $D = 5$, capable of detecting up to four bitflips. Figure 9.2 shows a virtual memory address, where the upper bits denote the residue redundancy and the lower 39-bits the original pointer value. This separation – a residue-code is a systematic code – supports direct access to the payload data without a dedicated decode operation, which is crucial for a fast memory lookup on the unencoded address space. The address space of the pointer is reduced to 39-bits allowing the pointer to store up to 25-bits for redundancy purposes. The smaller address space aligns with existing systems such as Linux [Ghi21] for RISC-V, which only uses 39-bits in its virtual address

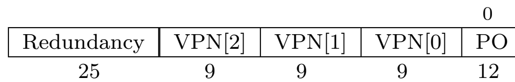


Figure 9.2: Encoded virtual address in Sv39. The upper 25-bits denote the redundancy information of the multi-residue code.

space. To efficiently operate on encoded pointers, we add new instructions to encode, decode, add, and subtract encoded pointers.

To protect the actual memory access, we establish a link between the encoded address and the actual data in the memory access. The linking operation scrambles the actual data when being written to memory and unscrambles it when reading it back using its encoded address information. We use a simple XOR-based link on byte granularity, where each encoded byte address scrambles the corresponding byte in the data. Only when reading from the correct memory location, the unscramble operation succeeds, and the correct data is loaded into the register of the processor. As the payload data uses a data encoding scheme, the unlink operation of a wrong memory access destroys the payload's redundancy properties. Thus, the wrong access is detectable in software.

9.3.2 Secure Page Table Walk

In order to protect the memory access in the virtual memory domain, it is required to provide a link between the virtual address and its translated physical address. To protect the actual memory access itself on the physical memory domain, an address-dependent link to the data is needed, which is applied during the page table walk.

The page table walk is the main operation to translate a virtual address to a physical address, which is eventually used for the memory access. In a protected program, all addresses, virtual and physical ones, are protected using the residual-based encoding scheme as described before. We now present the secure page table walk that translates a protected virtual address to a protected physical address and establishes a protected link in between. The design focuses on the RISC-V Sv39 virtual memory system [Wat+20], but the protection mechanism itself is generic and can also be applied to other virtual memory architectures.

In Sv39, a 39-bit virtual address is grouped into a 27-bit Virtual Page Number (VPN) and a 12-bit Page Offset (PO). During the three-step deep page table walk (the page table walk may abort early for larger pages), the VPN is translated to a 44-bit Physical Page number (PPN). The page offset remains untranslated. The final physical address is computed by concatenating the retrieved PPN with the page offset, forming a 64-bit address for the memory access. In Figure 9.2, we show the layout of a virtual address. Note that the upper bits of the address are used to store the redundancy information of the multi-residue code of the virtual address.

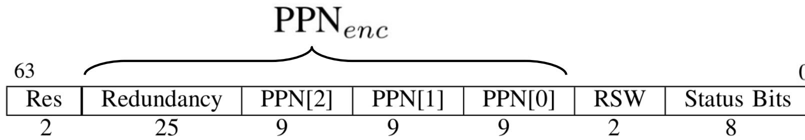


Figure 9.3: Sv39 page table entry with the extended encoded PPN to store the redundancy information.

To achieve a secure page table walk, we need to establish a link between the protected virtual address and the translated physical address. Only when performing the correct page table walk this link can be verified, and the page table walk is genuine. The verification is done by checking the integrity of the encoded PPN in the page table entry after applying the respective unlink operation. Otherwise, the translation yields an invalid PPN in terms of the encoding scheme. Due to the redundancy properties of the encoding scheme, the invalid PPN can be detected. However, the link, which is based on the virtual address, must not influence the actual physical address nor the data stored in the memory. This property is needed to support shared memory, where different virtual addresses map to the same physical address and data.

To design a protected link between the virtual and physical address of the page table walk and to make faults detectable, we add redundancy to a page table entry. We encode the PPN within the PTE using the same multi-residue code as used for pointers. We extend the size of the PPN by 8-bits to a total size of 52-bit, to include the redundancy information of the multi-residue code. Together with the 12-bit page offset, this forms a 64-bit physical address. Since the PPN is aligned to the page size of 4 KiB or larger, the lower 12-bits of the physical address pointed by the PPN are always zero. Eventually, PPN $\times$ 2^{12} forms a valid code word in terms of the multi-residue code, which can be verified. In Figure 9.3, we show the modified PTE, including the redundancy of the encoded PPN that we use to verify the correct translation. By including 25-bits of redundancy in the physical page number, we also reduce the physical address space to 39-bits.

The page table walk subsequently reads new page table entries, based on the VPN of the virtual address, from memory to determine the final physical address. In SecWalk, the PTEs are linked with the corresponding part of the VPN. Before using the PTE, it needs to be unlinked, followed by the verification of the residual integrity of the encoded PPN. If this check succeeds, the correct PTE was loaded from memory, and no wrong lookup was performed. If the check fails, it corresponds to an invalid memory read of the PTE or a manipulation of the PTE in memory. These steps, *i.e.*, the page table walk, are repeated until the final PTE is successfully loaded and the last encoded PPN is obtained. The last PPN itself is linked a second time with the fully encoded VPN, thus providing an end-to-end link between the encoded VPN and PPN. Finally, the physical address is computed by taking the encoded PPN and performing an encoded

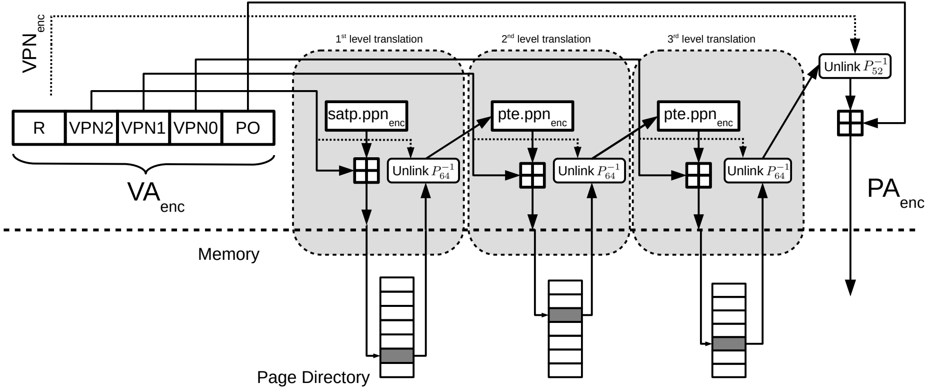


Figure 9.4: Secure page table walk with linked page table entries.

addition with the encoded page offset.

We achieve the link by applying a special linking function P_x to the PTEs during the page setup. During the page table walk, we apply the respective inverse unlink operation P_x^{-1}. Note that x denotes the bitwidth on which the linking is applied, which is 64-bit for linking the PTE. The final 52-bit encoded PPN in the last-level PTE is linked twice with a linking function where $x = 52$. This last step is needed to also incorporate the residual redundancy of VPN_{enc} to the page table walk. As soon as the page table walk is faulted and a wrong PTE is loaded, the unlinking step destroys the data such that the redundancy verification fails, which eventually causes a trap in the processor. Here, a fault during the address translation is transformed into a data error on the PTE, which is detectable due to its redundancy properties.

In RISC-V, the page table walk starts with a base register storing the initial physical page number. Similar to PPNs within a PTE, we also encode the initial PPN to the multi-residue domain stored within the Control and Status Register (CSR) `satp_enc`.ppn_{enc}. Note that we require a new CSR for this purpose to fit in the extended encoded PPN. Based on the original page table walk, as defined in the RISC-V privileged specification, we propose the following sequence of steps for the secure page table walk. The protected translation of the encoded virtual address VA_{enc} to the encoded physical address PA_{enc} works as follows. The suffix $_{enc}$ denotes multi-residue encoded data, $\boxplus$ the encoded addition, and $\boxminus$ an encoded subtraction. For Sv39, PAGE_SIZE is 2^{12}, and PTE_SIZE is 8.

1. Let a be `satp_enc`.ppn_{enc} $\times$ PAGE_SIZE and $i = 2$.

2. Let $\text{VPN}_{enc} = \text{VA}_{enc} \boxminus \text{Enc(PO)}$, where PO is the 12-bit page offset of the virtual address. Verify the lower 12-bit of VPN_{enc} to be zero.

3. Let the linked PTE_l be the value of the linked PTE at address $a \boxplus \text{VA}.vpn[i]$ $\times$ PTE_SIZE.

4. Perform the unlink step: $\text{PTE} = P_{64}^{-1}(\text{PTE}_l, \text{VA}.vpn[i])$.

5. If PTE.$r = 1$ or PTE.$x = 1$, we have a leaf PTE. Go to step 7.

6. The PTE is a pointer to the next level of the page table. Check the integrity the residue integrity of
PTE.ppn_{enc} × PAGE_SIZE. Fail if not valid. Let a be PTE.ppn_{enc} × PAGE_SIZE and $i = i - 1$. If $i < 0$, fail out. Continue at step 3.

7. A leaf PTE was found. Perform the second unlink operation of the PPN by PTE.$ppn_{enc} = P_{52}^{-1}$(PTE.ppn × PAGE_SIZE, VPN$_{enc}$) and check the residual integrity of PTE.ppn_{enc}. Fail if not valid.

8. The page table translation finished. The translated encoded physical address is given as
PA$_{enc}$ = PTE.$ppn_{enc} \boxplus$ PO$_{enc}$.

Note that original physical memory access and Physical Memory Protection (PMP) checks of RISC-V during the address translation are still in place. The page table walk returns an encoded physical address, which is then used for the linked memory access. In Figure 9.4, we visualize the page table walk using the steps as described before.

Linking Function

The general idea of the secure page table walk uses a linking function $P_x(y, k)$ to link the PTE with its corresponding parts of VPN. This link is performed on the whole PTE and in the last step only on the encoded PPN, thus requiring two different block sizes (52- and 64-bit). In the linking function, x denotes the block size, y is the data being linked, and k is the linking key.

To make the link secure but also practical, the un/linking function needs to fulfill three requirements.

1. The linking function $P_x(y, k)$ needs to be a bijective mapping, implying that there exists an inverse unlinking function $P_x^{-1}(y, k)$ such that $y = P_x^{-1}\left(P_x\left(y, k\right), k\right)$. During the page directory setup, the PTE gets linked using its corresponding part of the VPN as the linking key k. When performing the page table walk, the respective unlink operation is applied to retrieve the correct PTE data again.

2. The unlinking function is used to detect wrong page table walks by verifying the redundancy of the unlinked PTE. Thus, the unlinking function must not yield a correct code word in terms of the data encoding scheme if the wrong data is accessed.

3. Third, the linking function needs to provide diffusion over the whole data word, e.g., over the 64-bit PTE when $x = 64$. The diffusion is needed to mix all bits of the PTE, *i.e.*, the status bits and the PPN. Thus, an arbitrary fault on the PTE, even only on a status bit, also affects the redundancy bits of the encoded PPN. Therefore, a simple byte-granular XOR-based

linking function, such as the one used in Section 8.4.2, is not sufficient as there is no intra-word diffusion.

Many functions fulfill these requirements, but we aim for a small and efficient design in this work. We use a two-round reduced version of the PRINCE block cipher [Bor+12] to perform a 64-bit link, meeting the requirements discussed above. A round-reduced version of PRINCE is sufficient as the linking function only requires diffusion and no cryptographic strength. The second linking function aslo uses a two-round reduced version of PRINCE but with a reduced block size to 52-bit. The encryption operation of the cipher performs the linking operation, and the decryption operation performs the unlinking step, respectively.

In order to protect the page table walk, we apply the principle of linking and unlinking again, but between different levels of the page table walk. However, the linking is not applied on byte-level but on 64-bit word-level to ensure diffusion over the full page table entry.

9.3.3 TLB Design

The translation between virtual and physical addresses is a complex multi-step process, including multiple memory accesses under the hood. Modern processors have a dedicated cache for storing the most recent translations to speed up this operation, *i.e.*, the translation TLB. This buffer stores the most recent translations between virtual and physical addresses to avoid a costly MMU translation. The TLB is indexed using the VPN of the virtual address and returns the corresponding PTE if available. We apply the same 64-bit linking mechanism to secure this translation as used in the page table walk. The PTE in the TLB is linked using the encoded VPN_{enc} of the virtual address. When retrieving a PTE from the TLB, the PTE is unlinked, and the redundancy of the included PPN is verified. Only when using the correct encoded VPN_{enc} for unlinking the redundancy properties of the encoded PPN are preserved, and the lookup is valid. Otherwise, if the wrong or faulted VPN is used for unlinking, the redundancy properties of the encoded PPN are destroyed. In this case, the MMU traps and stops the application.

9.3.4 Page Directory Setup

When setting up virtual memory, it is necessary to create the corresponding mappings between virtual and physical addresses, *i.e.*, the page directory. This configuration is a manual task and is typically performed in software when the OS initializes a new process or a process asks for more memory. As discussed in Section 9.3.2, the different levels of the page table are linked using parts of the VPN as the linking key with a final link of the whole encoded VPN at the end. It is the page directory setup's responsibility to create these links.

There are different approaches possible for establishing these links. While creating this link can purely be done in software, we aim for a hardware-centric approach since the linking functionality is needed anyways for the TLB. We

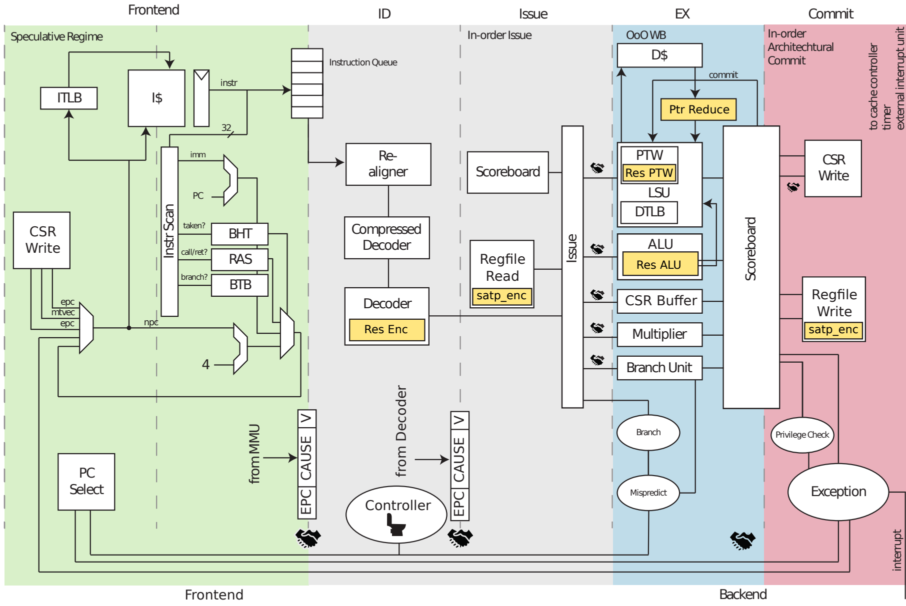


Figure 9.5: CVA6 hardware architecture with SecWalk. The yellow parts indicate changes in the design.

add a new instruction `vpnlink1 rd, rs1, rs2`, which creates the 64-bit link between the layers of the page tables. Furthermore, we add a second instruction `vpnlink2 rd, rs1, rs2`, which creates the final 52-bit link for the last PPN. These two instructions implement the linking functionality, as described above, and reuse already existing hardware blocks. While the speed might not be a requirement for the page table setup, *i.e.*, this is not an often used operation, the page table walk, which uses the inverse operation of the link, needs to be fast.

9.3.5 Shared Memory Support

SecWalk natively supports shared memory. By not having a hard link between the virtual address and the data in memory, a process can map the same physical page to multiple virtual addresses. Similarly, multiple processes can map the same physical page in their address space to allow inter-process communication. Due to the design of the page table walk, shared memory does not require to share any information between multiple mappings, as it is required for other protection schemes in related work.

9.4 Implementation

In this section, we first describe the hardware architecture of SecWalk and then discuss its custom toolchain to automatically instrument and protect arbitrary

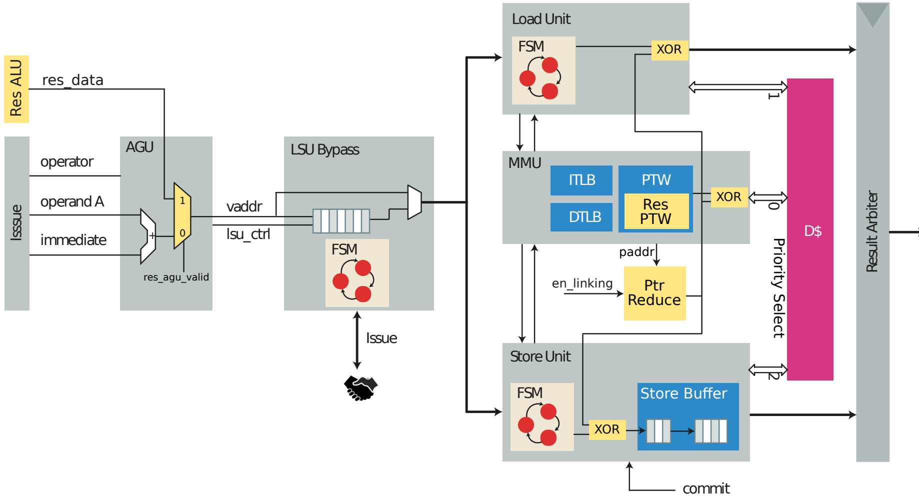


Figure 9.6: Hardware architecture the load-store-unit of CVA6. The yellow parts indicate changes in the design.

programs.

9.4.1 Hardware Implementation

We integrate SecWalk into the open-source RISC-V processor CVA6 [ZB19], formerly known as Ariane. CVA6 is a 64-bit, application-class, 6-stage, single issue, in-order RISC-V Central Processing Unit (CPU) written in SystemVerilog capable of running operating systems. In Figure 9.5, we show the modified hardware architecture of the processing system (the yellow parts indicate changes or additions). To support new instruction to deal with encoded pointers, e.g., add, subtract, encode, or decode, we extend the decoder and add a dedicated residue ALU. Furthermore, we add a CSR `satp_enc` to store the multi-residue encoded base address of the page directory needed for the page table walker. To support the linking operations needed for the page table setup, we add two new instructions `vpnlink1` and `vpnlink2`, to the decoder, which perform the 64-bit and 52-bit linking operation based on a round-reduced implementation of the PRINCE cipher.

In Figure 9.6, we show the modified Load-and-Store Unit (LSU) of the system. The LSU adds a new XOR-unit to the load- and store-unit, which is responsible for performing the linked memory address using the compressed encoded address coming from the ptr-reduce module. Furthermore, the MMU adds the residue-based page table walker, which transforms the encoded virtual address to the encoded physical address used for memory access in the load- or store-unit. The MMU has dedicated access to the memory to retrieve the page table entries needed for the address translation.

As shown in Figure 9.6, we extend the MMU with a dedicated Residue

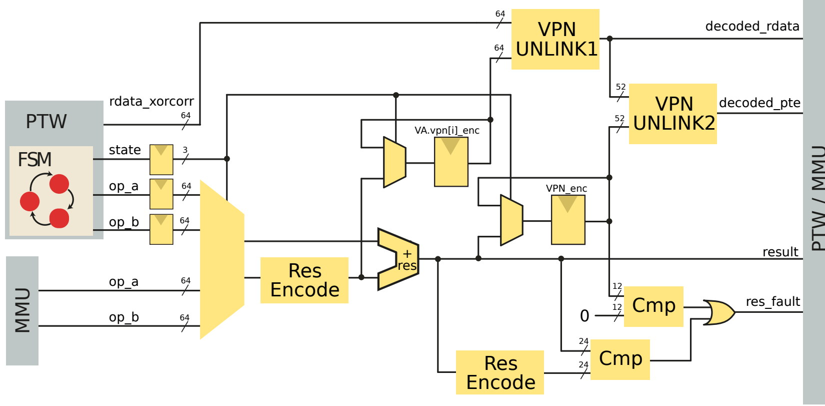


Figure 9.7: Residue page table walker exploiting the redundancy properties of residue codes.

Page Table Walker (ResPTW), detailed in Figure 9.7. The residue Page Table Walker (PTW) performs the additional operations needed by the existing PTW and MMU to enable secure virtual memory accesses. The block diagram in Figure 9.7 shows an overview of the implementation of the ResPTW, where it receives its data from the PTW or MMU and provides the results to the same two units. The original PTW, together with the residue PTW, performs the multi-level address translation according to the design of SecWalk. When an intermediate PTE is read, the 64-bit `vpnunlink1` operation decodes the whole PTE using the corresponding part of the VPN as the linking key. If a leaf PTE is read, the residue PTE performs the final 52-bit `vpnunlink2` operation to unlink the encoded page number using VPN_{enc} as the linking key. Both `vpnunlink` operations are based on a round-reduced version of the PRINCE block cipher with different block sizes. The final address is computed by adding the encoded page offset to the final PPN from the leaf PTE. Note that a residue addition is performed rather than a simple concatenation to yield an encoded physical address, which can be used to access the memory.

If the TLB already contains the requested translation, the PTW is not needed, and the MMU only requests the computation of PA_{enc}. The `vpnunlink` operations of the residue PTW are used to decode the accessed entry from the TLB. Note that all residual operations, *i.e.*, an addition, contain an integrated check with respect to the redundancy bits. As soon as an invalid code word is detected, the MMU traps, leading to aborting the program execution.

Although the prototype of SecWalk is based on the CVA6 processor, the protection mechanism is generic. Thus, SecWalk is compatible with other 64-bit RISC-V designs such as Rocket [Asa+16], (Sonic)Boom [CPA15; Zha+20], and many others. The only hard requirement is being able to modify the core, *i.e.*, having access to the source code. Furthermore, the protected page table walk itself is generic; thus, it is also applicable to other architectures. For example, the

ARM AArch64 architecture supports a similar 39-bit addressing scheme, where SecWalk can be added if core changes are possible.

9.4.2 Toolchain Implementation

To automatically compile arbitrary software for SecWalk, we develop a custom toolchain based on the LLVM compiler [LA04]. In this work, we reuse a modified toolchain from the protection scheme of Chapter 8 to encode pointers and pointer arithmetic to the multi-residue domain and only use linked memory accesses. Note that the toolchain currently does not support the automatic instrumentation of inline assembly code. If a program uses inline assembly, it requires the developer to manually modify the assembly code to use protected pointers and memory accesses.

To run a protected program, it requires support from the operating system. When starting a new application, the operating system takes care of setting up the memory mappings of the process. This part of the program requires a modification to take the linked page table entries into account. It needs to incorporate `vpnlink1` and `vpnlink2` to set up the link such that the hardware page table walker can unlink them when required. This task is a manual process and is not covered by the LLVM-based toolchain.

9.5 Evaluation

In this section, we first provide an evaluation showcasing the overheads of SecWalk in terms of hardware, code size, and runtime. We then discuss the security properties and how it protects against the defined threat model.

9.5.1 Hardware Evaluation

To measure the hardware overhead, we synthesize the design for a Xilinx Kintex-7 series FPGA. Our evaluation shows the prototype implementation of SecWalk increases the area of the design by less than 0.5 % in terms of flip-flops and 10 % in terms of lookup tables. In Table 9.1, we further split the utilization of the overheads between the handling of protected pointers and the changes related to virtual address translation in the MMU. Note that the hardware changes of SecWalk do not affect the critical path of CVA6, and the synthesis still reaches the original target frequency of 50 MHz.

9.5.2 Performance Evaluation

To evaluate the performance of SecWalk, we measure the code and runtime overhead of a set of microbenchmarks and then extend the evaluation to a micro-kernel. We use the custom LLVM-based toolchain to automatically instrument the programs and to transform all pointer arithmetic and memory instructions to the protected domain. The startup code configures the MMU and maps the

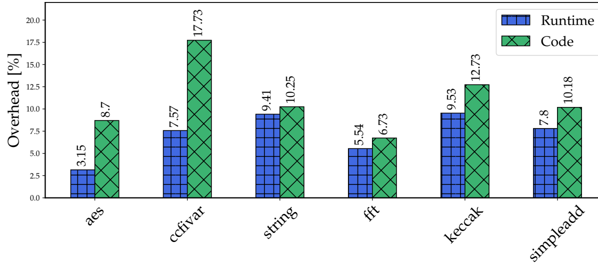


Figure 9.8: Performance evaluation using microbenchmarks.

virtual and physical pages accordingly. In Figure 9.8, we summarize the runtime and code overhead for the microbenchmark suite. SecWalk adds an average runtime overhead of 7.17 % and an average code size overhead of 11.05 %.

To showcase the applicability of our design for a larger application, we port the formally verified microkernel seL4 [EKE08; Kle+09; Kle+10] to SecWalk, which is used in many security-critical applications. seL4 already supports RISC-V but still requires minor adoptions for our design. First, we shift the operating system's address space to fit into the modified address layout of encoded pointers with its reduced address space. The instrumentation of the assembly code of seL4 requires manual modification, but these changes are minimal. The most crucial change in software is setting up the page tables using the custom linking instructions. These instructions are used to create the link between the virtual and physical addresses in the page directory, which are unlinked during the page table walk. When compiling seL4 with the extensions of SecWalk, we see an increase of 13.1 % in code size, which is solely coming from using protected pointers and pointer arithmetic. Conceptually, the only actual software overhead for the protection of virtual memory is the setup of page tables using the new linking instructions, which is negligible. When running the protected seL4 kernel on the prototype, the runtime in terms of cycles increases by 11.6 %. Both overhead numbers are reasonable considering that all pointers, all pointer arithmetic, and every memory access of the system is protected against fault attacks.

Table 9.1: Hardware utilization of SecWalk.

Hardware Overhead	LUTs [%]	Flip-flops [%]
Protected Pointers	6.4	0.12
Residue PTW	3.6	0.32
Sum	10.0	0.44

9.5.3 Security Evaluation

The protection of addresses and pointers in the virtual memory domain using the multi-residue code with the described set of moduli yields code words with a Hamming distance of $D = 5$ bits. Thus, this encoding scheme is able to detect up to four bitflips on pointers and addresses and its supported pointer arithmetic. Suppose the compiler detects an operation that is not supported by multi-residue codes, *i.e.*, a bitwise operation. In that case, it decodes the encoded pointer, performs the unsupported operation on the plain data, and then re-encodes the data back to the multi-residue domain. While the prototype implementation currently leaves the pointer unprotected for a short moment, other forms of redundancy, e.g., spatial redundancy, can be used to protect the pointer during such an operation. For example, instruction replication [BCR16; Hu+05] can be used to protect pointer arithmetic through unsupported operations by the multi-residue code. Note that such unsupported operations only occur very rarely, as pointer arithmetic tends to use simple operations such as additions and subtractions, which can operate in the protected domain.

The page table entries contain a multi-residue encoded PPN with a Hamming distance of $D = 5$ bits. The secure page table walk incorporates multiple operations in the encoded multi-residue domain. Throughout the translation of the virtual address, all residue additions of the page table walk are followed by a check operation in hardware, as depicted in Figure 9.7. Thus, faults cannot accumulate over multiple operations on the multi-residue code. With the selected parameters, the protected page table walk provides protection against four random bitflips.

9.6 Related Work

Starting with the ARMv8.3-A instruction set, ARM developed a feature named ARM Pointer Authentication [ARM20; Qua17]. This feature adds new instructions allowing the software to sign and verify a pointer cryptographically. The truncated Message Authentication Code (MAC) is thereby stored in the upper bits of the pointer, reducing its address space. Before accessing the memory, the pointer is authenticated, and the MAC is removed from the pointer. Then, a memory load or store operation can access the memory using the authenticated pointer. While ARM Pointer Authentication has similar design decisions, its scope of protection is different. They protect special pointers at runtime, *i.e.*, the stack pointer, to protect against classical software attackers [Lil+19]. However, they cannot protect pointer arithmetic, nor can they protect the memory access itself.

There are related works in the context of protecting memory accesses against fault attacks. ANB-codes [Sch+10] assign each variable a dedicated signature B at compile-time. When reading the data back from the memory, this signature is verified using the underlying data encoding scheme of ANB-codes. If this signature cannot be verified, it means the memory access was redirected and

read from a different location. Due to the static assignment of these signatures at compile-time, ANB-codes can only protect static memory and no dynamic allocations. Furthermore, they do not support shared memory, thus providing only a limited scope of protection for their expensive costs.

The protection mechanism presented in Chapter 8 adds redundancy to the pointer to perform linked memory accesses. To compensate for the overheads of encoded pointer arithmetic, they extend the processor with new instructions and develop a compiler using them. While their overheads are reasonably low, their protection mechanism only supports bare-metal applications of small embedded use cases. There is no support for virtual and shared memory; thus, it cannot protect memory accesses of application-class processors against faults.

SecWalk is superior to other protection mechanisms for memory accesses. While it has a low performance penalty, SecWalk outperforms related work in terms of supported features. SecWalk supports the protection of virtual memory accesses against fault attacks, including dynamic allocations and shared memory between different processes. In Table 9.2, we summarize the comparison of SecWalk against ANB-codes and purely encoded pointers.

Table 9.2: Feature comparison of SecWalk compared to related work.

Protection Scheme	Protection of Virtual Memory	Protection of Shared Memory	Overhead
ARM Pointer Authentication	✗	✗	Low
ANB-Codes	✓	✗	High
Encoded Pointer	✗	✗	Low
SecWalk	✓	✓	Low

9.7 Conclusion

The correct execution of a load or store operation is essential for the security of the system. With the rise of more powerful embedded systems, operating systems with virtual memory are commonly deployed in the IoT. When fault attacks are considered, virtual memory accesses cannot be trusted as there are different attacks possible which redirect the memory to a different location. Currently, there is no economic mechanism available that protects virtual memory accesses against fault attacks, including dynamic and shared memory.

In this chapter, we closed this gap and presented SecWalk, an efficient design to protect all memory accesses of a program in the virtual and physical domain against fault attacks. SecWalk protects all pointers and addresses in the virtual address space using a multi-residue code with no additional storage overhead. Furthermore, this encoding scheme supports encoded operations, thus also protecting the pointer arithmetic. We extend the domain of protection and develop a secure page table walk that propagates the redundancy from the virtual address to the physical address used for the memory access. The core idea of SecWalk is to add redundancy to page table entries, add a linking mechanism between virtual and physical addresses, and then verify the redundancy properties on the page table walk. The protection is comprehensive, covering the virtual address domain, the address translation within the MMU and TLB, and the actual memory access using the translated physical address. Furthermore, SecWalk supports arbitrary applications, including dynamic and shared memory.

We implemented SecWalk on an open-source RISC-V processor and mapped the design to an FPGA to showcase the hardware overhead. We developed a custom LLVM-based toolchain to automatically instrument arbitrary programs without user interaction. To evaluate the performance of SecWalk, we compile and execute a set of microbenchmarks. Furthermore, we integrate SecWalk into the existing microkernel seL4 to show its applicability to real-life applications using dynamic and shared memory. Our evaluation shows the hardware and software overheads of SecWalk are reasonable, considering that it protects all memory accesses of a program against fault attacks.

10
Conclusion and Outlook

In this thesis, we work towards protecting general-purpose software against fault attacks. Our advanced methodologies exploit the symbiosis of small hardware changes and the help of a compiler to automatically protect arbitrary software against fault attacks. By using this approach, we developed multiple techniques to extend the security of data, control-flow, and the memory subsystem towards this thesis.

The first part of the thesis was devoted to cryptographic offerings for fault- and side-channel secure encryption schemes. We showed that by using countermeasures against fault attacks at the algorithmic level, we can build energy-efficient hardware accelerators that can be used for today's requirements of IoT end-nodes. We integrated the accelerator into a multi-core SoC and explored the applicability of the system to different use cases.

Chapter 5, Chapter 6, and Chapter 7 were devoted to advancing the protection of the program's control-flow against fault attacks. First, in Chapter 5, we showed that existing architectural features of processors could be repurposed to develop countermeasures for fault security. We used ARM PA to develop FIPAC, which protects the control-flow of a program against software- and fault-based control-flow attacks at the basic block level. ARM PA was used to compute an authenticated CFG at compile-time, which is verified during runtime. By developing different checking policies, the designer can trade off the detection latency of an attack and the overheads of our protection scheme. Furthermore, since FIPAC is a software-based approach, it can be applied selectively to critical code where protection is needed.

In Chapter 6, we took FIPAC, which was designed to protect only the user-space of a program, and extended its level of protection to the kernel domain. We exploited the cryptographic state, which is the main pillar of FIPAC's protection,

and merged it with the system call interface to the kernel. By linking the system call to the CFI state, we protected the integrity of the system call flow against fault attacks with almost no additional overheads. Furthermore, by exploiting dynamic CFI instrumentation at program startup, we limit possible side-channel leakage during the CFI-protected software execution.

When looking back at the advances in CFI protection of this thesis, we showed that the cryptographic state of the CFI is a versatile building block for different countermeasures. By linking other data, e.g., the system call or a branch value, to that state, we transform errors between protection domains. This allows the CFI protection to take over the error detection, which is already in place. Consequently, it reduces the performance penalty since dedicated checks are not needed anymore.

During the last part of this thesis, we shifted the focus to protecting the memory subsystem of modern processing architectures against fault attacks. In Chapter 8, we showed that well-researched encoding schemes like multi-residue codes can be used to efficiently protect pointers. The pointer's limited set of operations fall into the class of natively supported operations of the residue code. Thus, we can protect pointers at rest, transportation, and also during computation without decoding them to plain values. By cleverly selecting the parameters for the residue code, the storage costs of the redundancy bits came to a minimum. We extended the scope of protection and used encoded pointers for scrambled memory accesses. Wrong memory accesses subsequently infected the data value allowing the software to detect the addressing error. This design allowed us to protect memory redirects even on the bus level.

In Chapter 9, we extended the memory access protection from bare-metal systems to application class processors and developed SecWalk. By developing a protected page table walk, we translated encoded pointers from the virtual memory domain to the physical one. We used the redundancy properties of residue codes throughout the page table walk, providing fault security in all memory domains. Our approach is compatible with mechanisms like shared memory, which is superior to state-of-the-art protection mechanisms in that area. With a hardware research prototype and a port of an off-the-shelf OS, we showed the applicability of SecWalk for real-life applications.

Outlook

In this thesis, we looked at different aspects of protecting software against fault attacks. A compiler-assisted approach, in combination with small hardware changes to the processor, is an efficient approach to deploying countermeasures to existing codebases. However, with fault attacks becoming available to larger systems, the protection of general-purpose software against fault attacks still requires additional research efforts in different directions.

Data Protection

In terms of protecting software against fault attacks, this thesis focused on protecting the control-flow of a program and the memory subsystem. Protecting conditional branches with our approach closed the gap where data from the program influences the control-flow. Our mechanism automatically encodes all branch-dependent data to the redundant AN-code domain that can be used to implement a secure comparison algorithm.

While this is a first step in the right direction of protecting data and its computation, it is not yet solved. Unfortunately, not all encoding schemes support all operations that are typically used within standard programs. While arithmetic codes natively support arithmetic operations in the encoded domain, they lack support for protected binary operations. Other codes, e.g., binary linear codes, support binary operations but cannot be used in ordinary arithmetic. Unfortunately, there are no conversion algorithms available that convert between different encoding schemes and preserve the redundancy properties of both encoding schemes.

A different approach to protect the computation is temporal redundancy in the form of duplicated instructions. Modern application-class cores already have multiple functional units to implement their super-scaler architectures. To include redundancy for the computation, the issuing unit of the processor can schedule the same instruction multiple times. At the end of the execution pipeline, the processor compares the results from all computations and triggers an exception if they mismatch. A CPU can even retry the computation in case of a failure. While this approach can be easily implemented for idempotent instructions, such as simple arithmetic operations, non-idempotent instructions, *i.e.*, a store to a memory-mapped device, must be handled differently. Ideally, this approach is cheaper in terms of overheads compared to simple duplication, *i.e.*, a lock-step of two cores. Furthermore, such an approach provides certain degrees of flexibility since the countermeasure can be activated only for some areas of the software that require a higher degree of protection.

Transparent Memory Encryption

Memory encryption is a trending topic for all modern computing platforms. While previously only used for special-purpose platforms, memory encryption is nowadays even included in commodity desktop systems. Memory encryption adds certain overheads to the system. Thus, the deployed cryptographic modes of operation are optimized for latency and throughput but not protected against faults. Future research could investigate the applicability of fault-secure cryptographic modes that can be used for high-performance applications such as application-class processors. With the rise of tweakable encryption schemes, the primitives can process additional data, *i.e.*, the tweak. This input can further be used to bind additional data to the encryption, e.g., to build cryptography-based memory safety.

Confidential Computing

Confidential computing has become a hot topic for existing and new processing architectures. However, with recent software-induced fault attacks becoming a threat to large-scale systems, they are also a threat to confidential computing. Fault attacks on Intel SGX and ARM TrustZone show that existing TEEs are not yet protected from these new threats. The developed countermeasures, especially the pure software-based approaches, can help to protect TEEs on existing hardware. Future system designs, especially newly developed standards for confidential computing, can incorporate those approaches to provide more in-depth security also with hardware support.

Attestation and Licensing

The developed control-flow mechanisms in this thesis allow the system to ensure that the program follows the right direction within the program. This mechanism can be extended to provide dynamic remote attestation for user applications. Especially for platforms supporting confidential computing, remote attestation can provide further measures of security.

In addition, the developed countermeasures in the context of CFI in Chapter 5 can be extended to provide fine-granular software licensing. Only if the system has loaded the correct encryption keys, *i.e.*, the license key, the control-flow to certain functions is valid. Thus, paid library functions can only be called if a valid license is available. In case of an invalid key or license, calling such a function would yield a detectable control-flow error.

Formal Verification

In this thesis, we analytically looked at the security of our countermeasures or used fault simulation environments. However, especially in high-security applications, *i.e.*, in the payment sector, pure fault simulation is often not enough. To improve the situation, formal verification can be used to provide a mathematical proof for certain properties. Recent developments in formal methods show that their performance scales better, thus, they can be used to prove the security properties of larger systems. Formal tools can be used during the development of hardware and software. In the context of hardware development, formal tools are already used to prove the functional correctness of designs. However, these tools can also be used in the security domain, e.g., they can be used to prove that fault-related countermeasures are preserved during a synthesis step. A proof then shows that a hardware design is capable of detecting up to a certain number of bitflips. On the software side, formal tools can possibly be integrated directly into a compiler that automatically generates a proof for the compiled and instrumented software.

List of Contributions

In this chapter, Section 10.1 first presents the list of publications on which this thesis is directly based on. Section 10.2 lists other publications, where I am the co-author, but which are not part of this thesis.

10.1 Main Publications

[Sch+23] **Robert Schilling**, Pascal Nasahl, Martin Unterguggenberger, and Stefan Mangard. "SFP: Providing System Call Flow Protection against Software and Fault Attacks." In: *CoRR* abs/2301.02915 (2023). DOI: 10.48550/arXiv.2301.02915.

[SNM22b] **Robert Schilling**, Pascal Nasahl, and Stefan Mangard. "FIPAC: Thwarting Fault- and Software-Induced Control-Flow Attacks with ARM Pointer Authentication." In: *Constructive Side-Channel Analysis and Secure Design - 13th International Workshop, COSADE 2022, Leuven, Belgium, April 11-12, 2022, Proceedings*. Springer, 2022, pp. 100–124. DOI: 10.1007/978-3-030-99766-3_5.

[Sch+21] **Robert Schilling**, Pascal Nasahl, Stefan Weiglhofer, and Stefan Mangard. "SecWalk: Protecting Page Table Walks Against Fault Attacks." In: *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2021, Tysons Corner, VA, USA, December 12-15, 2021*. IEEE, 2021, pp. 56–67. DOI: 10.1109/HOST49136.2021.9702269.

[Sch+18b] **Robert Schilling**, Thomas Unterluggauer, Stefan Mangard, Frank K. Gürkaynak, Michael Muehlberghuber, and Luca Benini. "High speed ASIC implementations of leakage-resilient cryptography." In: *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*. IEEE, 2018, pp. 1259–1264. DOI: 10.23919/DATE.2018.8342208.

[SWM18] **Robert Schilling**, Mario Werner, and Stefan Mangard. "Securing conditional branches in the presence of fault attacks." In: *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*. IEEE, 2018, pp. 1586–1591. DOI: 10.23919/DATE.2018.8342268.

[Sch+18c] **Robert Schilling**, Mario Werner, Pascal Nasahl, and Stefan Mangard. "Pointing in the Right Direction - Securing Memory Accesses in a Faulty World." In: *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, pp. 595–604. DOI: 10.1145/3274694.3274728.

[Con+17b] Francesco Conti, **Robert Schilling**, Pasquale Davide Schiavone, Antonio Pullini, Davide Rossi, Frank Kagan Gürkaynak, Michael Muehlberghuber, Michael Gautschi, Igor Loi, Germain Haugou, Stefan Mangard, and Luca Benini. "An IoT Endpoint System-on-Chip for Secure and Energy-Efficient Near-Sensor Analytics." In: *IEEE Trans. Circuits Syst. I Regul. Pap.* 64-I (2017), pp. 2481–2494. DOI: 10.1109/TCSI.2017.2698019.

10.2 Contributed Publications

[Nas+23] Pascal Nasahl, Martin Unterguggenberger, Rishub Nagpal, **Robert Schilling**, David Schrammel, and Stefan Mangard. "SCFI: State Machine Control-Flow Hardening Against Fault Attacks." In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2023, Antwerp, Belgium, April 17-19, 2023*. IEEE, 2023, pp. 1–6. DOI: 10.23919/DATE56975.2023.10137038.

[Unt+23] Martin Unterguggenberger, David Schrammel, Pascal Nasahl, **Robert Schilling**, Lukas Lamster, and Stefan Mangard. "Multi-Tag: A Hardware-Software Co-Design for Memory Safety based on Multi-Granular Memory Tagging." In: *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, ASIA CCS 2023, Melbourne, VIC, Australia, July 10-14, 2023*. ACM, 2023, pp. 177–189. DOI: 10.1145/3579856.3590331.

[NSM21] Pascal Nasahl, **Robert Schilling**, and Stefan Mangard. "Protecting Indirect Branches Against Fault Attacks Using ARM Pointer Authentication." In: *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2021, Tysons Corner, VA, USA, December 12-15, 2021*. IEEE, 2021, pp. 68–79. DOI: 10.1109/HOST49136.2021.9702268.

[Nas+21a] Pascal Nasahl, **Robert Schilling**, Mario Werner, Jan Hoogerbrugge, Marcel Medwed, and Stefan Mangard. "CrypTag: Thwarting Physical and Logical Memory Vulnerabilities using Cryptographically Colored Memory." In: *ASIA CCS '21: ACM Asia Conference on Computer and Communications Security, Virtual Event, Hong Kong, June 7-11, 2021*. ACM, 2021, pp. 200–212. DOI: 10.1145/3433210.3453684.

[Nas+21b] Pascal Nasahl, **Robert Schilling**, Mario Werner, and Stefan Mangard. "HECTOR-V: A Heterogeneous CPU Architecture for a Secure RISC-V Execution Environment." In: *ASIA CCS '21: ACM Asia Conference on Computer and Communications Security, Virtual Event, Hong Kong, June 7-11, 2021*. ACM, 2021, pp. 187–199. DOI: 10.1145/3433210.3453112.

[Sch+20] Michael Schwarz, Moritz Lipp, Claudio Canella, **Robert Schilling**, Florian Kargl, and Daniel Gruss. "ConTExT: A Generic Approach for Mitigating Spectre." In: *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. URL: https://www.ndss-symposium.org/ndss-paper/context-a-generic-approach-for-mitigating-spectre/.

[Kar+19] Anja F. Karl, **Robert Schilling**, Roderick Bloem, and Stefan Mangard. "Small Faults Grow Up - Verification of Error Masking Robustness in Arithmetically Encoded Programs." In: *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*. Springer, 2019, pp. 183–204. DOI: 10.1007/978-3-030-11245-5_9.

[Wer+19] Mario Werner, **Robert Schilling**, Thomas Unterluggauer, and Stefan Mangard. "Protecting RISC-V Processors against Physical Attacks." In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*. IEEE, 2019, pp. 1136–1141. DOI: 10.23919/DATE.2019.8714811.

[Gür+17] Frank K. Gürkaynak, **Robert Schilling**, Michael Muehlberghuber, Francesco Conti, Stefan Mangard, and Luca Benini. "Multi-core data analytics SoC with a flexible 1.76 Gbit/s AES-XTS cryptographic accelerator in 65 nm CMOS." In: *Proceedings of the Fourth Workshop on Cryptography and Security in Computing Systems, CS2@HiPEAC 2017, Stockholm, Sweden, January 24, 2017*. ACM, 2017, pp. 19–24. DOI: 10.1145/3031836.3031840.

[Unt+17] Thomas Unterluggauer, Thomas Korak, Stefan Mangard, **Robert Schilling**, Luca Benini, Frank K. Gürkaynak, and Michael Muehlberghuber. "Leakage Bounds for Gaussian Side Channels." In: *Smart Card Research and Advanced Applications - 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13-15, 2017, Revised Selected Papers*. Springer, 2017, pp. 88–104. DOI: 10.1007/978-3-319-75208-2_6.

[Wer+17] Mario Werner, Thomas Unterluggauer, **Robert Schilling**, David Schaffenrath, and Stefan Mangard. "Transparent memory encryption and authentication." In: *27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium,*

September 4-8, 2017. IEEE, 2017, pp. 1–6. DOI: 10.23919/FPL.2017.8056797.

[Sch+14] **Robert Schilling**, Manuel Jelinek, Markus Ortoff, and Thomas Unterluggauer. "A low-area ASIC implementation of AEGIS128—A fast authenticated encryption algorithm." In: *22nd Austrian Workshop on Microelectronics (Austrochip)*. 2014, pp. 1–5.

Bibliography

[Aba+05] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-flow integrity." In: *Conference on Computer and Communications Security – CCS*. 2005, pp. 340–353. DOI: 10.1145/1102120.1102165.

[AG03] Mehdi-Laurent Akkar and Louis Goubin. "A Generic Protection against High-Order Differential Power Analysis." In: *Fast Software Encryption – FSE*. 2003, pp. 192–205. DOI: 10.1007/978-3-540-39887-5_15.

[AMT13] Subidh Ali, Debdeep Mukhopadhyay, and Michael Tunstall. "Differential fault analysis of AES: towards reaching its limits." In: *J. Cryptogr. Eng.* 3 (2013), pp. 73–97. DOI: 10.1007/s13389-012-0046-y.

[Amb15] Ambiq. *Apollo Ultra-Low-Power Microcontrollers*. https://ambiq.com/apollo. [accessed 2023-01-01]. 2015.

[AK] Ross Anderson and Markus Kuhn. "Tamper resistance-a cautionary note." In: *Proceedings of the second Usenix workshop on electronic commerce*, pp. 1–11.

[App20] Apple Inc. *Preparing Your App to Work with Pointer Authentication*. https://developer.apple.com/documentation/security/preparing_your_app_to_work_with_pointer_authentication. [accessed 2023-01-01]. 2020.

[App21] Apple Inc. *Apple SoC security*. https://support.apple.com/guide/security/apple-soc-security-sec87716a080/web. [accessed 2023-01-01]. 2021.

[ARM10] ARM Limited. *CMSIS - Cortex Microcontroller Software Interface Standard*. https://developer.arm.com/tools-and-software/embedded/cmsis. [accessed 2023-01-01]. 2010. URL: https://www.arm.com/products/processors/cortex-m/cortex-microcontroller-software-interface-standard.php.

[ARM20] ARM Limited. *Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*. https://documentation-service.arm.com/static/5fa3bd1eb209f547eebd4141. [accessed 2023-01-01]. 2020.

[ARM21] ARM Limited. *Arm partners are shipping more than 900 Arm-based chips per second based on latest results*. https://www.arm.com/company/news/2021/05/arm-partners-are-shipping-more-than-900-arm-based-chips-per-second. [accessed 2023-01-01]. 2021.

[ARMa] ARM Limited. *Arm Architecture Reference Manual for A-profile architecture, v8.3A*. https://developer.arm.com/documentation/ddi0487/ca. [accessed 2023-01-01].

[ARMb] ARM Limited. *Arm Architecture Reference Manual for A-profile architecture, v8.6A*. https://developer.arm.com/documentation/ddi0487/fa. [accessed 2023-01-01].

[ARMc] ARM Limited. *Arm's solution to the future needs of AI, security and specialized computing is v9*. https://www.arm.com/company/news/2021/03/arms-answer-to-the-future-of-ai-armv9-architecture. [accessed 2023-01-01].

[ARMd] ARM Limited. *Inside the numbers: 100 billion ARM-based chips*. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/inside-the-numbers-100-billion-arm-based-chips-1345571105. [accessed 2023-01-01].

[Asa+16] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. "The rocket chip generator." In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* (2016).

[Atm17] Atmel. *Atmel AT02333: Safe and Secure Bootloader Implementation for SAM3/4*. http://www.atmel.com/Images/Atmel-42141-SAM-AT02333-Safe-and-Secure-Bootloader-Implementation-for-SAM3-4_Application-Note.pdf. [accessed 2023-01-01]. 2017.

[Ava17] Roberto Avanzi. "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." In: *IACR Trans. Symmetric Cryptol.* (2017), pp. 4–44. DOI: 10.13154/tosc.v2017.i1.4-44.

[Aza19] Brandon Azad. *Examining Pointer Authentication on the iPhone XS*. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html. [accessed 2023-01-01]. 2019.

[Bar+06] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. "The Sorcerer's Apprentice Guide to Fault Attacks." In: *Proc. IEEE* 94 (2006), pp. 370–382. DOI: 10.1109/JPROC.2005.862424.

[Bar+09] Alessandro Barenghi, Guido Bertoni, Emanuele Parrinello, and Gerardo Pelosi. “Low Voltage Fault Attacks on the RSA Cryptosystem.” In: *Fault Diagnosis and Tolerance in Cryptography – FDTC*. 2009, pp. 23–31. DOI: 10.1109/FDTC.2009.30.

[Bar+10a] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. “Countermeasures against fault attacks on software implemented AES: effectiveness and cost.” In: *International Conference on Compilers, Architectures, and Synthesis for Embedded Systems – CASES*. 2010, p. 7. DOI: 10.1145/1873548.1873555.

[Bar+10b] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. “Low Cost Software Countermeasures Against Fault Attacks: Implementation and Performances Trade Offs.” In: *Proceedings of 5th Workshop on Embedded Systems Security - WESS* (2010).

[BCR16] Thierno Barry, Damien Couroussé, and Bruno Robisson. “Compilation of a Countermeasure Against Instruction-Skip Fault Attacks.” In: *International Conference on High-Performance Embedded Architectures and Compilers – HiPEAC*. 2016, pp. 1–6. DOI: 10.1145/2858930.2858931.

[Bel+15] Sonia Belaïd, Jean-Sébastien Coron, Pierre-Alain Fouque, Benoît Gérard, Jean-Gabriel Kammerer, and Emmanuel Prouff. “Improved Side-Channel Analysis of Finite-Field Multiplication.” In: *Cryptographic Hardware and Embedded Systems – CHES*. 2015, pp. 395–415. DOI: 10.1007/978-3-662-48324-4_20.

[BFG14] Sonia Belaïd, Pierre-Alain Fouque, and Benoît Gérard. “Side-Channel Analysis of Multiplications in GF(2128) - Application to AES-GCM.” In: *Advances in Cryptology – ASIACRYPT*. 2014, pp. 306–325. DOI: 10.1007/978-3-662-45608-8_17.

[Ben+15] Simone Benatti, Filippo Casamassima, Bojan Milosevic, Elisabetta Farella, Philipp Schoenle, Schekeb Fateh, Thomas Burger, Qiuting Huang, and Luca Benini. “A Versatile Embedded Platform for EMG Acquisition and Gesture Recognition.” In: *IEEE Trans. Biomed. Circuits Syst.* 9 (2015), pp. 620–630. DOI: 10.1109/TBCAS.2015.2476555.

[Ben+16] Simone Benatti, Fabio Montagna, Davide Rossi, and Luca Benini. “Scalable EEG seizure detection on an ultra low power multi-core architecture.” In: *Biomedical Circuits and Systems – BIOCAS*. 2016, pp. 86–89. DOI: 10.1109/BioCAS.2016.7833731.

[Ber+09] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. “Keccak Sponge Function Family Main Document.” In: *Submission to NIST (Round 2)* 3 (2009), p. 30.

[BGN05] Eli Biham, Louis Granboulan, and Phong Q. Nguyen. "Impossible Fault Analysis of RC4 and Differential Fault Analysis of RC4." In: *Fast Software Encryption – FSE*. 2005, pp. 359–367. DOI: 10.1007/11502760_24.

[BS97] Eli Biham and Adi Shamir. "Differential Fault Analysis of Secret Key Cryptosystems." In: *Advances in Cryptology – CRYPTO*. 1997, pp. 513–525. DOI: 10.1007/BFb0052259.

[Bit22] Bitcraze. *The Crazyflie Nano Quadcopter*. https://www.bitcraze.io/products/crazyflie-2-1. [accessed 2023-01-01]. 2022.

[BS03] Johannes Blömer and Jean-Pierre Seifert. "Fault Based Cryptanalysis of the Advanced Encryption Standard (AES)." In: *Financial Cryptography – FC*. 2003, pp. 162–181. DOI: 10.1007/978-3-540-45126-6_12.

[Blö+14] Johannes Blömer, Ricardo Gomes da Silva, Peter Günther, Juliane Krämer, and Jean-Pierre Seifert. "A Practical Second-Order Fault Attack against a Real-World Pairing Implementation." In: *Fault Diagnosis and Tolerance in Cryptography – FDTC*. 2014, pp. 123–136. DOI: 10.1109/FDTC.2014.22.

[Bol+13] David Bol, Julien De Vos, Cédric Hocquet, François Botman, François Durvaux, Sarah Boyd, Denis Flandre, and Jean-Didier Legat. "SleepWalker: A 25-MHz 0.4-V Sub-mm^2 7-μW/MHz Microcontroller in 65-nm LP/GP CMOS for Low-Carbon Wireless Sensor Nodes." In: *IEEE J. Solid State Circuits* 48 (2013), pp. 20–32. DOI: 10.1109/JSSC.2012.2218067.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. "On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract)." In: *Advances in Cryptology – EUROCRYPT*. 1997, pp. 37–51. DOI: 10.1007/3-540-69053-0_4.

[Bor+12] Julia Borghoff et al. "PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract." In: *Advances in Cryptology – ASIACRYPT*. 2012, pp. 208–225. DOI: 10.1007/978-3-642-34961-4_14.

[Bor23] Pietro Borrello. *x86 PAC*. https://github.com/pietroborrello/CustomProcessingUnit/commit/936a68492ce17bea1dd6a86fdb81a1bb06661d84. [accessed 2023-01-01]. 2023.

[Bor+] Pietro Borrello, Catherine Easdon, Martin Schwarzl, Roland Czerny, and Michael Schwarz. "CustomProcessingUnit: Reverse Engineering and Customization of Intel Microcode." In: *IEEE Workshop on Offensive Technologies (WOOT 23)*.

[BH08] Arnaud Boscher and Helena Handschuh. "Masking Does Not Protect Against Differential Fault Attacks." In: *Fault Diagnosis and Tolerance in Cryptography – FDTC*. 2008, pp. 35–40. DOI: 10.1109/FDTC.2008.12.

[Bou+12] Kaouthar Bousselam, Giorgio Di Natale, Marie-Lise Flottes, and Bruno Rouzeyre. “On Countermeasures Against Fault Attacks on the Advanced Encryption Standard.” In: *Fault Analysis in Cryptography*. 2012. DOI: 10.1007/978-3-642-29656-7_6.

[BFP19] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. “Shaping the Glitch: Optimizing Voltage Fault Injection Attacks.” In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* (2019), pp. 199–224. DOI: 10.13154/tches.v2019.i2.199-224.

[Bro60] David T. Brown. “Error Detecting and Correcting Binary Codes for Arithmetic Operations.” In: *IRE Trans. Electron. Comput.* 9 (1960), pp. 333–337. DOI: 10.1109/TEC.1960.5219855.

[Bue19] Davidlohr Bueso. *tools/perf-bench: Add basic syscall benchmark*. https://lore.kernel.org/patchwork/patch/1048777. [accessed 2023-01-01]. 2019.

[Buh+21] Robert Buhren, Hans Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. “One Glitch to Rule Them All: Fault Injection Attacks Against AMD’s Secure Encrypted Virtualization.” In: *Conference on Computer and Communications Security – CCS*. 2021, pp. 2875–2889. DOI: 10.1145/3460120.3484779.

[Can+22] Claudio Canella, Sebastian Dorn, Daniel Gruss, and Michael Schwarz. “SFIP: Coarse-Grained Syscall-Flow-Integrity Protection in Modern Systems.” In: *CoRR* abs/2202.13716 (2022). URL: https://arxiv.org/abs/2202.13716.

[Can+21] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. “Automating Seccomp Filter Generation for Linux Applications.” In: *Cloud Computing Security Workshop – CCSW*. 2021, pp. 139–151. DOI: 10.1145/3474123.3486762.

[Car+19] Sébastien Carré, Matthieu Desjardins, Adrien Facon, and Sylvain Guilley. “Exhaustive single bit fault analysis. A use case against Mbedtls and OpenSSL’s protection on ARM and Intel CPU.” In: *Microprocess. Microsystems* 71 (2019). DOI: 10.1016/j.micpro.2019.102860.

[CB17] Lukas Cavigelli and Luca Benini. “Origami: A 803-GOp/s/W Convolutional Network Accelerator.” In: *IEEE Trans. Circuits Syst. Video Technol.* 27 (2017), pp. 2461–2475. DOI: 10.1109/TCSVT.2016.2592330.

[CMB15] Lukas Cavigelli, Michele Magno, and Luca Benini. “Accelerating real-time embedded scene labeling with convolutional networks.” In: *Design Automation Conference – DAC*. 2015, 108:1–108:6. DOI: 10.1145/2744769.2744788.

[CPA15] Christopher Celio, David A Patterson, and Krste Asanovic. "The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor." In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167* (2015).

[Che+10] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. "Return-oriented programming without returns." In: *Conference on Computer and Communications Security – CCS*. 2010, pp. 559–572. DOI: 10.1145/1866307.1866370.

[CY03] Chien-Ning Chen and Sung-Ming Yen. "Differential Fault Analysis on AES Key Schedule and Some Coutnermeasures." In: *Information Security and Privacy – ACISP*. 2003, pp. 118–129. DOI: 10.1007/3-540-45067-X_11.

[Che+16] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. "14.5 Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks." In: *International Solid-State Circuits Conference – ISSCC*. 2016, pp. 262–263. DOI: 10.1109/ISSCC.2016.7418007.

[Che18] Zhi Chen. "SIMD Assisted Fault Detection and Fault Attack Mitigation." PhD thesis. University of California, Irvine, USA, 2018.

[Che+17] Zhi Chen, Junjie Shen, Alex Nicolau, Alexander V. Veidenbaum, Nahid Farhady Ghalaty, and Rosario Cammarota. "CAMFAS: A Compiler Approach to Mitigate Fault Attacks via Enhanced SIMDization." In: *Fault Diagnosis and Tolerance in Cryptography – FDTC*. 2017, pp. 57–64. DOI: 10.1109/FDTC.2017.10.

[CO23] Zitai Chen and David F. Oswald. "PMFault: Faulting and Bricking Server CPUs through Management Interfaces." In: *CoRR* abs/2301.05538 (2023). DOI: 10.48550/arXiv.2301.05538.

[Che+21] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David F. Oswald, and Flavio D. Garcia. "VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface." In: *USENIX Security Symposium*. 2021, pp. 699–716. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/chen-zitai.

[Cle+17] Ruan de Clercq, Johannes Götzfried, David Übler, Pieter Maene, and Ingrid Verbauwhede. "SOFIA: Software and control flow integrity architecture." In: *Comput. Secur.* 68 (2017), pp. 16–35. DOI: 10.1016/j.cose.2017.03.013.

[Cle+16] Ruan de Clercq, Ronald De Keulenaer, Bart Coppens, Bohan Yang, Pieter Maene, Koen De Bosschere, Bart Preneel, Bjorn De Sutter, and Ingrid Verbauwhede. "SOFIA: Software and control flow integrity architecture." In: *Design, Automation & Test in Europe – DATE*. 2016, pp. 1172–1177. URL: https://ieeexplore.ieee.org/document/7459489/.

[Con+17a] Francesco Conti, Daniele Palossi, Renzo Andri, Michele Magno, and Luca Benini. "Accelerated Visual Context Classification on a Low-Power Smartwatch." In: *IEEE Trans. Hum. Mach. Syst.* 47 (2017), pp. 19–30. DOI: 10.1109/THMS.2016.2623482.

[Con+14] Francesco Conti, Chuck Pilkington, Andrea Marongiu, and Luca Benini. "He-P2012: architectural heterogeneity exploration on a scalable many-core platform." In: *Great Lakes Symposium on VLSI – GLVLSI*. 2014, pp. 231–232. DOI: 10.1145/2591513.2591553.

[CFC] Thomas Coudray, Arnaud Fontaine, and Pierre Chifflier. "PICON: control flow integrity on LLVM IR." In: *Symposium sur la sécurité des technologies de l'information et des communications, Rennes, France*, pp. 3–5.

[CDA14] John Criswell, Nathan Dautenhahn, and Vikram S. Adve. "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels." In: *IEEE Symposium on Security and Privacy – S&P*. 2014, pp. 292–307. DOI: 10.1109/SP.2014.26.

[CH17] Ang Cui and Rick Housley. "BADFET: Defeating Modern Secure Boot Using Second-Order Pulsed Electromagnetic Fault Injection." In: *Workshop on Offensive Technologies – WOOT*. 2017. URL: https://www.usenix.org/conference/woot17/workshop-program/presentation/cui.

[Dah+12] George E. Dahl, Dong Yu, Li Deng, and Alex Acero. "Context-Dependent Pre-Trained Deep Neural Networks for Large-Vocabulary Speech Recognition." In: *IEEE Trans. Speech Audio Process.* 20 (2012), pp. 30–42. DOI: 10.1109/TASL.2011.2134090.

[Din+19] Daniel Dinu, Yann Le Corre, Dmitry Khovratovich, Léo Perrin, Johann Großschädl, and Alex Biryukov. "Triathlon of lightweight block ciphers for the Internet of things." In: *J. Cryptogr. Eng.* 9 (2019), pp. 283–302. DOI: 10.1007/s13389-018-0193-x.

[Dob+18] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. "SIFA: Exploiting Ineffective Fault Inductions on Symmetric Cryptography." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* (2018), pp. 547–572. DOI: 10.13154/tches.v2018.i3.547-572.

[Dob+14] Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, and Florian Mendel. "On the Security of Fresh Re-keying to Counteract Side-Channel and Fault Attacks." In: *Smart Card Research and Advanced Applications – CARDIS*. 2014, pp. 233–244. DOI: 10.1007/978-3-319-16763-3_14.

[Dob+20] Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, Florian Mendel, Bart Mennink, Robert Primas, and Thomas Unterluggauer. "Isap v2.0." In: *IACR Trans. Symmetric Cryptol.* (2020), pp. 390–416. DOI: 10.13154/tosc.v2020.iS1.390-416.

[Dob+16] Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, Florian Mendel, and Thomas Unterluggauer. "ISAP - Authenticated Encryption Inherently Secure Against Passive Side-Channel Attacks." In: *IACR Cryptol. ePrint Arch.* (2016), p. 952. URL: http://eprint.iacr.org/2016/952.

[Dob+17] Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, Florian Mendel, and Thomas Unterluggauer. "ISAP - Towards Side-Channel Secure Authenticated Encryption." In: *IACR Trans. Symmetric Cryptol.* (2017), pp. 80–105. DOI: 10.13154/tosc.v2017.i1.80-105.

[Dob+15] Christoph Dobraunig, François Koeune, Stefan Mangard, Florian Mendel, and François-Xavier Standaert. "Towards Fresh and Hybrid Re-Keying Schemes with Beyond Birthday Security." In: *Smart Card Research and Advanced Applications – CARDIS*. 2015, pp. 225–241. DOI: 10.1007/978-3-319-31271-2_14.

[Du+15] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. "ShiDianNao: shifting vision processing closer to the sensor." In: *International Symposium on Computer Architecture – ISCA*. 2015, pp. 92–104. DOI: 10.1145/2749469.2750389.

[Dwo+10] Morris J Dworkin et al. *Recommendation for block cipher modes of operation: The XTS-AES mode for confidentiality on storage devices*. National Inst of Standards and Technology Gaithersburg MD Computer security Div, 2010.

[EEM] EEMBC. *CoreMark: An EEMBC Benchmark*. https://www.eembc.org/coremark/. [accessed 2023-01-01].

[EKE08] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. "Verified Protection Model of the seL4 Microkernel." In: *International Conference on Verified Software: Theories, Tools, and Experiments – VSTTE*. 2008, pp. 99–114. DOI: 10.1007/978-3-540-87873-5_11.

[Eva+15] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard E. Shrobe, Stelios Sidiroglou-Douskos, Martin C. Rinard, and Hamed Okhravi. "Missing the Point(er): On the Effectiveness of Code Pointer Integrity." In: *IEEE Symposium on*

Security and Privacy – S&P. 2015, pp. 781–796. DOI: 10.1109/SP.2015.53.

[Fan+22] Andrea Fanti, Carlos Chinea Perez, Rémi Denis-Courmont, Gianluca Roascio, and Jan-Erik Ekberg. "Toward Register Spilling Security Using LLVM and ARM Pointer Authentication." In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 41 (2022), pp. 3757–3766. DOI: 10.1109/TCAD.2022.3197511.

[FPS12] Sebastian Faust, Krzysztof Pietrzak, and Joachim Schipper. "Practical Leakage-Resilient Symmetric Cryptography." In: *Cryptographic Hardware and Embedded Systems – CHES*. 2012, pp. 213–232. DOI: 10.1007/978-3-642-33027-8_13.

[Fel22] Rich Felker. *musl libc*. https://musl.libc.org. [accessed 2023-01-01]. 2022.

[FSS09] Christof Fetzer, Ute Schiffel, and Martin Süßkraut. "AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware." In: *Computer Safety, Reliability and Security – SAFECOMP*. 2009, pp. 283–296. DOI: 10.1007/978-3-642-04468-7_23.

[For] P. Forin. "Vital coded microprocessor: Principles and application for various transit systems." In: *IFAC/IFIP/IFORS Symposium on Control, Computers, Communications in Transportation, Paris, France, 19-21 September*, pp. 79–84. DOI: https://doi.org/10.1016/S1474-6670(17)52653-1.

[Fre20] Free Software Foundation Inc. *GCC 7 Release Series Changes, New Features, and Fixes*. https://gcc.gnu.org/gcc-7/changes.html. [accessed 2023-01-01]. 2020.

[Fre11] Free60.org. *Reset Glitch Hack*. https://free60.org/Hacks/Reset_Glitch_Hack. [accessed 2023-01-01]. 2011.

[Fuh+13] Thomas Fuhr, Éliane Jaulmes, Victor Lomné, and Adrian Thillard. "Fault Attacks on AES with Faulty Ciphertexts Only." In: *Fault Diagnosis and Tolerance in Cryptography – FDTC*. 2013, pp. 108–118. DOI: 10.1109/FDTC.2013.18.

[Gau+17] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices." In: *IEEE Trans. Very Large Scale Integr. Syst.* 25 (2017), pp. 2700–2713. DOI: 10.1109/TVLSI.2017.2654506.

[GCC23] GCC Team. *GCC - [AArch64][1/4] Support Return address protection on AArch64*. https://gcc.gnu.org/git/?p=gcc.git;a=commit;h=db58fd8954f5dfd868dbed110f2c8a04bb4b0753. [accessed 2023-01-01]. 2023.

[Ge+16] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. "Fine-Grained Control-Flow Integrity for Kernel Software." In: *European Symposium on Security and Privacy – EuroS&P*. 2016, pp. 179–194. DOI: 10.1109/EuroSP.2016.24.

[Gha+14] Nahid Farhady Ghalaty, Bilgiday Yuce, Mostafa M. I. Taha, and Patrick Schaumont. "Differential Fault Intensity Analysis." In: *Fault Diagnosis and Tolerance in Cryptography – FDTC*. 2014, pp. 49–58. DOI: 10.1109/FDTC.2014.15.

[Ghi21] Alexandre Ghiti. *Virtual Memory Layout on RISC-V Linux*. https://www.kernel.org/doc/html/latest/riscv/vm-layout.html. [accessed 2023-01-01]. 2021.

[Gir+14] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation." In: *Computer Vision and Pattern Recognition Conference – CVPR*. 2014, pp. 580–587. DOI: 10.1109/CVPR.2014.81.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. "How to construct random functions." In: *J. ACM* 33 (1986), pp. 792–807. DOI: 10.1145/6490.6503.

[Gol+03] Olga Goloubeva, Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. "Soft-Error Detection Using Control Flow Assertions." In: *International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems – DFT*. 2003, pp. 581–588. DOI: 10.1109/DFTVS.2003.1250158.

[Goo15] Google Project Zero. *Exploiting the DRAM rowhammer bug to gain kernel privileges*. https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html. [accessed 2023-01-01]. 2015.

[GP99] Louis Goubin and Jacques Patarin. "DES and Differential Power Analysis (The "Duplication" Method)." In: *Cryptographic Hardware and Embedded Systems – CHES*. 1999, pp. 158–172. DOI: 10.1007/3-540-48059-5_15.

[Gru+18] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. "Another Flip in the Wall of Rowhammer Defenses." In: *IEEE Symposium on Security and Privacy – S&P*. 2018, pp. 245–261. DOI: 10.1109/SP.2018.00031.

[GMM16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript." In: *Detection of Intrusions and Malware & Vulnerability Assessment – DIMVA*. 2016, pp. 300–321. DOI: 10.1007/978-3-319-40667-1_15.

[Gue10] Shay Gueron. "Intel advanced encryption standard (AES) instructions set." In: *Intel White Paper, Rev* 3 (2010), pp. 1–94.

[Gue13] Shay Gueron. “AES-GCM software performance on the current high end CPUs as a performance baseline for CAESAR competition.” In: *Directions in Authenticated Ciphers (DIAC)* (2013).

[GJ16] Qian Guo and Thomas Johansson. “A New Birthday-Type Algorithm for Attacking the Fresh Re-Keying Countermeasure.” In: *IACR Cryptol. ePrint Arch.* (2016), p. 225. URL: http://eprint.iacr.org/2016/225.

[Ham50] Richard W Hamming. “Error detecting and error correcting codes.” In: *Bell Labs Technical Journal* (1950).

[He+16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Deep Residual Learning for Image Recognition.” In: *Computer Vision and Pattern Recognition Conference – CVPR*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[HBB16] Wei He, Jakub Breier, and Shivam Bhasin. “Cheap and Cheerful: A Low-Cost Digital Sensor for Detecting Laser Fault Injection Attacks.” In: *International Conference on Security, Privacy, and Applied Cryptography Engineering – SPACE*. 2016, pp. 27–46. DOI: 10.1007/978-3-319-49445-6_2.

[Her+21] Jan Van den Herrewegen, David F. Oswald, Flavio D. Garcia, and Qais Temeiza. “Fill your Boots: Enhanced Embedded Bootloader Exploits via Fault Injection and Binary Analysis.” In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* (2021), pp. 56–81. DOI: 10.46586/tches.v2021.i1.56-81.

[HLB19] Karine Heydemann, Jean-François Lalande, and Pascal Berthomé. “Formally verified software countermeasures for control-flow integrity of smart card C code.” In: *Comput. Secur.* 85 (2019), pp. 202–224. DOI: 10.1016/j.cose.2019.05.004.

[HKR15] Viet Tung Hoang, Ted Krovetz, and Phillip Rogaway. “Robust Authenticated-Encryption AEZ and the Problem That It Solves.” In: *Advances in Cryptology – EUROCRYPT*. 2015, pp. 15–44. DOI: 10.1007/978-3-662-46800-5_2.

[Hoc+11] Cédric Hocquet, Dina Kamel, Francesco Regazzoni, Jean-Didier Legat, Denis Flandre, David Bol, and François-Xavier Standaert. “Harvesting the potential of nano-CMOS for lightweight cryptography: an ultra-low-voltage 65 nm AES coprocessor for passive RFID tags.” In: *J. Cryptogr. Eng.* 1 (2011), pp. 79–86. DOI: 10.1007/s13389-011-0005-z.

[Hof+14] Martin Hoffmann, Peter Ulbrich, Christian Dietrich, Horst Schirmeier, Daniel Lohmann, and Wolfgang Schröder-Preikschat. “A Practitioner’s Guide to Software-Based Soft-Error Mitigation Using AN-Codes.” In: *International Symposium on High-Assurance Systems Engineering – HASE*. 2014, pp. 33–40. DOI: 10.1109/HASE.2014.14.

[HZH22] Konrad Hohentanner, Philipp Zieris, and Julian Horsch. "PACSafe: Leveraging ARM Pointer Authentication for Memory Safety in C/C++." In: *CoRR* abs/2202.08669 (2022). URL: https://arxiv.org/abs/2202.08669.

[Hu+16] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks." In: *IEEE Symposium on Security and Privacy – S&P*. 2016, pp. 969–986. DOI: 10.1109/SP.2016.62.

[Hu+05] Jie S. Hu, Feihui Li, Vijay Degalahal, Mahmut T. Kandemir, Narayanan Vijaykrishnan, and Mary Jane Irwin. "Compiler-Directed Instruction Duplication for Soft Error Detection." In: *Design, Automation & Test in Europe – DATE*. 2005, pp. 1056–1057. DOI: 10.1109/DATE.2005.98.

[HHF09] Ralf Hund, Thorsten Holz, and Felix C. Freiling. "Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms." In: *USENIX Security Symposium*. 2009, pp. 383–398. URL: http://www.usenix.org/events/sec09/tech/full_papers/hund.pdf.

[HS13] Michael Hutter and Jörn-Marc Schmidt. "The Temperature Side Channel and Heating Fault Attacks." In: *Smart Card Research and Advanced Applications – CARDIS*. 2013, pp. 219–235. DOI: 10.1007/978-3-319-08302-5_15.

[Inf23] Infineon Technologies AG. *AURIX™ 32-bit microcontrollers for automotive and industrial applications*. https://www.infineon.com/dgdl/Infineon-TriCore_Family_BR-ProductBrochure-v01_00-EN.pdf. [accessed 2023-01-01]. 2023.

[Jal+20] Georges-Axel Jaloyan, Konstantinos Markantonakis, Raja Naeem Akram, David Robin, Keith Mayes, and David Naccache. "Return-Oriented Programming on RISC-V." In: *Conference on Computer and Communications Security – CCS*. 2020, pp. 471–480. DOI: 10.1145/3320269.3384738.

[Jon] Douglas W. Jones. *Modulus without Division, a tutorial*. http://homepage.divms.uiowa.edu/~jones/bcd/mod.shtml. [accessed 2023-01-01].

[JWK04] Nikhil Joshi, Kaijie Wu, and Ramesh Karri. "Concurrent Error Detection Schemes for Involution Ciphers." In: *Cryptographic Hardware and Embedded Systems – CHES*. 2004, pp. 400–412. DOI: 10.1007/978-3-540-28632-5_29.

[KSV13] Dusko Karaklajic, Jörn-Marc Schmidt, and Ingrid Verbauwhede. "Hardware Designer's Guide to Fault Attacks." In: *IEEE Trans. Very Large Scale Integr. Syst.* 21 (2013), pp. 2295–2306. DOI: 10.1109/TVLSI.2012.2231707.

[Kar+02] Ramesh Karri, Kaijie Wu, Piyush Mishra, and Yongkook Kim. "Concurrent error detection schemes for fault-based side-channel cryptanalysis of symmetric block ciphers." In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 21 (2002), pp. 1509–1517. DOI: 10.1109/TCAD.2002.804378.

[Ken+20] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. "V0LTpwn: Attacking x86 Processor Integrity from Software." In: *USENIX Security Symposium*. 2020, pp. 1445–1461. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/kenjar.

[Ker22a] Kernel Authors. *Linux Kernel*. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tag/?h=v5.15.32. [accessed 2023-01-01]. 2022.

[Ker22b] Michael Kerrisk. *syscall(2) — Linux manual page*. https://man7.org/linux/man-pages/man2/syscall.2.html. [accessed 2023-01-01]. 2022.

[Kha+12] Rafiullah Khan, Sarmad Ullah Khan, Rifaqat Zaheer, and Shahid Khan. "Future Internet: The Internet of Things Architecture, Possible Applications and Key Challenges." In: *International Conference on Frontiers of Information Technology – FIT*. 2012, pp. 257–260. DOI: 10.1109/FIT.2012.53.

[KQ07] Chong Hee Kim and Jean-Jacques Quisquater. "Fault Attacks for CRT Based RSA: New Attacks, New Results, and New Countermeasures." In: *Information Security Theory and Practice – WISTP*. 2007, pp. 215–228. DOI: 10.1007/978-3-540-72354-7_18.

[Kim+14] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors." In: *International Symposium on Computer Architecture – ISCA*. 2014, pp. 361–372. DOI: 10.1109/ISCA.2014.6853210.

[Kis+16] Ágnes Kiss, Juliane Krämer, Pablo Rauzy, and Jean-Pierre Seifert. "Algorithmic Countermeasures Against Fault Attacks and Power Analysis for RSA-CRT." In: *Constructive Side-Channel Analysis and Secure Design – COSADE*. 2016, pp. 111–129. DOI: 10.1007/978-3-319-43283-0_7.

[Kle+09] Gerwin Klein et al. "seL4: formal verification of an OS kernel." In: *Workshop on System Software for Trusted Execution – SysTEX@SOSP*. 2009, pp. 207–220. DOI: 10.1145/1629575.1629596.

[Kle+10] Gerwin Klein et al. "seL4: formal verification of an operating-system kernel." In: *Commun. ACM* 53 (2010), pp. 107–115. DOI: 10.1145/1743546.1743574.

[Koc03] Paul C. Kocher. “Leak-resistant cryptographic indexed key update.” US Patent 6539092. Mar. 25, 2003.

[Kog+22] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. “Half-Double: Hammering From the Next Row Over.” In: *USENIX Security Symposium*. 2022, pp. 3807–3824. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/kogler-half-double.

[Kon+16] Mario Konijnenburg et al. “A Multi(bio)sensor Acquisition System With Integrated Processor, Power Management, 8×8 LED Drivers, and Simultaneously Synchronized ECG, BIO-Z, GSR, and Two PPG Readouts.” In: *IEEE J. Solid State Circuits* 51 (2016), pp. 2584–2595. DOI: 10.1109/JSSC.2016.2605660.

[KH14] Thomas Korak and Michael Hoefler. “On the Effects of Clock and Power Supply Tampering on Two Microcontroller Platforms.” In: *Fault Diagnosis and Tolerance in Cryptography – FDTC*. 2014, pp. 8–17. DOI: 10.1109/FDTC.2014.11.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks.” In: *Conference and Workshop on Neural Information Processing Systems – NIPS*. 2012, pp. 1106–1114. URL: https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

[Lac+18] Benjamin Lac, Anne Canteaut, Jacques J. A. Fournier, and Renaud Sirdey. “Thwarting Fault Attacks against Lightweight Cryptography using SIMD Instructions.” In: *International Symposium on Circuits and Systems – ISCAS*. 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351693.

[LHB14] Jean-François Lalande, Karine Heydemann, and Pascal Berthomé. “Software Countermeasures for Control Flow Integrity of Smart Card C Codes.” In: *European Symposium on Research in Computer Security – ESORICS*. 2014, pp. 200–218. DOI: 10.1007/978-3-319-11212-1_12.

[LA04] Chris Lattner and Vikram S. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.” In: *International Symposium on Code Generation and Optimization – CGO*. 2004, pp. 75–88. DOI: 10.1109/CGO.2004.1281665.

[Lav78] Simon H. Lavington. “The Manchester Mark I and Atlas: A Historical Perspective.” In: *Commun. ACM* 21 (1978), pp. 4–12. DOI: 10.1145/359327.359331.

[Lee+17] Jae-Hyuk Lee, Jin Soo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. “Hacking in Darkness: Return-oriented Programming against Secure Enclaves.” In: *USENIX Security Symposium*. 2017, pp. 523–539. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-jaehyuk.

[LV13] Kyong-Ho Lee and Naveen Verma. “A Low-Power Processor With Configurable Embedded Machine-Learning Accelerators for High-Order and Adaptive Analysis of Medical-Sensor Signals.” In: *IEEE J. Solid State Circuits* 48 (2013), pp. 1625–1637. DOI: 10.1109/JSSC.2013.2253226.

[Li+15] Haoxiang Li, Zhe Lin, Xiaohui Shen, Jonathan Brandt, and Gang Hua. “A convolutional neural network cascade for face detection.” In: *Computer Vision and Pattern Recognition Conference – CVPR*. 2015, pp. 5325–5334. DOI: 10.1109/CVPR.2015.7299170.

[Li+10] Yang Li, Kazuo Sakiyama, Shigeto Gomisawa, Toshinori Fukunaga, Junko Takahashi, and Kazuo Ohta. “Fault Sensitivity Analysis.” In: *Cryptographic Hardware and Embedded Systems – CHES*. 2010, pp. 320–334. DOI: 10.1007/978-3-642-15031-9_22.

[Li+22] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. “PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication.” In: *Conference on Computer and Communications Security – CCS*. 2022, pp. 1901–1915. DOI: 10.1145/3548606.3560598.

[LG19] Haohao Liao and Catherine H. Gebotys. “Methodology for EM Fault Injection: Charge-based Fault Model.” In: *Design, Automation & Test in Europe – DATE*. 2019, pp. 256–259. DOI: 10.23919/DATE.2019.8715150.

[Lil+21] Hans Liljestrand, Thomas Nyman, Lachlan J. Gunn, Jan-Erik Ekberg, and N. Asokan. “PACStack: an Authenticated Call Stack.” In: *USENIX Security Symposium*. 2021, pp. 357–374. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/liljestrand.

[Lil+19] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. “PAC it up: Towards Pointer Integrity using ARM Pointer Authentication.” In: *USENIX Security Symposium*. 2019, pp. 177–194. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand.

[Lin23] Linaro Limited. *Mbed TLS*. https://www.trustedfirmware.org/projects/mbed-tls/. [accessed 2023-01-01]. 2023.

[Lip+20] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. "Nethammer: Inducing Rowhammer Faults through Network Requests." In: *European Symposium on Security and Privacy – EuroS&P*. 2020, pp. 710–719. DOI: 10.1109/EuroSPW51379.2020.00102.

[LLV19] LLVM Team. *Pointer Authentication*. https://github.com/apple/llvm-project/blob/apple/master/clang/docs/PointerAuthentication.rst. [accessed 2023-01-01]. 2019.

[LLV23] LLVM Team. *LLVM - [AArch64] - Generate pointer authentication instructions*. https://github.com/llvm/llvm-project/commit/64dcdec60cdc3612cc5cf14dea6599b91a5f94bd. [accessed 2023-01-01]. 2023.

[Lv+18] ShaoHua Lv, Jian Wang, Yinqi Yang, and Jiqiang Liu. "Intrusion Prediction With System-Call Sequence-to-Sequence Model." In: *IEEE Access* 6 (2018), pp. 71413–71421. DOI: 10.1109/ACCESS.2018.2881561.

[MSY06] Tal Malkin, François-Xavier Standaert, and Moti Yung. "A Comparative Cost/Security Analysis of Fault Attack Countermeasures." In: *Fault Diagnosis and Tolerance in Cryptography – FDTC*. 2006, pp. 159–172. DOI: 10.1007/11889700_15.

[MIR04] Venkateswara Sarma Mallela, V Ilankumaran, and N.Srinivasa Rao. "Trends in Cardiac Pacemaker Batteries." In: *Indian Pacing and Electrophysiology Journal* 4 (2004), pp. 201–212.

[Mar20] Catalin Marinas. *Memory Layout on AArch64 Linux*. https://www.kernel.org/doc/html/latest/arm64/memory.html. [accessed 2023-01-01]. 2020.

[Mas+15] Ali José Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. "CCFI: Cryptographically Enforced Control Flow Integrity." In: *Conference on Computer and Communications Security – CCS*. 2015, pp. 941–951. DOI: 10.1145/2810103.2813676.

[Mas64] James L Massey. "Survey of residue coding for arithmetic errors." In: *International Computation Center Bulletin* 3 (1964), pp. 3–17.

[Mat+15] Sanu Mathew, Sudhir Satpathy, Vikram B. Suresh, Mark A. Anders, Himanshu Kaul, Amit Agarwal, Steven Hsu, Gregory K. Chen, and Ram Krishnamurthy. "340 mV-1.1 V, 289 Gbps/W, 2090-Gate NanoAES Hardware Accelerator With Area-Optimized Encrypt/Decrypt GF(2 4) 2 Polynomials in 22 nm Tri-Gate CMOS." In: *IEEE J. Solid State Circuits* 50 (2015), pp. 1048–1058. DOI: 10.1109/JSSC.2014.2384039.

[Mat+14] Sanu Mathew, Sudhir Satpathy, Vikram B. Suresh, Himanshu Kaul, Mark A. Anders, Gregory K. Chen, Amit Agarwal, Steven Hsu, and Ram Krishnamurthy. "340mV-1.1V, 289Gbps/W, 2090-gate NanoAES hardware accelerator with area-optimized encrypt/decrypt GF$(2^4)^2$ polynomials in 22nm tri-gate CMOS." In: *Symposium on VLSI Circuits – VLSIC*. 2014, pp. 1–2. DOI: 10.1109/VLSIC.2014.6858420.

[Max17] Maxim Integrated, Analog Devices. *Maxim Integrated MAXQ1061 DeepCover Cryptographic Controller for Embedded Devices*. https://www.analog.com/en/products/maxq1061.html. [accessed 2023-01-01]. 2017.

[MV04] David A. McGrew and John Viega. "The Security and Performance of the Galois/Counter Mode (GCM) of Operation." In: *Progress in Cryptology – INDOCRYPT*. 2004, pp. 343–355. DOI: 10.1007/978-3-540-30556-9_27.

[MM11] Marcel Medwed and Stefan Mangard. "Arithmetic logic units with high error detection rates to counteract fault attacks." In: *Design, Automation & Test in Europe – DATE*. 2011, pp. 1644–1649. DOI: 10.1109/DATE.2011.5763261.

[Med+11] Marcel Medwed, Christophe Petit, Francesco Regazzoni, Mathieu Renauld, and François-Xavier Standaert. "Fresh Re-keying II: Securing Multiple Parties against Side-Channel and Fault Attacks." In: *Smart Card Research and Advanced Applications – CARDIS*. 2011, pp. 115–132. DOI: 10.1007/978-3-642-27257-8_8.

[MS09] Marcel Medwed and Jörn-Marc Schmidt. "Coding Schemes for Arithmetic and Logic Operations - How Robust Are They?" In: *Information Security Applications – WISA*. 2009, pp. 51–65. DOI: 10.1007/978-3-642-10838-9_5.

[Med+10] Marcel Medwed, François-Xavier Standaert, Johann Großschädl, and Francesco Regazzoni. "Fresh Re-keying: Security against Side-Channel and Fault Attacks for Low-Cost Devices." In: *Progress in Cryptology – AFRICACRYPT*. 2010, pp. 279–296. DOI: 10.1007/978-3-642-12678-9_17.

[Mor+13] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. "Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller." In: *Fault Diagnosis and Tolerance in Cryptography – FDTC*. 2013, pp. 77–88. DOI: 10.1109/FDTC.2013.9.

[Mur+20] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. "Plundervolt: Software-based Fault Injection Attacks against Intel SGX." In: *IEEE Symposium on Security and Privacy – S&P*. 2020, pp. 1466–1482. DOI: 10.1109/SP40000.2020.00057.

[Mut+22] Md Rafid Muttaki, Tao Zhang, Mark M. Tehranipoor, and Farimah Farahmandi. "FTC: A Universal Sensor for Fault Injection Attack Detection." In: *IEEE International Symposium on Hardware Oriented Security and Trust – HOST*. 2022, pp. 117–120. DOI: 10.1109/HOST54066.2022.9840177.

[Mye+15] James Myers, Anand Savanth, David Howard, Rohan Gaddh, Pranay Prabhat, and David Flynn. "8.1 An 80nW retention 11.7pJ/cycle active subthreshold ARM Cortex-M0+ subsystem in 65nm CMOS for WSN applications." In: *International Solid-State Circuits Conference – ISSCC*. 2015, pp. 1–3. DOI: 10.1109/ISSCC.2015.7062967.

[Nak+16] Masami Nakajima, Ichiro Naka, Fumihiro Matsushima, and Tadaaki Yamauchi. "A 20uA/MHz at 200MHz microcontroller with low power memory access scheme for small sensing nodes." In: *Symposium on Low-Power and High-Speed Chips and Systems – COOL CHIPS*. 2016, pp. 1–3. DOI: 10.1109/CoolChips.2016.7503677.

[NT] Pascal Nasahl and Niek Timmers. "Attacking AUTOSAR using Software and Hardware Attacks." In: *escar USA*.

[NCC] NCC Group. *There's A Hole In Your SoC: Glitching The MediaTek BootROM*. https://research.nccgroup.com/2020/10/15/theres-a-hole-in-your-soc-glitching-the-mediatek-bootrom. [accessed 2023-01-01].

[New23] NewAE Technology Inc. *ChipSHOUTER®*. https://www.newae.com/chipshouter. [accessed 2023-01-01]. 2023.

[NIS01] NIST FIPS. "Advanced Encryption Standard (AES)." In: *Federal Information Processing Standards Publication* 197 (2001), pp. 441–0311.

[NXP22] NXP Semiconductors. *LPC54000 Series: Low Power Microcontollers (MCUs) based on ARM® Cortex-M4 Cores with optional Cortex-M0+ co-processor*. https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc54000-arm-cortex-m4-:MC_1414576688124. [accessed 2023-01-01]. 2022. URL: www.nxp.com/LPC54000.

[OFl20] Colin O'Flynn. "BAM BAM!! On Reliability of EMFI for in-situ Automotive ECU Attacks." In: *IACR Cryptol. ePrint Arch.* (2020), p. 937. URL: https://eprint.iacr.org/2020/937.

[OSM02] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. "Control-flow checking by software signatures." In: *IEEE Trans. Reliab.* 51 (2002), pp. 111–122. DOI: 10.1109/24.994926.

[Ope23a] OpenHW Group. *OpenHW Group CORE-V CV32E40P RISC-V IP*. https://github.com/openhwgroup/cv32e40p. [accessed 2023-01-01]. 2023.

[OPE12] OPENRISC.IO. *OpenRISC 1000 Architecture Manual*. https://raw.githubusercontent.com/openrisc/doc/master/openrisc-arch-1.4-rev0.pdf. [accessed 2023-01-01]. 2012.

[Ope23b] OpenTitan. *Introduction to OpenTitan*. https://opentitan.org/documentation/index.html. [accessed 2023-01-01]. 2023.

[Oro+22] Lois Orosa, Ulrich Rührmair, Abdullah Giray Yaglikçi, Haocong Luo, Ataberk Olgun, Patrick Jattke, Minesh Patel, Jeremie S. Kim, Kaveh Razavi, and Onur Mutlu. "SpyHammer: Using RowHammer to Remotely Spy on Temperature." In: *CoRR* abs/2210.04084 (2022). DOI: 10.48550/arXiv.2210.04084.

[PC18] Ramiro Pareja and Santiago Cordoba. *Fault injection on automotive diagnostic protocols*. https://www.riscure.com/publication/fault-injection-automotive-diagnostic-protocols. [accessed 2023-01-01]. 2018.

[Par+15] Seongwook Park, Junyoung Park, Kyeongryeol Bong, Dongjoo Shin, Jinmook Lee, Sungpill Choi, and Hoi-Jun Yoo. "An Energy-Efficient and Scalable Deep Learning/Inference Processor With Tetra-Parallel MIMD Architecture for Big Data Applications." In: *IEEE Trans. Biomed. Circuits Syst.* 9 (2015), pp. 838–848. DOI: 10.1109/TBCAS.2015.2504563.

[Pat+] David Patterson, Jeremy Bennett, Cesare Garlati Palmer Dabbelt, G. S. Madhusudan, and Trevor Mudge. *Embench™: A Modern Embedded Benchmark Suite*. https://www.embench.org. [accessed 2023-01-01].

[Pau+16] Somnath Paul, Vinayak Honkote, Ryan Gary Kim, Turbo Majumder, Paolo A. Aseron, Vaughn Grossnickle, Robert Sankman, Debendra Mallik, Sandeep Jain, Sriram R. Vangal, James W. Tschanz, and Vivek De. "An energy harvesting wireless sensor node for IoT systems featuring a near-threshold voltage IA-32 microcontroller in 14nm tri-gate CMOS." In: *Symposium on VLSI Circuits – VLSIC*. 2016, pp. 1–2. DOI: 10.1109/VLSIC.2016.7573485.

[PB09] Andreas Persson and Lars Bengtsson. "Forward and Reverse Converters and Moduli Set Selection in Signed-Digit Residue Number Systems." In: *J. Signal Process. Syst.* 56 (2009), pp. 1–15. DOI: 10.1007/s11265-008-0249-8.

[PM16] Peter Pessl and Stefan Mangard. "Enhancing Side-Channel Analysis of Binary-Field Multiplication with Bit Reliability." In: *Topics in Cryptology – CT-RSA*. 2016, pp. 255–270. DOI: 10.1007/978-3-319-29485-8_15.

[Pet] William W. Peterson. *Error-correcting codes*. M.I.T. Press [u.a.]

[Pie09] Krzysztof Pietrzak. "A Leakage-Resilient Mode of Operation." In: *Advances in Cryptology – EUROCRYPT*. 2009, pp. 462–482. DOI: 10.1007/978-3-642-01001-9_27.

[PQ03] Gilles Piret and Jean-Jacques Quisquater. "A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD." In: *Cryptographic Hardware and Embedded Systems – CHES*. 2003, pp. 77–88. DOI: 10.1007/978-3-540-45238-6_7.

[Pul+16] Antonio Pullini, Francesco Conti, Davide Rossi, Igor Loi, Michael Gautschi, and Luca Benini. "A heterogeneous multi-core system-on-chip for energy efficient brain inspired vision." In: *International Symposium on Circuits and Systems – ISCAS*. 2016, p. 2910. DOI: 10.1109/ISCAS.2016.7539213.

[PULa] PULP Team. *PULPino: An open-source single-core microcontroller system*. https://github.com/pulp-platform/pulpino. [accessed 2023-01-01].

[PULb] PULP Team & OpenHW Group. *CV32E40P RISC-V IP*. https://github.com/openhwgroup/cv32e40p. [accessed 2023-01-01].

[QEM20] QEMU. *QEMU the FAST! processor emulator*. https://www.qemu.org. [accessed 2023-01-01]. 2020.

[Qiu+19a] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. "VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies." In: *Conference on Computer and Communications Security – CCS*. 2019, pp. 195–209. DOI: 10.1145/3319535.3354201.

[Qiu+19b] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. "VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults." In: *Asian Hardware Oriented Security and Trust Symposium – AsianHOST*. 2019, pp. 1–6. DOI: 10.1109/AsianHOST47458.2019.9006701.

[Qiu+20] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. "VoltJockey: Abusing the Processor Voltage to Break Arm TrustZone." In: *GetMobile Mob. Comput. Commun.* 24 (2020), pp. 30–33. DOI: 10.1145/3427384.3427394.

[Qua17] Qualcomm Technologies Inc. *Pointer Authentication on ARMv8.3*. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf. [accessed 2023-01-01]. 2017.

[Rah+11] Abbas Rahimi, Igor Loi, Mohammad Reza Kakoee, and Luca Benini. "A fully-synthesizable single-cycle interconnection network for Shared-L1 processor clusters." In: *Design, Automation & Test in Europe – DATE*. 2011, pp. 491–496. DOI: 10.1109/DATE.2011.5763085.

[Rao70] Thammavarapu R. N. Rao. "Biresidue Error-Correcting Codes for Computer Arithmetic." In: *IEEE Trans. Computers* 19 (1970), pp. 398–402. DOI: 10.1109/T-C.1970.222937.

[RG71] Thammavarapu R. N. Rao and Oscar N. Garcia. "Cyclic and multiresidue codes for arithmetic operations." In: *IEEE Trans. Inf. Theory* 17 (1971), pp. 85–91. DOI: 10.1109/TIT.1971.1054579.

[Ras20] Raspberry Pi Foundation. *Raspberry Pi 4 Model B*. https://www.raspberrypi.org/products/raspberry-pi-4-model-b. [accessed 2023-01-01]. 2020.

[RG14] Pablo Rauzy and Sylvain Guilley. "Countermeasures against High-Order Fault-Injection Attacks on CRT-RSA." In: *Fault Diagnosis and Tolerance in Cryptography – FDTC*. 2014, pp. 68–82. DOI: 10.1109/FDTC.2014.17.

[Rea15] RealTimeLogic. *SharkSSL/RayCrypto v2.4 crypto library benchmarks with ARM Cortex-M3*. https://realtimelogic.com/products/sharkssl/Cortex-M3. [accessed 2023-01-01]. 2015.

[Rei+05] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. "SWIFT: Software Implemented Fault Tolerance." In: *International Symposium on Code Generation and Optimization – CGO*. 2005, pp. 243–254. DOI: 10.1109/CGO.2005.34.

[RSG21] Jan Richter-Brockmann, Pascal Sasdrich, and Tim Güneysu. "Revisiting Fault Adversary Models - Hardware Faults in Theory and Practice." In: *IACR Cryptol. ePrint Arch.* (2021), p. 296. URL: https://eprint.iacr.org/2021/296.

[Ris17] Riscure. *Bypassing Secure Boot using Fault Injection*. https://www.blackhat.com/docs/eu-16/materials/eu-16-Timmers-Bypassing-Secure-Boot-Using-Fault-Injection.pdf. [accessed 2023-01-01]. 2017.

[Ris23a] Riscure. *Inspector Fault Injection*. https://www.riscure.com/security-tools/inspector-fi. [accessed 2023-01-01]. 2023.

[Ris23b] Riscure. *Secure Application Programming in the presence of Side Channel Attacks*. https://riscureprodstorage.blob.core.windows.net/production/2018/11/201708_Riscure_Whitepaper_Side_Channel_Patterns.pdf. [accessed 2023-01-01]. 2023.

[Rog04] Phillip Rogaway. "Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC." In: *Advances in Cryptology – ASIACRYPT*. 2004, pp. 16–31. DOI: 10.1007/978-3-540-30539-2_2.

[Ros+15] Davide Rossi, Francesco Conti, Andrea Marongiu, Antonio Pullini, Igor Loi, Michael Gautschi, Giuseppe Tagliavini, Alessandro Capotondi, Philippe Flatresse, and Luca Benini. "PULP: A parallel ultra low power platform for next generation IoT applications." In: *Hot Chips Symposium – HOT CHIPS*. 2015, pp. 1–39. DOI: 10.1109/HOTCHIPS.2015.7477325.

[Ros+14] Davide Rossi, Igor Loi, Germain Haugou, and Luca Benini. “Ultra-low-latency lightweight DMA for tightly coupled multi-core clusters.” In: *Computing Frontiers – CF*. 2014, 15:1–15:10. DOI: 10.1145/2597917.2597922.

[Ros+16] Davide Rossi et al. “193 MOPS/mW @ 162 MOPS, 0.32V to 1.15V voltage range multi-core accelerator for energy efficient parallel and sequential digital processing.” In: *Symposium on Low-Power and High-Speed Chips and Systems – COOL CHIPS*. 2016, pp. 1–3. DOI: 10.1109/CoolChips.2016.7503670.

[RND19] Thomas Roth, Dmitry Nedospasov, and Josh Datko. *Wallet.fail - Poof goes your crypto*. https://wallet.fail. [accessed 2023-01-01]. 2019.

[Rot12] Jeffrey Keith Rott. *Intel® Advanced Encryption Standard Instructions (AES-NI)*. https://software.intel.com/content/www/us/en/develop/articles/intel-advanced-encryption-standard-instructions-aes-ni.html. [accessed 2023-01-01. 2012.

[Roy+16] Abhishek Roy, Peter J. Grossmann, Steven A. Vitale, and Benton H. Calhoun. “A 1.3μW, 5pJ/cycle sub-threshold MSP430 processor in 90nm xLP FDSOI for energy-efficient IoT applications.” In: *International Symposium on Quality Electronic Design – ISQED*. 2016, pp. 158–162. DOI: 10.1109/ISQED.2016.7479193.

[Rut17] Mark Rutland. *Pointer authentication in AArch64 Linux*. https://www.kernel.org/doc/Documentation/arm64/pointer-authentication.rst. [accessed 2023-01-01]. 2017.

[SBM15] Santanu Sarkar, Subhadeep Banik, and Subhamoy Maitra. “Differential Fault Attack against Grain Family with Very Few Faults and Minimal Assumptions.” In: *IEEE Trans. Computers* 64 (2015), pp. 1647–1657. DOI: 10.1109/TC.2014.2339854.

[SMS23] Marvin Saß, Richard Mitev, and Ahmad-Reza Sadeghi. “Oops..! I Glitched It Again! How to Multi-Glitch the Glitching-Protections on ARM TrustZone-M.” In: *CoRR* abs/2302.06932 (2023). DOI: 10.48550/arXiv.2302.06932.

[Sch+16] David Schaffenrath, Markus Wegmann, Antonio Pullini, Davide Schiavone, Beat Muheim, Stefan Mangard Mangard, and Mario Werner. *Patronus ASIC*. http://asic.ethz.ch/2016/Patronus.html. [accessed 2023-01-01]. 2016.

[Sch+18a] Tao B. Schardl, Tyler Denniston, Damon Doucet, Bradley C. Kuszmaul, I-Ting Angelina Lee, and Charles E. Leiserson. “The CSI Framework for Compiler-Inserted Program Instrumentation.” In: *International Conference on Measurement and Modeling of Computer Systems – SIGMETRICS*. 2018, pp. 100–102. DOI: 10.1145/3219617.3219657.

[Sch+10] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. "ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software." In: *Computer Safety, Reliability and Security – SAFECOMP*. 2010, pp. 169–182. DOI: 10.1007/978-3-642-15651-9_13.

[SNM22a] Robert Schilling, Pascal Nasahl, and Stefan Mangard. *FIPAC LLVM Project*. https://github.com/Fipac/llvm-project. [accessed 2023-01-01]. 2022.

[SH08] Jörn-Marc Schmidt and Christoph Herbst. "A Practical Fault Attack on Square and Multiply." In: *Fault Diagnosis and Tolerance in Cryptography – FDTC*. 2008, pp. 53–58. DOI: 10.1109/FDTC.2008.10.

[SKP15] Florian Schroff, Dmitry Kalenichenko, and James Philbin. "FaceNet: A unified embedding for face recognition and clustering." In: *Computer Vision and Pattern Recognition Conference – CVPR*. 2015, pp. 815–823. DOI: 10.1109/CVPR.2015.7298682.

[SD15] Mark Seaborn and Thomas Dullien. "Exploiting the DRAM rowhammer bug to gain kernel privileges." In: *Black Hat* 15 (2015), p. 71.

[Sel+15] Bodo Selmke, Stefan Brummer, Johann Heyszl, and Georg Sigl. "Precise Laser Fault Injections into 90 nm and 45 nm SRAM-cells." In: *Smart Card Research and Advanced Applications – CARDIS*. 2015, pp. 193–205. DOI: 10.1007/978-3-319-31271-2_12.

[SGS23] SGS Brightsight. *Brightsight Security Lab in a Box (BSLB)*. https://www.brightsight.com/test-tools. [accessed 2023-01-01]. 2023.

[Sha07] Hovav Shacham. "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)." In: *Conference on Computer and Communications Security – CCS*. 2007, pp. 552–561. DOI: 10.1145/1315245.1315313.

[Sim+16] Jaehyeong Sim, Jun-Seok Park, Minhye Kim, Dongmyung Bae, Yeongjae Choi, and Lee-Sup Kim. "14.6 A 1.42TOPS/W deep convolutional neural network recognition processor for intelligent IoE systems." In: *International Solid-State Circuits Conference – ISSCC*. 2016, pp. 264–265. DOI: 10.1109/ISSCC.2016.7418008.

[SA02] Sergei P. Skorobogatov and Ross J. Anderson. "Optical Fault Induction Attacks." In: *Cryptographic Hardware and Embedded Systems – CHES*. 2002, pp. 2–12. DOI: 10.1007/3-540-36400-5_2.

[Sof23] Software Defined Automation. *Software Defined Automatio*. https://www.softwaredefinedautomation.io. [accessed 2023-01-01]. 2023.

[sta21] stacksmashing. *How the Apple AirTags were hacked*. https://youtu.be/_E0PWQvW-14. [accessed 2023-01-01]. 2021.

[Sta+10] François-Xavier Standaert, Olivier Pereira, Yu Yu, Jean-Jacques Quisquater, Moti Yung, and Elisabeth Oswald. "Leakage Resilient Cryptography in Practice." In: *Towards Hardware-Intrinsic Security - Foundations and Practice*. 2010. DOI: 10.1007/978-3-642-14452-3_5.

[Sta19] Standard Performance Evaluation Corporation. *SPEC CPU 2017*. https://www.spec.org/cpu2017. [accessed 2023-01-01]. 2019.

[STM22] STMicroelectronics. *STMicroelectronics STM32L476xx Datasheet*. https://www.st.com/resource/en/datasheet/stm32l476je.pdf. [accessed 2023-01-01]. 2022.

[Sug11] Makoto Sugihara. "A Dynamic Continuous Signature Monitoring Technique for Reliable Microprocessors." In: *IEICE Trans. Electron.* 94-C (2011), pp. 477–486. DOI: 10.1587/transele.E94.C.477.

[Sug20] Yuichi Sugiyama. *Pointer Authentication Support in Ibex*. https://mmxsrup.github.io/2020/08/31/gsoc2020-final-report.html. [accessed 2023-01-01]. 2020.

[Sul+17] Dean Sullivan, Orlando Arias, David Gens, Lucas Davi, Ahmad-Reza Sadeghi, and Yier Jin. "Execution Integrity with In-Place Encryption." In: *CoRR* abs/1703.02698 (2017). URL: http://arxiv.org/abs/1703.02698.

[Syn23] Synopsys Inc. *High-Reliability Design: No Room for Error*. https://www.synopsys.com/implementation-and-signoff/fpga-based-design/high-reliability.html. [accessed 2023-01-01]. 2023.

[TSS17] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. "CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management." In: *USENIX Security Symposium*. 2017, pp. 1057–1074. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang.

[Tat+18] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. "Throwhammer: Rowhammer Attacks over the Network and Defenses." In: *USENIX Annual Technical Conference*. 2018, pp. 213–226. URL: https://www.usenix.org/conference/atc18/presentation/tatar.

[Tem+16] Adam Teman, Davide Rossi, Pascal Meinerzhagen, Luca Benini, and Andreas Burg. "Power, Area, and Performance Optimization of Standard Cell Memory Arrays Through Controlled Placement." In: *ACM Trans. Design Autom. Electr. Syst.* 21 (2016), 59:1–59:25. DOI: 10.1145/2890498.

[Tex22] Texas Instruments. *Texas Instruments MSP430 Low-Power MCUs*. https://www.ti.com/microcontrollers-mcus-processors/microcontrollers/msp430-microcontrollers/overview.html. [accessed 2023-01-01]. 2022.

[The21] The kernel development community. *Seccomp BPF (SECure COMPuting with filters)*. https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html. [accessed 2023-01-01]. 2021.

[Tic+14] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM." In: *USENIX Security Symposium*. 2014, pp. 941–955. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice.

[Tim21] Niek Timmers. *Using Fault Injection to Turn Data Transfers into Arbitrary Execution*. https://niektimmers.com/research/2019-using-fault-injection-for-turning-data-transfers-into-arbitrary-execution-slides-1.0.pdf. [accessed 2023-01-01]. 2021.

[TM17] Niek Timmers and Cristofaro Mune. "Escalating Privileges in Linux Using Voltage Fault Injection." In: *Fault Diagnosis and Tolerance in Cryptography – FDTC*. 2017, pp. 1–8. DOI: 10.1109/FDTC.2017.16.

[TS16] Niek Timmers and Albert Spruyt. *Bypassing secure boot using fault injection*. https://www.blackhat.com/docs/eu-16/materials/eu-16-Timmers-Bypassing-Secure-Boot-Using-Fault-Injection.pdf. [accessed 2023-01-01]. 2016.

[TSW16] Niek Timmers, Albert Spruyt, and Marc Witteman. "Controlling PC on ARM Using Fault Injection." In: *Fault Diagnosis and Tolerance in Cryptography – FDTC*. 2016, pp. 25–35. DOI: 10.1109/FDTC.2016.18.

[Tor22] Linus Torvalds. *The Linux Kernel*. https://kernel.org. [accessed 2023-01-01]. 2022.

[TBC19] Thomas Trouchkine, Guillaume Bouffard, and Jessy Clédière. "Fault Injection Characterization on Modern CPUs." In: *Information Security Theory and Practice – WISTP*. 2019, pp. 123–138. DOI: 10.1007/978-3-030-41702-4_8.

[Tro+21] Thomas Trouchkine, Sebanjila Kevin Bukasa, Mathieu Escouteloup, Ronan Lashermes, and Guillaume Bouffard. "Electromagnetic fault injection against a complex CPU, toward new micro-architectural fault models." In: *Journal of Cryptographic Engineering* (2021), pp. 1–15.

[TMA11] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. "Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault." In: *Information Security Theory and Practice – WISTP*. 2011, pp. 224–233. DOI: 10.1007/978-3-642-21040-2_15.

[Var22] Raj Vardhman. *How many Linux users are there?* https://findly.in/how-many-linux-users-are-there. [accessed 2023-01-01]. 2022.

[Vas+20] Aurélien Vasselle, Hugues Thiebeauld, Quentin Maouhoub, Adèle Morisset, and Sébastien Ermeneux. "Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot-Extended Version." In: *IEEE Trans. Computers* 69 (2020), pp. 1449–1459. DOI: 10.1109/TC.2018.2860010.

[Vee+16] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms." In: *Conference on Computer and Communications Security – CCS*. 2016, pp. 1675–1689. DOI: 10.1145/2976749.2978406.

[VHM03] Rajesh Venkatasubramanian, John P. Hayes, and Brian T. Murray. "Low-Cost On-Line Fault Detection Using Control Flow Assertions." In: *International Symposium on On-Line Testing and Robust System Design – IOLTS*. 2003, pp. 137–143. DOI: 10.1109/OLT.2003.1214380.

[VKS11] Ingrid Verbauwhede, Dusko Karaklajic, and Jörn-Marc Schmidt. "The Fault Attack Jungle - A Classification Model to Guide You." In: *Fault Diagnosis and Tolerance in Cryptography – FDTC*. 2011, pp. 3–8. DOI: 10.1109/FDTC.2011.13.

[Vey+12] Nicolas Veyrat-Charvillon, Marcel Medwed, Stéphanie Kerckhof, and François-Xavier Standaert. "Shuffling against Side-Channel Attacks: A Comprehensive Study with Cautionary Note." In: *Advances in Cryptology – ASIACRYPT*. 2012, pp. 740–757. DOI: 10.1007/978-3-642-34961-4_44.

[WKK09] Zhen Wang, Mark G. Karpovsky, and Konrad J. Kulikowski. "Replacing linear Hamming codes by robust nonlinear codes results in a reliability improvement of memories." In: *Dependable Systems and Networks – DSN*. 2009, pp. 514–523. DOI: 10.1109/DSN.2009.5270297.

[Wat+20] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A. Patterson, and Krste Asanović. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.12-draft*. EECS Department, University of California, Berkeley, Aug. 29, 2020. URL: https://github.com/riscv/riscv-isa-manual/releases/download/draft-20200829-c159933/riscv-privileged.pdf.

[Wat+14] Andrew Waterman, Yunsup Lee, David A. Patterson, Krste Asanovic, Volume I User-level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. *The RISC-V Instruction Set Manual*. 2014.

[Wer+18] Mario Werner, Thomas Unterluggauer, David Schaffenrath, and Stefan Mangard. "Sponge-Based Control-Flow Protection for IoT Devices." In: *European Symposium on Security and Privacy – EuroS&P*. 2018, pp. 214–226. DOI: 10.1109/EuroSP.2018.00023.

[WWM15] Mario Werner, Erich Wenger, and Stefan Mangard. “Protecting the Control Flow of Embedded Processors against Fault Attacks.” In: *Smart Card Research and Advanced Applications – CARDIS*. 2015, pp. 161–176. DOI: 10.1007/978-3-319-31271-2_10.

[WP17] Nils Wiersma and Ramiro Pareja. “Safety != Security: On the Resilience of ASIL-D Certified Microcontrollers against Fault Injection Attacks.” In: *Fault Diagnosis and Tolerance in Cryptography – FDTC*. 2017, pp. 9–16. DOI: 10.1109/FDTC.2017.15.

[WS88] Kent D. Wilken and John Paul Shen. “Continuous Signature Monitoring: Efficient Concurrent-Detection of Processor Control Errors.” In: *International Test Conference – ITC*. 1988, pp. 914–925. DOI: 10.1109/TEST.1988.207880.

[WS90] Kent D. Wilken and John Paul Shen. “Continuous signature monitoring: low-cost concurrent detection of processor control errors.” In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 9 (1990), pp. 629–641. DOI: 10.1109/43.55193.

[WWM11] Jasper G. J. van Woudenberg, Marc F. Witteman, and Federico Menarini. “Practical Optical Fault Injection on Secure Microcontrollers.” In: *Fault Diagnosis and Tolerance in Cryptography – FDTC*. 2011, pp. 91–99. DOI: 10.1109/FDTC.2011.12.

[WP13] Hongjun Wu and Bart Preneel. “AEGIS: A Fast Authenticated Encryption Algorithm.” In: *Selected Areas in Cryptography – SAC*. 2013, pp. 185–201. DOI: 10.1007/978-3-662-43414-7_10.

[Wu+15] May Wu, Ravi Iyer, Yatin Hoskote, Steven Zhang, Julio Zamora-Esquivel, German Fabila Garcia, Ilya Klotchkov, and Mukesh Bhartiya. “Design of a low power SoC testchip for wearables and IoTs.” In: *Hot Chips Symposium – HOT CHIPS*. 2015, pp. 1–27. DOI: 10.1109/HOTCHIPS.2015.7477326.

[ZB19] Florian Zaruba and Luca Benini. “The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology.” In: *IEEE Trans. Very Large Scale Integr. Syst.* 27 (2019), pp. 2629–2640. DOI: 10.1109/TVLSI.2019.2926114.

[ZS13] Mingwei Zhang and R. Sekar. “Control Flow Integrity for COTS Binaries.” In: *USENIX Security Symposium*. 2013, pp. 337–352. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang.

[Zha+16] Yiqun Zhang, Kaiyuan Yang, Mehdi Saligane, David T. Blaauw, and Dennis Sylvester. “A compact 446 Gbps/W AES accelerator for mobile SoC and IoT in 40nm.” In: *Symposium on VLSI Circuits – VLSIC*. 2016, pp. 1–2. DOI: 10.1109/VLSIC.2016.7573553.

[ZPZ08] Zhengdao Zhang, Zhumiao Peng, and Zhiping Zhou. "The Study of Intrusion Prediction Based on HsMM." In: *Asia-Pacific Services Computing Conference – APSCC*. 2008, pp. 1358–1363. DOI: 10.1109/APSCC.2008.107.

[Zha+14] Zhi-Kai Zhang, Michael Cheng Yi Cho, Chia-Wei Wang, Chia-Wei Hsu, Chong Kuan Chen, and Shiuhpyng Shieh. "IoT Security: Ongoing Challenges and Research Opportunities." In: *International Conference on Service Oriented Computing and Applications – SOCA*. 2014, pp. 230–234. DOI: 10.1109/SOCA.2014.58.

[Zha+20] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine." In: *Fourth Workshop on Computer Architecture Research with RISC-V* (2020).

[ZHA15] Wenfeng Zhao, Yajun Ha, and Massimo Alioto. "Novel Self-Body-Biasing and Statistical Design for Near-Threshold Circuits With Ultra Energy-Efficient AES as Case Study." In: *IEEE Trans. Very Large Scale Integr. Syst.* 23 (2015), pp. 1390–1401. DOI: 10.1109/TVLSI.2014.2342932.

[ZAV04] Haissam Ziade, Rafic A. Ayoubi, and Raoul Velazco. "A Survey on Fault Injection Techniques." In: *Int. Arab J. Inf. Technol.* 1 (2004), pp. 171–186. URL: http://www.iajit.org/ABSTRACTS-2.htm%5C#04.

[Zim] Reto Zimmermann. "Efficient VLSI Implementation of Modulo ($2^n \pm 1$) Addition and Multiplication." In: *Symposium on Computer Arithmetic – ARITH 1999*, pp. 158–167. DOI: 10.1109/ARITH.1999.762841.